学位論文　博士（工学）

ウェブアプリケーションにおける
性能異常の検出と原因究明に関する研究

2011 年度

慶應義塾大学大学院理工学研究科

岩 田　聡

ウェブアプリケーションにおける
性能異常の検出と原因究明に関する研究

岩田　聡

論文要旨

近年，オンラインバンキングや証券取引など大きな責任を伴うサービスがウェブアプリケーションで提供されるようになってきた．しかし，システム障害が度々報告されているように，その信頼性は十分とはいえず，対策が求められている．システム障害の一つとして近年特に注目を集めているのが応答時間の増大やスループットの低下などの性能異常である．現在ウェブアプリケーションは非常に複雑になっており，性能異常の発生を事前に防ぐのは非現実的である．

本研究では，発生した性能異常から早期に復旧し，被害を最小限に抑えることを目的とする．そのためには異常発生をいち早く検出し，さらに原因究明に有益な情報を管理者へ提供する手法が必要になる．この異常検出，原因究明支援手法には，精度だけでなく導入する際のシステムへの変更を最小限に抑えることも要求される．なぜなら，近年システムの大規模化，複雑化が進み，システムに変更を加えるには専門的な知識と技術を要するためである．これまでは検出，原因究明の精度か導入の容易さかのどちらか一方を重視した手法が提案されてきており，2 点を共に満たす手法はなかった． 精度を重視した手法の例としてはコンポーネントごとに処理時間を監視する手法がある．ウェブアプリケーションはさまざまなソフトウェアコンポーネントを組み合わせて構成されており，即座に異常を絞り込むことは復旧にかかる時間の大きな短縮につながる． しかしその一方でコンポーネントごとに処理時間を監視する機構を導入するために，システムへの大きな変更を要する．

そこで，本研究では検出と原因究明の精度が高いだけでなく，対象システムへの導入が容易な手法を提案する．具体的には，処理時間の監視単位をリクエストの種類ごとにやや粗くすることで導入を容易にする．一般的なウェブサーバではリクエストごとに URL と処理時間を記録する機能が備わっており，対象システムへ変更を加えることなく，本手法を実現可能である．実際に Apache ウェブサーバでもこの機能を備えている．

一方で，比較的粗い監視情報から精度高く検出を行うために，管理図という既存の統計手法を応用する．ウェブアプリケーションの処理時間は一定ではなく，正常時でも揺らぎが生じる．この揺らぎと異常な変化の見極めを管理図の応用によって行う．管理図は過去のデータと現在のデータを統計的に比較し，分布が異なると判断した場合に警告を発する．過去のデータとして正常時のリクエスト処理時間を与え，運用時の処理時間と比較することで早期に性能異常を検出する．また，リクエストは種類ごとに異なるコンポーネントを用いて処理が行われるため，異常が発生しているリクエストの種類に注目することで，原因の絞込みを行うことができる．

さらに，本研究では複数の性能異常が同時に発生した場合にも対応する．異常を同時に複数検出した場合，それらが同じ原因により発生したものかどうかを判定することが求められる．同じ原因によるものだと判断できれば，そのリクエストの種類が共通して利用しているコンポーネントが原因である可能性が高い．逆に異なる原因によるものだと判断できれば，個々のリクエストの種類が個別に利用しているコンポーネントが原因の可能性が高い．

提案手法では，処理時間の変化傾向に着目することで，性能異常の原因が同じか異なるかを自動的に判定する．まず処理時間の分布の変化傾向を棒グラフの形式で抽出し性能異常のシグネチャとする．そして，シグネチャ同士を比較し類似度を計算する．最後に，この類似度を基に階層的クラスタリングの群平均法を用いてクラスタリングを行う．その後クラスタリング結果を参考にしながら管理者が原因究明を行う．リクエストの種類が同じクラスタに分類されれば，それらに発生している性能異常は同じ原因である可能性が高く，異なるクラスタに分類されれば，それらに発生している性能異常の原因は異なる可能性が高い．

提案手法を，オークションサイトを模したベンチマークウェブアプリケーション RUBiS に適用したところ，性能異常の早期検出と原因究明支援を行うことができた．クライアントの増加に伴い発生した性能異常を処理時間が約 100 ミリ秒増大した段階で検出できた．さらに，提案手法により得られた情報を利用し，異常の原因の絞込みをサーバ単位や，Java のクラス，メソッド単位で行うことができた．

Research for Detection and Diagnosis of Performance Anomalies in Web Applications

Satoshi Iwata

Abstract

Today, responsive services such as stock trading and online banking are becoming to be provided via web applications. However, they are not dependable enough and so, performance anomalies, such as increase of response time or decrease of server throughput, are often reported. It is required to recover a faulty system as quick as possible with early detection and diagnosis of anomalies, to prevent the service from getting severe damage.

Although several methods have been proposed to achieve the goal, none of them have fulfilled both of two major requirements. Not only should detection and diagnosis be done effectively, a method should be deployed easily to a target system. If not so, practical use of a method is hindered.

We propose a method which fulfills both requirements. Our method observes processing times with the granularity of request types, which can be differentiated via URL. As modern server software usually has a function to log URL and processing time of each request, our method can be applied easily to existing systems. For example, Apache web server has the functionality. Moreover, our method is effective to determine root causes, since we can narrow down root causes to components which have been used to process faulty request types.

To effectively detect and diagnose anomalies with coarse-grained monitoring, we take an advantage of statistical analysis. We apply *control charts* and monitor processing times. Control charts enable us to detect anomalies without being confused by natural fluctuations in processing time. Control charts compare current data to data which has been gathered during past normal operation. If control charts detect statistical differences between two distributions of data, they raise a warning. In our method, control charts monitor four types of statistics, i.e., average, maximum, median, and minimum of requests' processing time.

When several request types are simultaneously detected as faulty, we have to determine if their root causes are the same or not. Our method automates this task by clustering them based on the similarity in deviations from the non-anomalous distribution of measurements. Our clustering method involves three steps. First, we distill a *performance anomaly signature* from the processing time of requests. A performance anomaly signature characterizes how the "distribution" of the processing time has changed after an anomaly has occurred. Second, we calculate the *similarity* in the signatures. The similarity is a scalar that represents the degree to which two signatures, i.e., two bar graphs, overlap each other. Finally, we cluster anomalies based on the similarities. If two or more anomalies are clustered together, this implies that they are affected by the same root cause.

The results from case studies, which were conducted using an auction prototype *RUBiS*, are encouraging. The increases of processing times were around 100 milliseconds when our method detected anomalies. Afterwards, guided by the results of our clustering method, we determined suspicious components, such as server software, Java classes, or methods in a Java class.

目 次

図目次

表 目 次

第1章　序論

1.1　性能異常の検出と原因究明を行う手法の必要性

1.1.1　ウェブアプリケーション

現在ショッピングサイトや地図検索サービス，オンラインバンキングなどさまざまなサービスがウェブアプリケーションにより提供されている．ウェブアプリケーションとはウェブを利用したアプリケーションの総称である．今日では広く普及しており，我々の生活に欠かせないものとなっている．

図 1.1 に示すように，ウェブアプリケーションを利用するユーザは HTTP（Hyper Text Transfer Protocol）を利用してインターネット越しにサーバへリクエストを送信する．サーバ側ではリクエストに応じて処理が行われ，その結果をレスポンスとしてクライアントへ送信する．レスポンスを受け取ったクライアントはウェブブラウザ等を利用してレスポンスをユーザに表示する．

図 1.1 を用いて，サーバ側の動作を説明する．サーバ側は一般的にウェブ層，アプリケーション層，データベース層の 3 層構造によって構成されている．ウェブ層ではリクエストの受信およびレスポンスの送信だけではなく，静的なコンテンツの配信やアプリケーションサーバへの負荷分散を行う．アプリケーション層では動的なコンテンツの生成を，データベース層では永続的なデータの管理を行う．各層の働きを行うサーバアプリケーションをそれぞれウェブサーバ，アプリケーションサーバ，データベースサーバと呼び，リクエストの内容に応じてサーバ間をまたがり処理が行われる．各層はサービスの規模に応じて複数台のサーバから構成され，処理を分散しながら互いに協調して動作する．

さらに，各サーバ内では多数のソフトウェアコンポーネントが動作し，リクエスト内容に応じて適切なコンポーネントが処理を行う．ソフトウェアコンポーネントの例としてはウェブページを記述した HTML（HyperText Markup Language）ファイルや Java Platform, Enterprise Edition [1] 環境で用いられる Servlet や EJB（Enterprise Java Beans）などの Java のクラス，JDBC ドライバと呼ばれる Java と

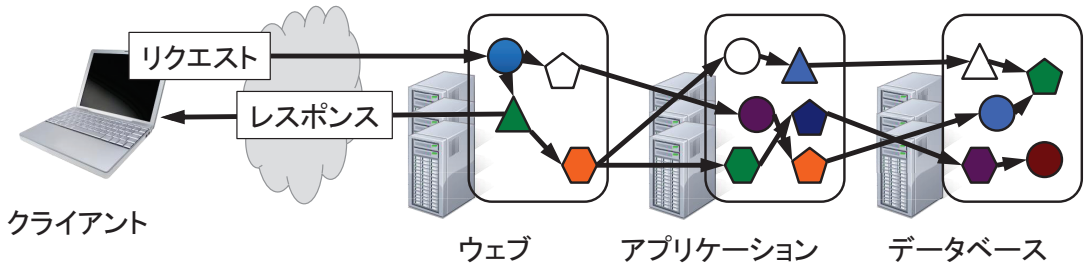


図 1.1: ウェブアプリケーション提供基盤の一般的な構成

データベースを接続するためのドライバなどがある．

1.1.2 ウェブアプリケーションにおける性能異常

近年，ウェブアプリケーションにより高い信頼性が求められるサービスを提供するようになってきた．高い信頼性が求められるサービスとしては，オンラインバンキングや証券取引，ショッピングなど金銭の授受を伴い，サービスの不具合発生時に損害賠償が発生するようなサービスなどがある．

しかしその一方で，現在でもシステム障害が度々報告されているように，ウェブアプリケーションの信頼性は十分に高いとは言えず，今後も継続して信頼性を向上していく必要がある．例えば，インターネットの検索エンジンやウェブメールのサービスを提供している Google [2] において 2009 年に発生した障害では，Google の提供する全てのサービスでユーザの約 14%にサービスの遅延もしくは中断を引き起こした [3]．また，過去に YAHOO! JAPAN [4] のオークションサービスや Amazon Web Services [5] が提供するクラウドコンピューティング環境 Amazon EC2（Elastic Compute Cloud）でも障害が報告されている．YAHOO! JAPAN のオークションサービスでは 2010 年に一部のユーザがオークションにアクセスできない等のシステム障害が発生した [6]．Amazon EC2 では 2011 年に障害が発生し，その上で稼働するサービスが利用できなくなったり，制限を受けるなどした [7]．さらに，楽天ネット証券 [8] では 2010 年 10 月 1 日〜2011 年 9 月 30 日の 1 年間に 5 回のシステム障害が発生し，合計 299 分間，正常にサービスを提供できなかったと報告されている [9]．

ウェブアプリケーションの信頼性を阻害する要因は複数あるが，その一つが**性能異常**である．この性能異常とはサービスの性能が想定した値に比べて著しく劣

化することを指し，リクエスト応答時間の増大やサーバスループットの低下などがある．具体例としては楽天ネット証券で 2010 年 10 月 5 日に発生した性能異常がある．この性能異常では処理速度が低下し，画面表示やログインに通常よりも長い時間がかかるといった事象が発生したと報告されている [9]．また，この性能異常を復旧するにはプログラムの改修を行う必要があったことも報告されている．

性能異常の原因は，大きく分けて次の 3 種類である．一つ目がソフトウェアのバグである．例えば，バグによりメモリリークが発生した場合，メモリの枯渇につれてスワップ領域へのデータ待避が進む．その結果，メモリに比べてアクセスが低速なハードディスクへのアクセス回数が増加し，性能異常が発生することがある．

二つ目はハードウェアの故障である．現在のシステムは負荷分散のために複数台のマシンから構成されていることが多い．そのため，数台のマシンが故障してもサービスを継続することは可能である．しかし，処理を行うマシンの台数が減少するため，性能異常が発生することがある．また，ネットワークが分断した場合も性能異常が発生する．マシン故障の場合と同様に，処理を行うサーバの台数が減少してしまうためである．

三つ目は管理者の設定誤りである．近年のサーバシステムは非常に複雑になっており，さまざまな設定を管理者が適切に行う必要がある．例えば，複数台のマシンへの負荷分散を行う際は，各マシンの能力と用途に合わせて適切にリクエストの配分を行う必要がある．もしこの設定が誤っていた場合，マシンの能力を超えた数のリクエストが配分され，性能が低下する可能性がある．その他には，データベースのインデックス作成の例が挙げられる．データベースでは検索速度向上のために，あらかじめインデックスを作成しておくことがある．テーブル内のデータが少ないときには，インデックスを作成していなかったとしても，検索速度はさほど大きな問題にはならない．しかし，サービスの提供が続きデータ量が増加するにつれて，インデックスの不足が性能異常を引き起こしてしまうことがある．

性能異常への対処が近年，研究対象として注目を集めている一方，これまではシステム停止による障害への対処を中心に研究が行われてきていた．その理由は大きく二つ挙げられる．一つ目に，性能異常による障害に比べてシステム停止による障害の方が被害が大きいということである．性能異常が発生した場合，処理が行われないリクエストは一部だけであり，その他のリクエストに関しては遅延が発生する可能性があるものの，処理自体は正常に完了することができる．それに対して，システムが停止してしまえば全てのリクエストの処理が停止する．二つ

目に，性能異常に比べてシステム停止の方が障害としては比較的単純だということが挙げられる．その結果，比較的扱いやすいシステム停止が先に研究対象にされてきた．システム停止の場合，何か異常な処理内容がシステムの停止を引き起こすことが多く，多くの場合，適切なエラーメッセージを出力することができる．一方性能異常の場合，処理自体は正しく行われるため，異常が発生しているかどうかを見極めることすらも難しい．しかし，性能異常もサービス提供者に大きな被害をもたらすことが知られており，対策が急務になっている．例えば，Linden [10] は Amazon [11] が提供しているサービスの応答時間が 100 ミリ秒増大すれば売上が 1%低下し，Google が提供しているサービスの応答時間が 500 ミリ秒増大すればトラフィック量が 20%減少するだろうと予測している．

1.1.3　性能異常の早期検出と原因究明支援の必要性

システムの信頼性を向上させるためには性能異常の発生を防ぐことが理想であるが，現状では難しい．その理由は，大きく次の四つに分けられる．一つ目は，ウェブアプリケーションを提供するシステムが巨大であること．二つ目は，ウェブアプリケーションを提供するシステムが複雑であること．三つ目は，サイトごとに適切な運用が異なるために，他サイトのノウハウがそのまま適用できないということ．そして四つ目は，システムとそれを取り巻く環境が変化し続けるため，性能異常を引き起こす要因が新たに加わる危険性が常に存在することである．

まず，近年ウェブアプリケーションを提供するシステムは非常に巨大である．例えば，Apache HTTP サーバ [12] の配布パッケージに含まれるファイルの総行数は約 85 万行，JBoss アプリケーションサーバ [13] では約 58 万行に及ぶ．システムが巨大になれば性能異常を引き起こすバグが混入する可能性が高くなる．全てのプログラムや設定ファイルを適切に開発することが難しい．

また，複雑なシステムほど適切に実装するのが難しくなり，性能異常を引き起こす要因が混入する可能性が高くなる．近年ではウェブアプリケーションで複雑な処理を行うことは一般的である．一つのリクエストの処理を行うために複数のコンポーネントを呼び出すことは珍しくない．例えば，一つのモジュールはさまざまな種類のプログラムから呼び出される可能性がある．従って，連携する可能性のある全てのプログラムに対して適切に動作するように実装し，動作確認を行う必要がある．

性能異常を未然に防ぐことが困難な理由は他にも存在する．まず，サイトごと

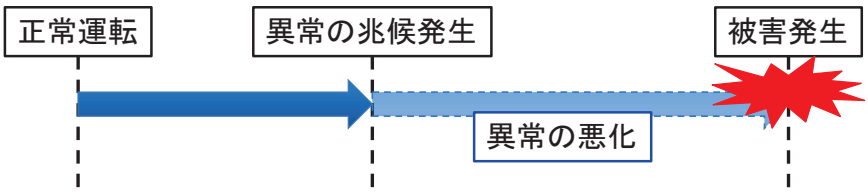
図 1.2: 性能異常からの回復を行う手順

に適切な運用方法が異なるために，他サイトのノウハウがそのまま適用できない．サイトごとに提供するサービスが異なるためウェブアプリケーションの特性も異なる．例えば，地図検索を提供するサービスと証券取引を行うサービスでは，必要な画像の量が異なる．そのため，サーバの各種性能パラメータの適切な値は異なる可能性が高い．表示する画像が多い場合，一つのリクエストに対してたくさんのレスポンスが発行される．よって，再利用できるように，コネクションを比較的長時間維持する必要がある．次に，マシンの能力もサイトごとに異なる．そのため，適切なサーバの台数や負荷分散の設定がサイトごとに異なってくる．

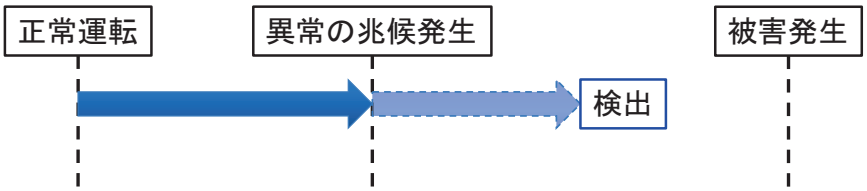
もう一つの理由は，システムとそれを取り巻く環境が変化し続けるため，性能異常を引き起こす要因が新たに加わる危険性が常に存在することである．サービスに人気が出てサイトへのアクセス数が増加したり，人気のあるコンテンツが変わりリクエストの傾向が変化したりするなど，ワークロードは変化する．アプリケーションへの機能追加やサーバソフトウェアの更新など，システムをアップデートする際には新たなバグが混入する危険がある．サービスへの負荷状況に応じてマシンを追加したり，メンテナンスのためにマシンをシステムから離脱させたりする際は負荷分散の設定を適切に設定し直す必要がある．性能異常の発生を未然に防ぐことが困難であることは，Patterson ら [14] によっても指摘されている．

そこで，性能異常が発生した際にシステムを修復し，性能異常から回復することが求められる．システムを修復するためには図 1.2 に示すように，まず異常の検出を行い，次に原因の究明を行う必要がある．このため，性能異常の早期検出手法と原因究明支援手法が注目を集めている．

まず，性能異常を早期に検出する必要がある．図 1.3(a) のように，通常，システムが正常に稼働している状態から，突然，被害が発生するほどの性能異常が発生することはほとんどない．何かしらの異常の兆候が発生し，それに気づかずにシステムをそのまま稼働し続けることで異常が悪化し，その異常がシステムに被害を及ぼすまでに大きなものになってしまう．そこで図 1.3(b) のように，異常の兆



(a) 性能異常による被害が発生する例



(b) 性能異常による被害が発生する前に早期検出できた例

図 1.3: 性能異常の早期検出の有用性

候が発生している段階で異常を検出することが必要である．

性能異常検出後は，管理者が行う原因究明を支援する手法が必要になる．システムが巨大かつ複雑で，さらに性能異常の原因が多岐にわたるため，原因究明を人手のみで行うのは困難である．処理時間が異常な値になっているコンポーネントやボトルネックになっている資源を管理者に提示する手法が一般的である．

1.2 既存手法の問題点

性能異常の早期検出と原因究明支援を行う手法に求められる要件は二つある．一つ目は，検出と原因究明の能力である．検出手法は，性能異常が発生した際になるべく早い段階で検出できることが求められる．原因究明支援手法は，管理者に対して原因究明を行うために有用な情報を提供することが求められる．二つ目は，手法導入の容易さである．提案手法導入の際にシステムへの修正を行う必要がない手法が理想的である．また，もし修正が必要だった場合でも，その修正を最小限に抑える必要がある．現在，ウェブアプリケーション自体とそれを提供する基盤は複雑かつ巨大であり，それらの修正には大きなコストを要する．監視機構を導入する場合，システム内部やアプリケーションの動作を熟知したプログラマが必要になる．また，たとえそのようなプログラマを用意することができたとして

表 1.1: 二つの要件に注目した既存手法のまとめ

	コンポーネント	資源使用率	ログ	アクセス数
能力	◎	×	○	×
導入の容易さ	×	○	×	○

も，プログラムを変更する際，新たにバグを混入し障害の要因を追加してしまう可能性がある．さらに，修正を行うために時間を費やし，サービスの提供を妨げてしまうことも考えられる．こうした問題は修正箇所が多くかつ複雑であればあるほど，顕著になる．

ここでは，この二つの要件に注目しながら既存の性能異常検出，原因究明支援手法について述べる．性能異常の検出と原因究明を行うためには，処理時間や資源使用率など，システムの性能指標を監視する必要がある．本論文では，監視粒度に着目して，既存手法を四つのグループに分類する．そして，それぞれについて要件を達成しているか確認する．それぞれ，コンポーネント単位での監視を行う手法，資源使用率の監視を行う手法，ログの監視を行う手法，アクセス数の監視を行う手法である．

先の四つの性能異常検出，原因究明支援手法を二つの要件に注目してまとめたのが表 1.1 である．二つの要件である検出，原因究明の能力と手法導入の容易さを両方ともに満たす手法が存在しないことが分かる．次では，それぞれについて詳しく説明する．

1.2.1 コンポーネント単位での監視を行う手法

ウェブアプリケーションは図 1.4 で模式的に表しているように，多数のソフトウェアコンポーネントから構成される．丸や三角など，色のついた図形を用いてソフトウェアコンポーネントを模式的に表している．このとき，リクエストの処理は矢印で示したように，さまざまなソフトウェアコンポーネントを利用して行われる．

このソフトウェアコンポーネントに対して個々に監視機構を導入する手法が提案されている [15–18]．図 1.4 において赤い四角で模式的に表しているのがモニタリング機構であり，全てのソフトウェアコンポーネントに実装されている．そし

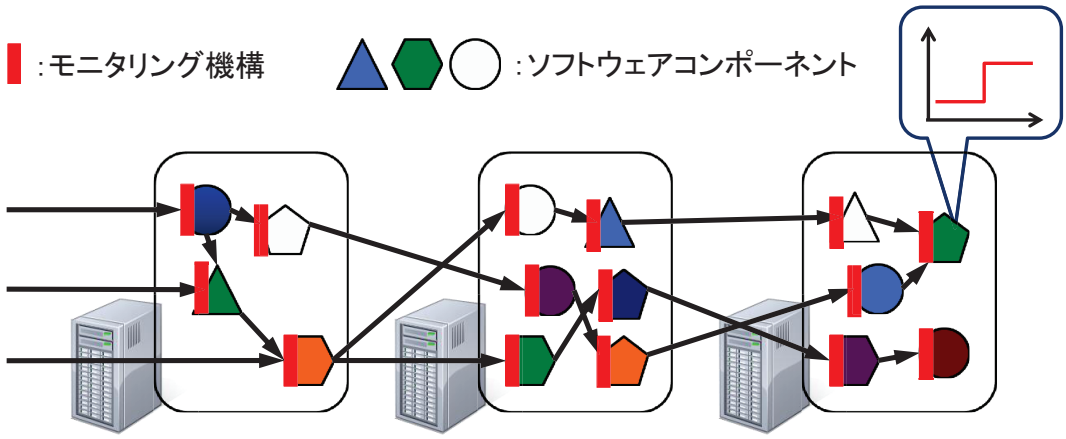


図 1.4: ソフトウェアコンポーネント単位での監視を行う手法

て，導入した監視機構で個々のコンポーネントでの滞在時間や処理に要した CPU 使用率を測定する．1 番右にある緑の五角形で表したソフトウェアコンポーネントのように，処理時間などの性能指標をグラフ等の形式で出力する．

このように細粒度で監視を行うことで，性能異常発生時には即座に原因となっているコンポーネントを絞り込むことができ，復旧にかかる時間を大幅に短縮することができる．このため，検出と原因究明の能力が高い手法であるといえる．

しかしその一方で導入の容易さは低い．なぜならば監視機構導入のためにシステムに大きな変更を要するからである．アプリケーションか，もしくはサーバを改変してコンポーネントごとの処理時間を測定可能にする必要がある．アプリケーションを改変する場合はもちろん，サーバを改変する場合でも大きな修正が必要になる．ウェブアプリケーションを提供する際，複数の種類のコンポーネントを利用することが一般的であるため，その種類ごとにソースコードの修正を行う必要がある．

1.2.2 資源使用率の監視を行う手法

資源使用率を統計的に解析し，異常な値を取っている指標を検出する手法が提案されている [19]．ここでいう資源使用率とは CPU やメモリの使用率，ネットワークやディスク I/O の発行量などである．一般的なオペレーティングシステムではマシン全体の資源使用率を監視する機構が用意されている場合がほとんどである．そのため，この手法は導入が容易であるといえる．

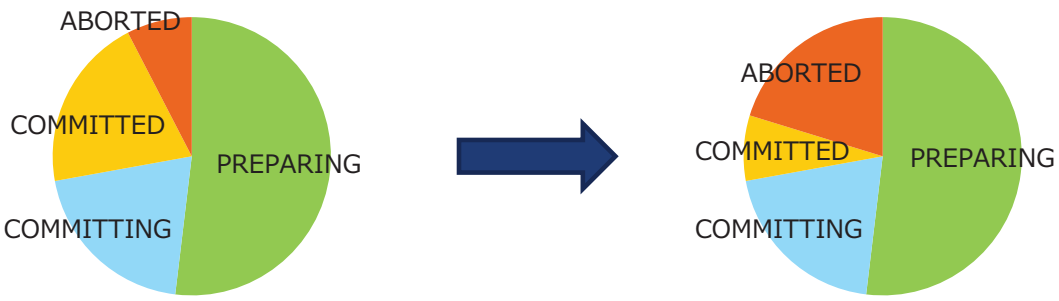


図 1.5: ログの出力割合を表した円グラフ

しかしその一方で検出と原因究明の能力は低くなっている．なぜならばマシン単位で監視した資源使用率と原因究明の間には大きな隔たりがあるからである．例えば，CPU 使用率が異常な値を示していると検出したとしても，その原因がシステムのどの部分にあるかを突き止めることは容易ではない．

1.2.3 ログの監視を行う手法

サーバが出力するログを統計解析し，ログの異常な出現パターンを検出する手法が提案されている [20]．一部のサーバでは各コンポーネントが処理ステータスをログに出力するよう実装されていることがある．図 1.5 の例は出力割合を表した円グラフであり，四つのステータス `PREPARING`，`COMMITTING`，`COMMITTED`，`ABORTED` が出力されている．図 1.5 では `COMMITTED` と `ABORTED` の出現割合が変化しており，サーバ内部で何かしらの異常が発生している可能性が高いことがわかる．この手法は検出と原因究明の能力は高い．なぜなら，ログは各コンポーネントの内部状態を表す有力な情報であり，処理が成功したか失敗したか，現在リトライしているかなどといったことが分かる．

しかし，導入の容易さは低くなっている．なぜならば，詳細なログを出力する機構を備えているサーバにしか適用できないからである．例えば，一般的なウェブサーバである Apache HTTP サーバ [12] は詳細なログを出力しないことが知られており，ログに着目した既存研究 [20,21] でもその対象は Darkstar オンラインゲームサーバ [22] や分散処理フレームワークの Hadoop [23] に限られている．また，詳細なログを出力するように既存のサーバを修正することは容易に行えない．ログはプログラムの作成者が，サーバの動作を熟知し，有用だと感じた変数を出力す

表 1.2: 提案手法と既存手法の比較

	コンポーネント	資源使用率	ログ	アクセス数	提案手法
能力	◎	×	○	×	○
導入の容易さ	×	○	×	○	○

る必要がある．

1.2.4 アクセス数の監視を行う手法

リクエストの種類ごとにアクセス数を監視し，その変化を検出する手法が提案されている [24]．性能異常の影響はアクセス数に現れることがある．例えば，性能異常により，あるページが表示されづらい場合，ユーザが更新を繰り返しアクセス数が増加することがある．

この手法は導入が容易である．なぜなら，多くのサーバではリクエストの URL をログに保存するためである．実際に Apache HTTP サーバではそのような機構を備えている．このログからリクエストの種類ごとのアクセス数を得ることができる．一方，この手法の検出，原因究明能力は低い．なぜなら，アクセス数が変化しているということはユーザに異常が感知されているということであるため，異常の早期検出を行えていないからである．

1.3 本研究の提案

本論文では，検出と原因究明の能力が高いだけでなく導入も容易な性能異常の検出，原因究明手法を提案する．本論文の提案手法と既存手法の比較を行ったのが表 1.2 である．提案手法はコンポーネント単位の監視に近いアプローチを採用する．ただし，監視粒度をやや粗くすることで導入を容易にする．さらにその一方で，取得したデータを統計的に解析し，検出と原因究明の能力を高く保つ．本節では提案手法の概要を述べながら，提案手法が二つの要件を満たしていることを示していく．

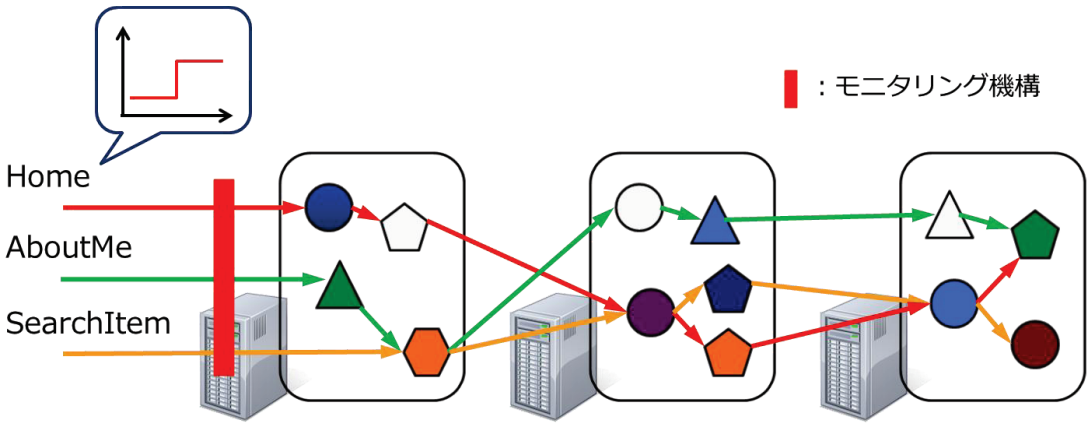


図 1.6: 提案手法における監視単位

1.3.1 監視粒度

まず，提案手法では図 1.6 のように，リクエストの種類ごとに処理時間を監視する．通常クライアントは目的とする処理に応じて異なるリクエストをサーバへ送信する．例えばオークションサイトであれば，ホームのページを表示する `Home` というリクエストの種類やユーザの登録情報を確認する `AboutMe` というリクエストの種類，商品を検索する `SearchItem` などがある．このリクエストの種類は URL などを用いて分類可能である．URL が異なれば，利用する HTML ファイルや Servlet が異なってくる．赤い四角で模式的に表したモニタリング機構がリクエストの入り口に実装されている．また，1 番左にある `Home` というリクエストの種類のように，処理時間をリクエストの種類ごとに計測する．

リクエストの種類ごとに監視を行うことで，提案手法は導入の容易さを達成している．通常，サーバはリクエストごとに URL と処理時間をログに記録する機構を備えている．そのため，このログを利用すれば，システムへの変更を行わずにリクエストの種類ごとに処理時間を取得することができる．必要なのは，ログを解析するスクリプトのみである．実際に Apache HTTP サーバは URL と処理時間をログに取得する機構を備えている．

また，もし利用しているサーバに上記のようなリクエスト処理時間を計測する機構が実装されていなかった場合でも，本手法はシステムへの導入が容易だといえる．なぜならば，新たな監視機構の実装がリクエストの入り口 1 箇所に限定できるからである．コンポーネント単位に監視機構を実装する手法に比べ，システム内部で動作を詳細に把握すべき場所が限られるため，実装前の調査に費やす時

間が短縮できる．また，ソースコードの修正にかかる時間も短縮でき，導入を早く行うことができる．さらに，修正すべきソースコードが少ないため，バグが混入する可能性も削減できるという利点もある．

1.3.2 性能異常の検出

提案手法は，リクエストの種類ごとに取得した処理時間を**管理図** [25] を利用して監視し，性能異常を検出する．管理図は過去に取得した正常時のデータの分布と現時点で観測したデータの分布を統計的に比較し，分布に生じる変化を検出する手法である．この管理図を利用し，早期かつ正確な異常検出を達成する．処理時間はシステムが正常に稼働している際もスケジューリングや資源への競合などの不確定要因により揺らぐことが知られている．そこで，管理図を用いて，計測した処理時間から，正常時の揺らぎと性能異常の兆候となる変化を見分ける必要がある．

提案手法では処理時間をリクエストの種類という粗い粒度で監視するため，管理図を用いて分布の変化を統計的に検出する必要がある．コンポーネントごとの監視では，計測した処理時間に寄与するのが個々のコンポーネントに限られるため，性能異常が顕著に表れる．実際に，既存手法では計測したデータをグラフなどの形式で管理者へ表示することまでを行い，その後の判断は管理者に委ねる．一方，監視粒度が粗ければ，複数コンポーネントが監視データへ寄与することになり，変化が不明瞭になる．図 1.6 で `Home` というリクエストの種類で得られる処理時間は，処理の流れを表す赤い線の上に存在する六つのコンポーネントの処理時間の合計ということになる．

1.3.3 性能異常の原因究明

提案手法はリクエストの種類ごとに処理時間を監視することで原因究明に有用な情報を提供することができる．まず，性能異常の原因となっているコンポーネントを絞り込むことができる．異常を検出したリクエストの種類だけが利用するコンポーネントが原因である可能性が高いからである．図 1.7 は `Home` だけで性能異常を検出した例である．`Home` の処理時間が異常なため，`Home` が利用するコンポーネントに異常が発生している可能性が高い．また，`AboutMe` と `SearchItem` では処理時間が正常なため，この二つのリクエストの種類が利用するコンポーネ

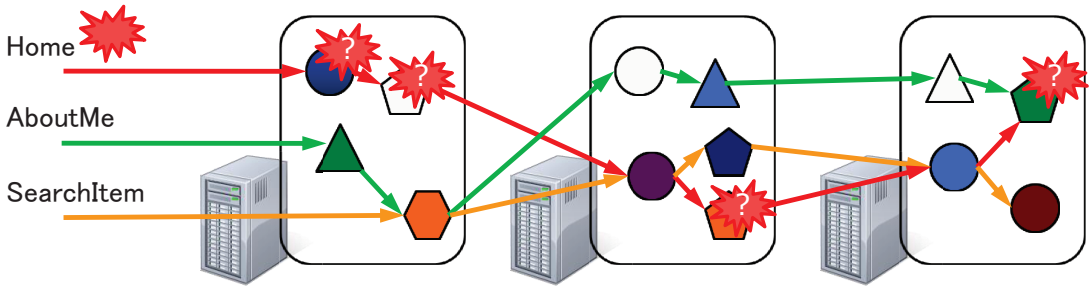


図 1.7: 原因コンポーネント絞り込みの例（Home だけで異常を検出した場合）

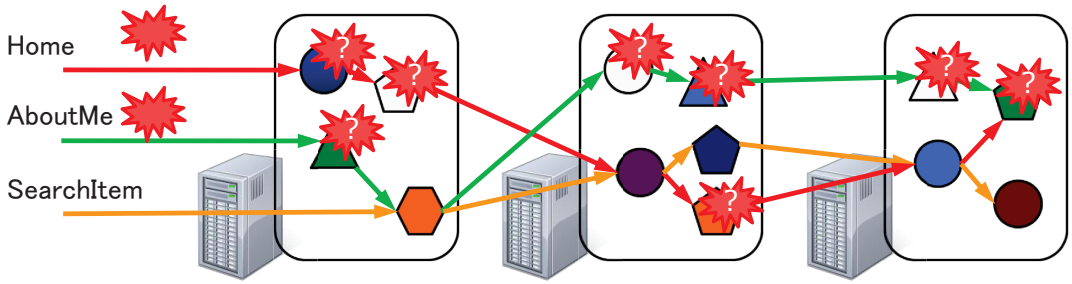


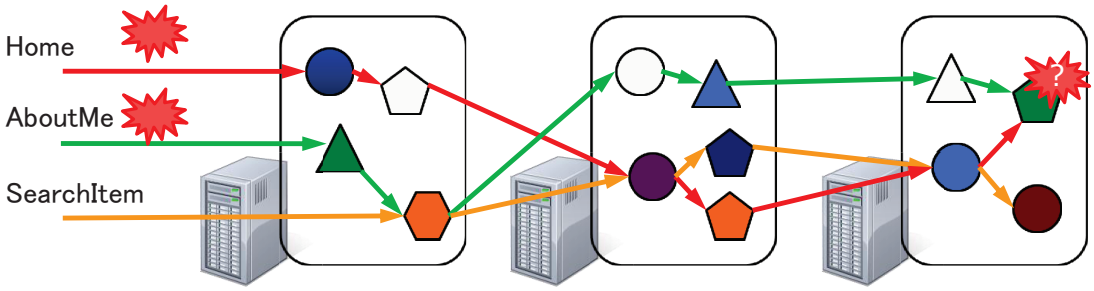
図 1.8: 性能異常分類手法を利用しない場合に原因コンポーネントを絞り込む例（Home と AboutMe 両方で異常を検出した場合）

ントは正常である可能性が高い．結果として，図 1.7 で疑問符を付した四つのコンポーネントが原因である可能性が高いと分かり，原因を絞り込むことができる．

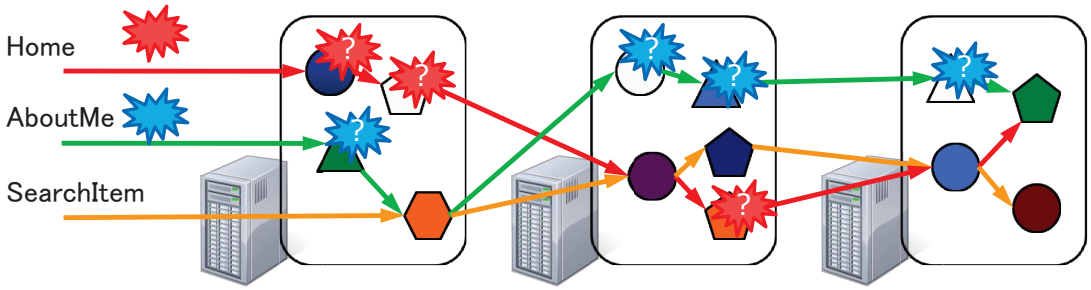
さらに原因コンポーネントを絞り込むために，本論文では性能異常分類手法という要素技術を提案する．性能異常分類手法を提案することで，監視の粒度が粗いにも関わらず，原因究明を細かい粒度で行うことができる．

性能異常分類手法は，性能異常が複数のリクエストの種類に同時に発生した場合に有効である．図 1.8 において，Home と AboutMe 二つのリクエストの種類で同時に性能異常を検出した場合を考える．このとき，性能異常分類手法を利用せずに原因として絞り込めるのは，疑問符を付した八つのコンポーネントである．なぜならば，現在得ているのは，性能異常を検出していない SearchItem が利用しているコンポーネントが正常に動作している可能性が高いという情報だけだからである．

性能異常分類手法は，性能異常を検出したリクエストの種類間で，異常の原因が同じか異なるかを管理者に提示する．このことを図 1.9 を用いて説明する．図 1.9(a)



(a) 発生した性能異常の原因が同じ例



(b) 発生した性能異常の原因が異なる例

図 1.9: 性能異常分類手法を利用した場合に原因コンポーネントを絞り込む例（Home と AboutMe 両方で異常を検出した場合）

のように，もし二つのリクエストの種類で発生している性能異常の原因が同じだと分かれば，両者が共通して利用しているコンポーネントが原因である可能性が高い．よって，疑問符を付した一つのコンポーネントに原因が絞り込める．逆に図 1.9(b) のように，もし二つのリクエストの種類で発生している性能異常の原因が異なると分かれば，各々のリクエストの種類だけが利用しているコンポーネントがそれぞれ原因である可能性が高い．疑問符を付した七つのコンポーネントに原因が絞り込める．赤い記号の上に疑問符を付したコンポーネントが Home に発生している性能異常の原因，青い記号の上に疑問符を付したコンポーネントが AboutMe に発生している性能異常の原因である可能性が高い．

提案手法は上記のように，複数のリクエストの種類に同時に発生した性能異常を自動的に原因ごとに分類する．性能異常発生前後での処理時間の分布の変化傾向を統計的に解析し，この目的を達成する．具体的には，性能異常発生前後それ

ぞれでリクエスト処理時間の累積分布関数（CDF）を作成し，二つの CDF の形の差を性能異常の特徴として抽出する．この抽出した特徴が似ていれば，それらのリクエストの種類は同じ原因により性能異常が発生していると判断する．

1.4 本論文の構成

本論文は全 6 章からなる．以下，本論文は次のように構成されている．まず第 2 章では本研究と関連のある既存研究について述べる．ウェブアプリケーションにおける性能異常への対策として，検出と原因究明を行う手法だけでなく，発生そのものを未然に防ぐ手法についても述べる．加えて，ウェブアプリケーションにおけるシステム停止への対策手法，さらにはウェブアプリケーション以外のシステムでの障害対策手法についても述べる．

次に，第 3 章と第 4 章で提案手法を詳細に説明する．第 3 章では提案手法で必要になる要素技術の一つである**管理図を用いた性能異常検出手法**を説明する．第 4 章では，もう一つの要素技術である**性能異常を原因ごとに分類する手法**について詳しく説明する．提案手法では，まず管理図を用いて性能異常を検出する．その後，性能異常を原因ごとに分類することで，管理者が行う原因究明を支援する．

そして，第 5 章では提案手法が性能異常からの早期復旧に有用であることを示すために，ケーススタディを報告する．ケーススタディでは，管理図を用いて早期かつ正確に性能異常を検出できたこと，さらに性能異常分類手法を利用して原因コンポーネントを絞り込むことができたことを示す．第 5 章では実験手順を述べた後に，三つのケーススタディを報告し，その後性能異常分類手法についての議論を加える．

最後に，第 6 章で本論文をまとめ，さらに本研究の今後の展望について述べる．

第2章　関連研究

本章では，本研究と関連する研究を大きく 4 種類に分けて説明する．それぞれの位置関係を図 2.1 に示す．本論文では，コンピュータシステム全般の障害対策手法を調査した．そのうち，本論文の対象であるウェブアプリケーションについては，特別に詳しく分けて説明する．まず，ウェブアプリケーションにおける障害を，動作異常とシステム停止，性能異常に分けて説明する．さらに，性能異常への対策方法として，発生を未然に防ぐ手法，検出と原因究明を行う手法に分ける．

まず，第 2.1 節で本論文で取り上げる関連研究の全体像を示す．そうすることで，本論文と既存研究の関連を明確にする．その後，第 2.2 節と第 2.3 節，第 2.4 節では個々の既存研究について簡単に紹介する．

2.1　全体像

2.1.1　性能異常の検出と原因究明を支援する手法

まず，本研究と同じ目的を持つ手法である，性能異常の検出と原因究明支援を行う手法についてまとめる．第 1.2 節で説明したように，性能異常の早期検出と原因究明支援を行う既存手法は，検出と原因究明の能力か導入の容易さのどちらか一方のみを満たすものになっている．第 1.2 節同様，4 種類に分けて説明を行う．図 2.1 における赤い領域に当たる．

(1) コンポーネント単位での監視を行う手法

ウェブアプリケーションはさまざまなソフトウェアコンポーネントから構成されている．ここでいうソフトウェアコンポーネントとは，例えば，HTML ファイルや Servlet，EJB を指す．各コンポーネント単位で監視を行い，性能異常を検出することができれば，即座に異常なコンポーネントを絞り込むことができる．よって，この手法は検出と原因究明の能力が高いといえる．しかし，各コンポーネントに監視機構を実装する必要があるため，導入が容易であるといえない．

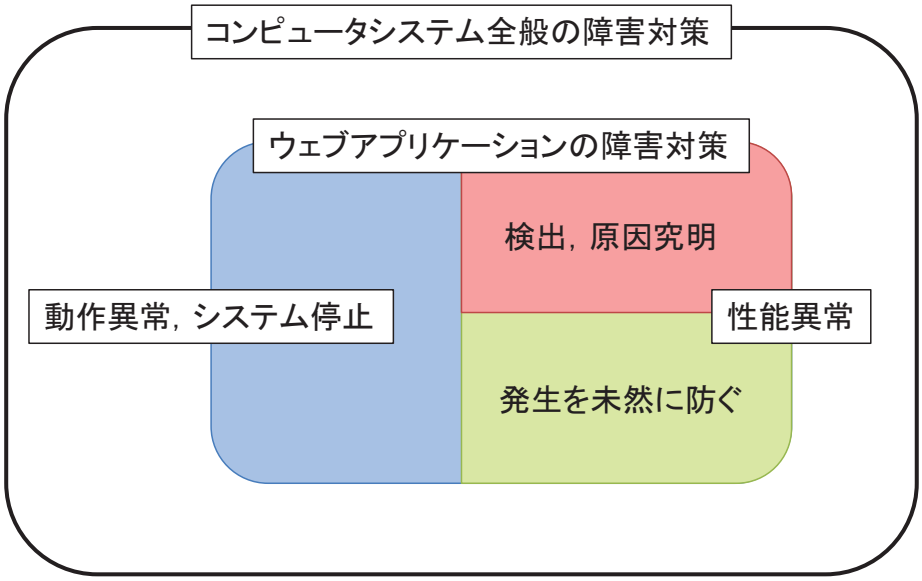


図 2.1: 関連研究同士の位置関係

(2) 資源使用率の監視を行う手法

資源使用率を監視することで，性能異常を検出する手法が提案されている．資源使用率は，マシンの負荷状況を知る上で重要な情報である．資源使用率が正常時と異なる値を示すということは，マシンの負荷が通常と異なることを表している．そのような場合，性能異常が発生する可能性が高い．

資源使用率を監視する手法は，システムへの導入が容易な一方，検出と原因究明の能力は低い．マシン単位の資源使用率を監視する機構はオペレーティングシステムによってすでに提供されているため，導入は容易である．しかし，資源使用率によって得られる情報と原因究明の隔たりは大きく，十分な原因究明支援を行うことができない．例えば，CPU 使用率が異常に大きな値となっていることを検出したとしても，その原因がシステム内部のどこに存在するかを突き止めることは容易ではない．

(3) ログの監視を行う手法

ウェブアプリケーションを提供するためのサーバの中には，処理状況の詳細な内容をログへ出力するものがある．ログに出力された詳細な情報を利用して，性能異常の検出と原因究明支援を行う手法が提案されている．

ログの監視を行う手法は，検出と原因究明の能力は高いものの，導入は容易だ

といえない．ログに出力される情報はサーバの処理状況を詳細に知ることができるため，検出と原因究明にとって非常に有用な情報となる．しかし，全てのサーバに詳細なログを出力する機構が備わっているわけではなく，導入が容易だとはいえない．また，ログを出力する機構を新たにサーバに導入するのは困難である．システムを詳細に把握し，適切な箇所で適切な情報を出力する必要がある．

(4) アクセス数の監視を行う手法

各ページへのアクセス数を監視し，性能異常の検出と原因究明支援を行う手法が提案されている．アクセス数が異常に変化した場合，そのページで性能異常が発生している可能性が高い．例えば，アクセス数が急激に増加した場合，ページの読込に時間がかかり，ユーザが更新を繰り返している可能性がある．

アクセス数を監視する手法はシステムへの導入が容易である反面，検出と原因究明の能力は低い．通常，サーバは各リクエストの URL をログへ記録する機構を備えており，そのログからアクセス数を取得することができる．しかし，アクセス数に変化が現れているということは，性能異常がすでにユーザに感知されているということである．検出を早期に行えているとはいえない．

2.1.2 性能異常の発生を未然に防ぐ手法

次に，性能異常の発生を未然に防ぐ手法についてまとめる．これらの手法を用いて，完璧に性能異常の発生を防ぐことができれば，本研究のように性能異常の検出と原因究明支援を行う必要がなくなる．しかし，それは現状では難しい．それを示すことで，本研究のように異常発生後の対処を迅速に行う手法の必要性を示す．図 2.1 における緑の領域に当たる．

現状では，これらの手法で性能異常の発生を完璧に防ぐのは非現実的である．全ての可能性を網羅できるわけではなく，不測の事態が起こる可能性を排除できないからである．性能異常を未然に防ぐ手法を用いて性能異常の発生を完璧に防ぐには，システム内で性能異常を引き起こす可能性がある全ての要因を事前に洗い出しておく必要がある．しかし，複雑な実システムでこれを行うのは困難だといえる．

システムに起きる変化の種類に応じて対策法も大きく異なるため，一つの手法がある特定の要因にしか対応できない．よって，全ての可能性を排除するためには事前に全ての要因を洗い出しておき，複数の手法を個々に適用する必要がある．

性能異常を防ぐためには，システムの変化に対応しながらサービスを提供していく必要がある．ウェブアプリケーションを提供するシステムはサービスを提供し続ける内に，さまざまな要因が変化する．クライアント数や人気のあるコンテンツなどワークロードが変化したり，マシンの追加や削除などにより計算機資源が変化したり，アプリケーションの変更などプログラムが変化したりする．

既存手法は，大きく三つに分けることができる．一つ目はシミュレーションを行うことで，さまざまに変化する状況下においてシステムに性能異常が発生するかどうかを確認する手法である．二つ目はシステムの変化を監視し，その変化へ適切に対応していくことで性能異常の発生を防ぐ手法である．三つ目は，現状のシステム内部を詳細に把握することで，性能異常を引き起こす要因の混入を防ぐ手法である．

(1) シミュレーションによる性能異常の確認

シミュレーションを行い，さまざまに変化する状況下において性能異常が発生するかどうか確認する手法が提案されている．性能異常が発生しないことがわかれば，安心してサービスの運用を開始できる．また，ある変化により性能異常が発生することが分かった場合は，対策をサービス運用前に行うことができる．

しかし，実際にサービスを提供しているシステムは巨大かつ複雑であるため，全ての可能性を網羅してシミュレーションを行うことは非現実的である．このため，性能異常発生の可能性を事前に全て排除することができるとはいえない．既存手法では，シミュレーションを行うためにワークロードやシステムをモデル化し，簡略化する．

また，シミュレーションにより性能異常が発生する可能性を検出した場合は，本研究のように原因究明支援が必要になる．サービスの運用開始前であっても，異常の原因究明を支援することは重要である．原因究明が遅れれば遅れるほど，サービスの提供開始が遅れ，ユーザ獲得の機会を失うことになる．

(2) サーバの性能パラメータを自動設定する手法

二つ目の手法は，リアルタイムで性能を監視しながらサーバの性能パラメータを自動で変更していく手法である．サーバには，最大同時接続数や各処理に割り当てるメモリのサイズなど，性能に関わるパラメータが多数存在する．本来は，このパラメータを管理者が，サービスを提供するシステムの特性に合わせて適切に設定する必要がある．しかし，これを人手で行うのが困難なため，自動化する研究が盛んに行われている．

この手法はサーバの性能パラメータに特化した手法であり，ソフトウェアのバグやハードウェア故障による性能異常を防ぐことは行わない．また，全てのパラメータを自動で適切に設定するのは現実的ではない．存在する全てのパラメータを適切に設定するには膨大なパラメータ空間を探索する必要があるため，各手法では一部のパラメータのみを設定するのが通常である．もし仮に全てのパラメータ設定を自動で行おうとすれば，適切な設定を発見するまでに膨大な時間を要したり，適切な設定を発見する過程で性能異常を引き起こすような設定をしてしまう危険がある．

(3) マシン間の依存関係を解決する手法

三つ目の手法は，マシン間の依存関係を解決する手法である．管理者がシステムを詳細に把握することで，性能異常を引き起こす要因が混入するのを防ぐのが目的である．この手法は，サーバ間の依存関係が要因で発生する性能異常の回避が目的である．各サーバ内部で発生するような性能異常については対象外である．

2.1.3 動作異常やシステム停止への対策

さらに，動作異常やシステム停止など，性能異常とは異なる障害への対策手法についてまとめる．そうすることで，性能異常が他の障害に比べて異なる性質を持つことを示し，性能異常への対処手法が別途必要であることを示す．図 2.1 における青い領域に当たる．

以下，既存手法を大きく二つに分けて説明する．一つ目は，動作異常やシステム停止が発生した際に早期検出と原因究明支援を行う手法である．二つ目が，動作異常やシステム停止の発生を未然に防ぐための手法である．

(1) 早期検出と原因究明支援を行う手法

始めに，動作異常やシステム停止の早期検出と原因究明支援を行う手法を説明する．ここで挙げる手法をそのまま利用して，本研究の目的である性能異常の早期検出と原因究明支援を達成することはできない．動作異常やシステム停止は性能異常に比べて，検出と原因究明が比較的容易な傾向がある．なぜなら，普段と異なる処理を行うため，異常が表面化しやすい傾向があるからである．第 2.3 節では，各手法を説明し，性能異常の検出と原因究明にはそのまま利用できないことを示す．

本節では，一例のみを挙げる．例えば，アサーションや例外を利用して障害の

検出と原因究明を行う手法が提案されている [26]．しかし，性能異常の場合，処理自体は正常に行われるため，アサーションや例外を引き起こすことなく発生する．そのため，処理時間の監視など他の手段を考案する必要がある．

(2) 発生を未然に防ぐための手法

次に，動作異常とシステム停止の発生を未然に防ぐ手法を説明する．そうすることで，性能異常とは異なる性質を持つことを示し，性能異常への対処が別途求められることを示す．

この分類に含まれる手法の例としては，他サイトと情報を共有することでシステムの設定を自動化する手法が挙げられる．サーバへのモジュール導入など，動作異常を引き起こすような設定を行う場合はサイト間で似たような設定をすることが多い．そのため，他サイトとの情報共有が可能になる．しかし，性能異常の場合，マシンスペックやアプリケーションの性質などサイトごとに適切な設定が異なることが多く，情報共有を容易に行うことができない．

2.1.4 ウェブアプリケーション以外のシステムでの障害対策

最後に，オペレーティングシステムにおける障害対策などウェブアプリケーション以外を対象にした研究についてまとめる．対象の違いにより本研究とは異なる点，また対象が異なっても本研究に応用できる点両方に注目しながらまとめる．図 2.1 における白い領域に当たる．

本節では既存手法を二つに分ける．まず始めに，異常の原因究明を行う手法についてまとめ，次に，それ以外の手法についてまとめる．

(1) 異常の原因究明を行う手法

例として，オペレーティングシステム内で発生した性能異常の原因究明を行うために，処理時間の分布に着目する手法が挙げられる [27,28]．ウェブアプリケーションとは異なる対象であるものの，処理時間の分布が性能異常を特徴づけるために有用な情報であることがわかる．

(2) その他の手法

例として，アプリケーションの設定を他のユーザと共有する手法が挙げられる [29]．オペレーティングシステムのように，多数のユーザがその上で同じアプリケーションを動作させている場合には有用な手法である．ウェブアプリケーショ

ンはサイトごとに異なるため，アプリケーションに関する情報も同時に共有するなどの工夫が必要だといえる．

2.2 性能異常への対策

本節ではウェブアプリケーションにおける性能異常への対策手法についてまとめる．まず，早期検出と原因究明支援を行う手法についてまとめ，次に発生を未然に防ぐための手法についてまとめる．

2.2.1 検出，原因究明支援手法

(1) コンポーネント単位での監視を行う手法

Chen ら [17, 30] は，各リクエストの処理を行う際に利用したコンポーネントと各コンポーネントでの滞在時間を記録するフレームワークを提案している．それだけでなく，収集した情報に統計処理を施すことで異常なパスを通過したリクエストを発見したり，処理時間の分布の変化を見つけることも行う．この研究はあくまでもフレームワークの提案であり，具体的にどのように情報を収集するのか，またどのような統計処理を施すべきかについては手法を確立しているわけではない．その代わりに，提案フレームワークを三つのシステムに実装した結果をケーススタディとして報告している．ケーススタディで報告されているように，システムを熟知したユーザが情報収集システムを適切に実装し，さらに収集した情報に適切な統計処理を施すことができれば，異常の検出と原因究明を行う際の強力な支援となる．

しかし，システムに大きな変更を加える必要があるため，導入が容易であるとはいえない．本研究のケーススタディでも用いたような 3 層構造のウェブアプリケーション提供基盤に適用した例では，サーバ内のさまざまな場所でプログラムを改変している．具体的には，ウェブサーバでリクエストごとに ID を付加する処理を実装し，アプリケーションサーバで先の ID を引き継ぐために JSP（Java Server Pages），Servlet，EJB（Enterprise Java Beans）それぞれのコンテナと JDBC ドライバに実装を加えている．追加したコードの行数は 1,000 行近くに及ぶと報告されている．

Aguilera ら [18] は各コンポーネントでの滞在時間とコンポーネント間の遷移に要した時間の平均値を管理者に提示する手法を提案している．コンポーネントご

とに細粒度で監視を行えるため，検出と原因究明の能力は高いといえる．

しかし，システムに大きな変更を要するため，導入が容易であるとはいえない．各コンポーネントでリクエストの受信，送信時刻を記録する必要がある．Chen ら [17,30] の手法同様，各コンポーネントのコンテナや JDBC ドライバへの実装を避けることはできない．ただ，Aguilera らの手法は Chen らの手法に比べれば，システムへの実装量が削減できる．なぜならば，リクエストごとに ID を引き継ぐ処理の実装が不要だからである．代わりに，取得した情報を統計的に処理することで，リクエストが遷移したパスと遷移に要した時間を推測する．具体的には，考えられる遷移の組合わせの中から多数派を正しいとする．

Chanda ら [15] は各コンポーネントでの CPU 使用率とロック取得に要した時間の平均値を管理者に提示する手法を提案している．計算資源を大量に消費したり資源への競合を引き起こしているコンポーネントを見つけることができるため，検出と原因究明の能力は高いといえる．この手法ではそれだけでなく，リクエストが通過したパスに関する情報も合わせて提示する．そうすることで，性能異常を引き起こしているコンポーネントを利用するリクエストを特定することができ，さらに原因究明の支援となる．

しかし，システムに大きな変更を要するため，導入が容易であるとはいえない．Chanda らの手法は，さまざまな方法で行われるリクエスト処理の引継ぎを逃さずに追跡することを目的としている．共有メモリやイベント，SEDA [31] サーバアーキテクチャにおけるステージを介して行われる処理の引継ぎを追跡する．そのために，オペレーティングシステムやサーバによって提供されているライブラリを適切に変更する必要がある．

Barham ら [16,32] は各リクエストの資源使用率と発行したイベントを取得する手法を提案している．さらに，得た情報をクラスタリングし，異常な処理を行ったリクエストを発見する．例えば，似たような処理を行うリクエストにも関わらず，少数のリクエストのみ異なるイベントを発行している場合，それらに性能異常が発生している可能性が高い．ほとんどのリクエストはデータをキャッシュから取得しているにも関わらず，少数のリクエストのみディスクからデータを取得しているような場合が当てはまる．少数のリクエストのみに発生している性能異常を検出，原因究明できることから，能力は高いといえる．

しかし，システムに大きな変更を要するため，導入が容易であるとはいえない．まず，オペレーティングシステムやサーバ，アプリケーションがイベントを適切に発行するようにプログラムを修正する必要がある．さらに，管理者がそのイベ

ント間の依存関係を詳細に記述する必要がある．Barham らの手法では，リクエスト ID を伝播させる実装を回避するために，取得したイベント情報からリクエストの実行パスを組み立てる．そのためにはイベント間を結びつけるために利用すべき属性を管理者が記述しておく必要がある．例えば，同一リクエストを処理するために発行された `W3Server/Start` と `W3Server/End` という二つのイベントでは `W3Id` という属性の値が等しくなるということを記述しておく必要がある．

(2) 資源使用率の監視を行う手法

Cohen ら [19, 33] は性能異常を引き起こすような資源使用率のパターンを特定することで，性能異常を予測する手法を提案している．資源使用率を統計的に処理し，過去に性能異常を引き起こした資源使用率のパターンを記憶しておく．そして，資源使用率が，異常を引き起こす可能性の高いパターンに近づいた場合，近いうちに性能異常が発生する可能性が高いと予測することができる．一般的なオペレーティングシステムではマシン単位の資源使用率を取得する機構が提供されていることがほとんどであるため，Cohen らの手法は導入が容易だといえる．

しかし，原因究明の能力は高くない．Cohen らの手法では多数存在する資源使用率の内，性能異常に関連する指標を自動的にいくつか選択する．よって，性能異常発生時にはその指標に注目することで原因究明を行うことができる．しかし，資源使用率と実際の原因究明の隔たりは大きく，システムを熟知した管理者でないと原因の見当をつけることすら難しい．例えば，CPU 使用率とディスク I/O 発行量が異常な値を取っていることが分かったとしても，システム内のどのコンポーネントがその変化に寄与しているかを見つけることは難しい．

Cohen ら [34] と Bodík ら [35] は資源使用率を性能異常のシグネチャとして利用する手法を提案している．発生している性能異常が過去に発生した性能異常と同じものであることがわかれば，過去に適用した修復方法を再度システムに施すことで，迅速に性能異常から回復することができる．例えば，クライアント数の増加によりコネクション不足が発生した場合，最大接続数を増加させることで性能異常から回復することができる．しかし，最大接続数を大きく設定しすぎると，過負荷によりシステムが停止してしまう可能性があるので，最大接続数はあまり大きな値に設定することができない．よって，さらにクライアント数が増加した場合はもう一度コネクション不足が発生する可能性がある．マシンごとの資源使用率を取得する機構はすでに用意されている場合がほとんどであり，導入は容易である．

Cohen らの手法と Bodík らの手法は再発する性能異常の原因究明が目的のため，本研究のような原因箇所を特定することは行わない．また，もし仮に得られる情報を基に原因究明を行おうとしても，異常な値を示している資源使用率が分かるだけで，詳細な原因究明を行うことは難しい．

Bhatia ら [36] は資源使用率に加えてオペレーティングシステムによって提供される，より詳細なシステム情報から性能異常の原因究明を行う手法を提案している．オペレーティングシステムによって提供される情報とは，L2 キャッシュミスやページの割当てに関する情報である．そして，異常を検出した際には収集しておいた情報をユーザへ提示する．詳細な情報を取得しているため，原因究明の能力は高いといえる．ある程度原因の絞込みを行えていて，その後の詳細な原因究明を行う際に有用な情報になる可能性が高い．

しかし，導入は容易ではない．L2 キャッシュミスやページの割当てに関する情報など低レイヤの情報を取得するために，カーネルへの実装を要する．

(3) ログの監視を行う手法

Xu ら [20] は サーバが出力するログを統計解析し，異常な出現パターンを検出する手法を提案している．一部のサーバでは各コンポーネントが処理ステータスをログに出力するよう実装されていることがある．この情報を利用して，一定時間ごとに各ログメッセージの出現割合を計算し，異常な出現パターンを検出する．この手法は検出と原因究明の能力は高い．なぜなら，ログは各コンポーネントの内部状態を表す有力な情報であり，処理が成功したか失敗したか，現在リトライしているかなどといったことが分かる．

しかし，導入の容易さは低くなっている．なぜならば，詳細なログを出力する機構を備えているサーバにしか適用できないからである．実際，ログに着目した既存研究 [20, 21] でもその対象は Darkstar オンラインゲームサーバ [22] や分散処理フレームワークの Hadoop [23] に限られている．また，詳細なログを出力するように既存のサーバを修正することは難しい．ログはプログラムの作成者が，サーバの動作を熟知し，有用だと感じた変数を出力する必要がある．

(4) アクセス数の監視を行う手法

Bodík ら [24] はリクエストの種類ごとにアクセス数を監視し，その変化を検出する手法を提案している．性能異常の影響はアクセス数に現れることがある．例えば，性能異常によりページが表示されづらい場合，ユーザが更新を繰り返しアクセス数が増加することがある．この手法は導入が容易だということができる．な

ぜなら，通常サーバではリクエストの URL をログに保存するためである．実際に Apache HTTP サーバ [12] ではそのような機構を備えている．このログからリクエストの種類ごとのアクセス数を得ることができる．

一方，この手法の検出，原因究明能力は低い．なぜなら，異常の早期検出を行えていないからである．アクセス数が変化しているということは，すでにユーザに異常が感知されているということである．

2.2.2 発生を未然に防ぐ手法

(1) シミュレーションによる性能異常の確認

Stewart と Shen [37] は各コンポーネントの処理に要する資源量やコンポーネント間の通信にかかる資源量などを計測し，モデルを作成する手法を提案している．作成したモデルを利用し，リクエスト数に応じて性能がどのように変化するかを計算する．

Stewart と Shen の手法では，コンポーネント内に存在するバグなどの不確定要素は対象外としている．モデル化の際に注目しているのはコンポーネント間の通信にかかるオーバーヘッドである．そのため，システムが正常に動作している時には高い精度を得ることができるものの，バグによる異常な動作はシミュレーションできない．

Zheng ら [38] は対象のデータセンタを縮小化した環境を利用して，実データセンタの性能を予測する手法を提案している．実データセンタ内の隔離された場所に，縮小化した環境を構築し，現実的なワークロードを与える．仮想化技術を利用したコンソリデーション環境での，資源割り当て量の変化がシステムの性能にもたらす影響をシミュレーションする．

Zheng らの手法では，未知のワークロードに対してシステムが正常に動作するかは確認しない．各仮想マシンに対して必要な計算機資源量を見積もることを目的としているため，実データセンタで取得したワークロードのみを利用してシミュレーションを行う．また，Zheng らの手法ではデータセンタを縮小化した環境で実験を行うため，実環境とは異なる振舞いを見せる可能性がある．例えば，三層構造で構成されているウェブアプリケーション提供基盤の場合，Zheng らの手法では各層のマシンを 1 台ずつに制限する．実データセンタ内に仮想マシンを配置するため，実験にかかる資源を少なくとどめる必要があるからである．

Li ら [39] はクライアント側で観測される応答時間を予測するための手法を提案している．これまで，性能の指標として予測できるのはサーバの処理にかかる時間が一般的であった．なぜなら，クライアントが観測するリクエストの応答時間はサーバでのリクエスト処理時間以外にもさまざまな処理時間を含んでいるためである．サーバへのリクエスト送信，サーバからのレスポンス取得，外部サービスとのやり取りにかかる時間がこれに当たる．Li らの手法はこれらの処理時間を含めモデルを作成する．そして，ある処理を高速化できた場合に，クライアントが観測する応答時間がどの程度高速になるかを予測する．たとえある処理にかかる時間を 100 ミリ秒高速化したとしても，リクエスト全体の応答時間が 100 ミリ秒高速化するとは限らない．高速化できた処理と並列に行われている処理がリクエスト全体の処理のボトルネックになっている可能性があるからである．

Li らの手法では個々の処理自体に発生する性能異常は対象外である．処理同士の依存関係の変化を対象にしてモデルを作成しており，処理自体にかかる時間は正しく与えられるものとしている．

Stewart ら [40] は，対象のウェブアプリケーションをスペックの異なるマシン上で動作させたときの性能を予測する手法を提案している．マシンのスペックと性能の間には定式化可能な関係が存在することがあり，それを利用して性能を予測する．例えば，キャッシュサイズとキャッシュミス数には定式化可能な関係があることを三つのウェブアプリケーションで実験を行い示している．

しかし，Stewart らの手法はマシンスペックとリクエストの傾向をモデル化の対象にしており，他の要因によって発生する性能異常を予測することは行わない．例えば，ソフトウェアにバグがあり，クライアント数の増加によって性能異常が発生するような場合には対応しない．

Stewart ら [41] は，リクエストの量だけでなく種類ごとの分布にも着目して，ワークロードをモデル化する手法を提案している．ウェブアプリケーションはリクエストの種類に応じてさまざまな処理を行う．当然，行う処理に応じて必要な資源の量や種類は異なってくる．それにもかかわらず，Stewart らの手法以前は各リクエストの種類が発行される割合の変動は無視していた．割合は一定として，リクエスト量だけに注目してワークロードのモデル化を行っていた．

Stewart らの手法はワークロードのモデル化を行う手法であり，その他の変化により発生する性能異常は対象としていない．例えば，マシンの追加や削除が行われたり，アプリケーションを変更した場合の性能への影響は予測しない．

Stewart ら [42] は，管理者による設定誤りを試運用時に発見する手法を提案して

いる．ウェブアプリケーションを運用する際には，ソフトウェアコンポーネントをどのマシンに配置するか，ソフトウェアキャッシュの一貫性をどのように維持するかなど，さまざまな設定を管理者が適切に行う必要がある．Stewart らの手法は理想的な性能を発揮する前提で作成したモデルと実際に得た性能を比較することで，設定誤りを発見する．

Stewart らの手法は管理者による設定誤りを対象にした手法である．ソフトウェアの動作はモデル化を行わないため，ソフトウェアバグを発見することはできない．ハードウェア故障についても同様である．

Nagaraja ら [43] は，管理者による設定誤りがシステムへ混入するのを防ぐために，管理者に試験環境を提供する手法を提案している．試験環境には，以下の二つが求められ，これらを満たす手法となっている．まず，試験環境は実環境となるべく近い環境であることが求められる．また，試験環境から実環境へ移行する際に設定誤りが混入するのを防ぐ必要がある．

Nagaraja らの手法は管理者による設定誤りを対象としており，性能異常を引き起こす他の要因については対象外である．ソフトウェアバグやハードウェア故障についてはシミュレーションを行わない．

(2) サーバの性能パラメータを自動設定する手法

Zhang ら [44] は JBoss [13] の `MaxPoolSize` の自動設定を行う手法を提案している．`MaxPoolSize` は JBoss がデータベースへアクセスするためのコネクションのプールサイズを設定するパラメータであり，サーバの性能に大きく影響することが知られている．Zhang らの手法はファジィ制御を利用することで，クライアント数に合わせて適切に `MaxPoolSize` を変更する．

Sugiki ら [45] は Apache HTTP サーバ [12] の性能パラメータである `KeepAliveTimeout` と `MaxClients` の自動設定を行う手法を提案している．`KeepAliveTimeout` と `MaxClients` はともにサーバの性能に大きく影響を与えることが知られている．`KeepAliveTimeout` はレスポンス送信後，コネクションを維持する時間，`MaxClients` はサーバの最大同時接続数である．リクエストの到着間隔などをオンラインで分析し，ワークロードの変化に自動で追従する．

Zheng ら [46] はマシン追加時の設定を自動化し，さらに性能パラメータ間の依存関係を解決する手法を提案している．システムへマシンを追加する際には負荷分散の設定などを適切に変更することが求められる．Zheng らの手法は事前に設定ファイルのテンプレートを用意しておき，マシン追加時にはテンプレートから

適切な設定ファイルを自動で作成する．また，キャッシュサイズやプールサイズなど性能パラメータの間には依存関係が存在することがある．Zheng らの手法はその依存関係を自動で解決することで，管理者がどのパラメータを変更する必要があるのかを自動で提示する．

Xi ら [47] はパラメータ空間を探索し，ワークロードに適した設定を発見する手法を提案している．早急に適切な設定を発見するために，空間探索アルゴリズムを考案している．Xi らの手法は理論上，全てのパラメータを対象に探索することが可能である．しかし，パラメータ数が増加すると探索時間も増大するため，数個のパラメータを対象にするのが現実的だといえる．実験では WebSphere アプリケーションサーバ [48] の四つのパラメータを設定するために 15 分を要したと報告している．

Osogami と Kato [49] もパラメータ空間を探索し，ワークロードに適した設定を発見する手法を提案している．Osogami と Kato の手法は各パラメータが性能へ及ぼす影響を見極め，影響が少ないと判断した場合はそのパラメータの探索を即座にやめることで，探索にかかる時間を短縮している．Osogami と Kato の手法でも，実験では四つのパラメータを設定するケーススタディを報告している．

(3) マシン間の依存関係を解決する手法

Tak ら [50] は 3 層構造の各層間で行われる通信回数を把握する手法を提案している．例えば，あるリクエストを処理する際にアプリケーションサーバとデータベースサーバの間で発生する通信回数が極端に多ければ，本来はアプリケーションを修正して，無駄な通信を削減する必要がある．しかし，近年，ウェブアプリケーションを提供する環境は非常に複雑であり，システム内部全てを管理者が正確に把握するのは容易ではない．

Chen ら [51] は各サーバ間の依存関係を解決する手法を提案している．現在，データセンタなどでは複数のサービスを同時に提供していることがある．このような環境では DNS サーバやアプリケーションサーバ，データベースサーバなどさまざまなサーバが複雑に互いを利用しながらリクエストを処理している．システムのメンテナンスを行う際などには，サーバ間の依存関係を把握して行わないと，サービスを適切に運用できなくなる可能性がある．人手により依存関係を解決するには多大な時間と労力を要するため，Chen らの手法はこれを自動化している．

2.3 動作異常やシステム停止への対策

(1) 早期検出と原因究明支援を行う手法

Yamanishi と Maruyama [52] は，ログを監視してネットワーク障害を検出する手法を提案している．ログを確率モデルで表し，異常な出現パターンを発見する．Yamanishi と Maruyama の手法で監視するログは警告やエラーなど，Xu ら [20] の手法で用いているものに比べて重要度の高い情報のみを出力するものである．システムの内部状態を細かく監視することができないため，性能異常の検出への適用は難しいと考えられる．

Lou ら [21] は，ログを監視してアプリケーションの動作異常を検出する手法を提案している．Lou らの手法では，ログメッセージ同士の依存関係を自動で抽出し，その依存関係の違反を検出する．例えば，ファイルを開いたことを示すメッセージ数とファイルを閉じたことを示すメッセージ数は一定時間内で同じであるという依存関係を自動抽出し，違反があった場合に警告を発する．手法の性質上，性能異常の検出への適用も不可能ではないと考えられるが，実験ではタスク実行の失敗など，動作異常の検出を対象にしている．また，仮に性能異常検出に適用可能だとしても，詳細なログを要するため，導入が容易だとはいえない．

Jiang ら [53] は，システムの内部状態の変動を監視して動作異常の検出と原因究明を行う手法を提案している．Jiang らの手法で利用する情報はスレッドプール内のアイドルスレッドの数やウェブサーバで保持しているセッション数，SQL が取得した行の数などである．Jiang らの手法の性質上，性能異常への適用を行える可能性もあると考えられる．しかし，実験で対象としている障害はコンポーネント内で発生した例外やコンポーネントの削除であり，動作異常を引き起こすものである．また，仮に性能異常検出に適用可能だとしても，詳細な情報を必要とするため，導入が容易だとはいえない．

Attariyan と Flinn [54] はサーバの動作異常を引き起こしている設定を発見する手法を提案している．プログラム内でのデータ受渡しを監視し，発生したエラーに関連の高い設定を発見する．Attariyan と Flinn の手法は明示的にエラーが発生する障害にのみ対応可能なため，性能異常への適用は難しいといえる．

Chen ら [26] は障害を発生させたソフトウェアコンポーネントを発見する手法を提案している．Chen らの手法はまず，システムに変更を施し，各リクエストが利用したソフトウェアコンポーネントを記録する．そして，各リクエストごとにアサーションや例外，ページが表示されないといった動作異常を検出する．そして

両者の情報を利用し，障害を発生させた可能性の高いコンポーネントを見つけ出す．アサーションや例外などを監視するだけでは，性能異常を発見できない可能性が高い．

So と Sirer [55] はサーバの停止を早期検出する手法を提案している．サーバ停止の検出にはハートビートパケットを送信する方法が広く知られている．一定時間ごとに監視ノードから各ノードへハートビートパケットを送信し，一定時間内に返答がない場合は対象ノードが停止していると判断する．So と Sirer の手法は，これまで静的に定められていたハートビートパケットの送信間隔を，ノードに合わせて動的に設定する．過去の各ノードの停止履歴を保持し，頻繁に停止するノードへは頻繁にハートビートパケットを送信し，停止が少ないノードへはあまりハートビートパケットの送信を行わない．ハートビートパケットはノードが停止しているかどうかを確認するための手段である．サーバの性能異常を検出することはできない．

(2) 発生を未然に防ぐための手法

Su ら [56, 57] はプログラムの設定を自動で行う手法を提案している．同じプログラムに対して他人が行った設定を共有し，自分の環境に適用することで設定を自動化する．ただし，Su らの手法は性能異常の回避への適用は困難である．Su らの手法で対象としているのは，異なる環境においても似たような動作で設定可能なものである．例えば，Apache HTTP サーバで PHP を動作可能にする設定などがこれに当たる．性能異常を引き起こす要因は，性能パラメータなど，環境によって適切な値が異なることが多々ある．

Oliveira ら [58] は管理者がシステムに対して誤った動作を行うのを防ぐ手法を提案している．まず，システムを変更する際に必要な動作を，システムを熟知した管理者が事前に記述しておく．そして，実際の作業を行う他の管理者が記述にない不要な動作を行おうとした際に自動で防ぐ．Oliveira らの手法で対象としているのは，あらかじめ決まりきった作業を行う際の管理者の動作である．例えば，マシンを追加する際に，どの設定ファイルを変更すべきなのか，またどのマシンを再起動すべきなのか，といった動作を記述する．性能異常は不測の事態に発生することがあり，正しい動作を事前に記述しておくことができない可能性が高い．

Marinescu ら [59] は効率よくフォールトインジェクションを行い，ソフトウェアバグの混入を防ぐための手法を提案している．Marinescu らの手法はプログラムの解析を行うことで，他の変数の状況を考慮し，フォールトを挿入する．これまでは，

ライブラリ呼び出し時に常にフォールトを挿入したり，ランダムに挿入するのが一般的であった．Marinescu らの手法では，例えば，対象スレッドがロックを保持している場合だけフォールトを挿入するような設定を行うことができる．Marinescu らの手法ではライブラリがエラーを返すような障害が対象である．一方，性能異常はエラーを返すような明示的な変化を生じずにシステムの挙動がおかしくなることも多い．

Zhong ら [60] はデータの複製数をデータコンテンツの人気に応じて定める手法を提案している．データを複製しておくことで，ハードディスク故障やマシン停止の場合でもサービスを継続することができる．ウェブアプリケーションではコンテンツの人気の分散が大きい場合が多いため，Zhong らは人気のある重要なコンテンツの複製は多く，人気がなくあまり重要ではないコンテンツの複製は少なくする手法を提案している．データ損失に特化した手法であり，性能異常を引き起こす他の要因は対象にしていない．

Sen ら [61] は複製データ間の整合性の確認を高速に行う手法を提案している．複製を作成しておくことで，一部のデータに不具合が生じた場合でも適切にサービスを提供することができる．これまでの手法では，毎回全ての複製を取得し，整合性を確認する必要があり，オーバーヘッドが大きかった．Sen らの手法では中間ノードにデータの履歴を配置することで，やや信頼性は劣るものの，処理を高速に行うことができる．データに不具合が生じた場合を対象にしており，性能異常を引き起こす他の要因は対象にしていない．

Candea ら [62] はソフトウェアをコンポーネントごとに再起動する手法を提案している．メモリリークなど，一部の異常はシステムを再起動し，初期状態に戻すことで復旧できる．しかし，再起動中はサービスを行えないため，Candea らの手法のように，再起動を高速化する手法が提案されている．再起動は全ての障害を復旧できるわけではなく，他の手法と併用する必要がある．

Perkins ら [63] はサーバが攻撃された際に自動で攻撃を無効化するパッチの生成を行う手法を提案している．Perkins らが対象としているのはバッファオーバーフローなどの攻撃によりサーバに発生する障害である．実際に，作成するパッチはコントロールフローを変更し，不正な処理を行わせないようにプログラムの挙動を変更するようなものである．処理が適切に行われている場合でも性能異常は発生するため，上述のパッチによる対策は不十分である．

2.4 ウェブアプリケーション以外のシステムでの障害対策

(1) 異常の原因究明を行う手法

Joukov ら [27] と Traeger ら [28] は関数の処理時間の分布に注目したオペレーティングシステムの性能プロファイリング手法を提案している．異なる関数にも関わらず，処理時間の分布が似ている場合，資源競合が発生している可能性があり，調査を行う必要がある．Joukov らと Traeger らの手法は処理時間の分布の類似度に着目している点で，本研究の性能異常分類手法と似ている．ウェブアプリケーションとオペレーティングシステムで対象が異なっても，処理時間の分布はシステムの内部状態を表す重要な情報だということがわかる．

Yuan ら [64] は発行したシステムコール列をシグネチャとして，アプリケーションの動作異常を分類する手法を提案している．システムコールの引数や返値に着目してシグネチャを作成する．異常を分類する手法は性能異常においても必要である．しかし，Yuan らの手法はそのままウェブアプリケーションの性能異常に適用できない可能性が高い．性能異常による影響がシステムコールの引数や返値に現れる可能性がそれほど高くないからである．Yuan らの手法で対象としているのは，ブラウザでウェブページが適切に表示されないといった異常や，共有フォルダが開けないといった異常である．

Lim ら [65] は VoIP（Voice over Internet Protocol）のような電話技術システムにおいて，ログを障害の原因究明に利用する手法を提案している．ウェブアプリケーションにおいてもログはシステムの内部状態を知るための重要な資源である．しかし，プログラムの開発者によって適切に実装されている必要があり，既存システムへの導入が容易に行えるとは限らない．

Wang ら [66] は Windows レジストリ [67] の設定誤りを発見する手法を提案している．Windows 上で動作しているアプリケーションが異常な動作をした場合，Windows レジストリが原因のことがあり，その場合に原因究明を行う．Wang らの手法では，異常な動作を示した場合，他のユーザとレジストリ情報を比較する．その結果，大多数のユーザと異なるレジストリが見つかった場合，そのレジストリが原因の可能性が高いと判断する．ウェブアプリケーションの性能異常に対しても，ユーザ間で情報を共有するシステムは有用だと考えられる．しかし，各環境で適切な設定は異なるため，単純には情報を共有できない．

Glerum ら [68] は Windows のバグに関する情報を適切に収集し，デバッグを円滑に行うための手法を提案している．Windows で障害が発生した場合，スタックに

関する情報などを適切に取得しバグレポートとして報告する．ウェブアプリケーションの性能異常に対しても，ユーザ間で情報を共有するシステムは有用だと考えられる．しかし，各環境で適切な設定は異なるため，単純には情報を共有できない．

Mirgorodskiy ら [69] はスーパーコンピューティング環境において，異常な振舞いをしているプロセスを発見する手法を提案している．Mirgorodskiy らの手法では各プロセスのプロパティを記録し，他とプロパティが大きく異なるプロセスを異常なプロセスとして提示する．Mirgorodskiy らの手法はスーパーコンピューティング環境のように，多数のプロセスが似通った処理を行う環境を対象としている．ウェブアプリケーションでは，処理によってプロセスの振舞いが大きく異なる．

(2) その他の手法

Kushman と Katabi [29] はソフトウェアの設定を自動化する手法を提案している．各ユーザが設定時に行った動作を収集，記録しておき，他のユーザはそのレポジトリからダウンロードすることで再利用する．サーバの性能パラメータは環境ごとに適切な値が異なるため，ウェブアプリケーションの性能異常への対策にはそのまま利用することはできない．サービスを提供する環境に関する情報も同時に収集するなど，さまざまな変更が必要だと考えられる．

Oliner と Stearley [70] はスーパーコンピュータのログを調査し，障害検出と原因究明に利用する上で注意すべきことを分析している．ログはフィルタリングなどの適切な処理を行って初めて利用可能であり，出力されたまま利用するのは現実的ではないことなどが述べられている．ウェブアプリケーションでも同様のことがいえる可能性が高い．

Brown と Patterson [71] はメールサーバの管理者がミスを犯した際に，システムをロールバックする手法を提案している．通常，システムをロールバックするためには，スナップショットを取得しておき，問題が発生した際にそのスナップショットを利用する．しかし，対象がメールサーバだった場合，既読のメールはロールバックして削除してはならないなどの制約が伴う．Brown らの手法はそのような問題に対処している．ロールバック手法はウェブアプリケーションにおいても障害からの復旧に有用である．しかし，性能異常はワークロードやデータベース内のデータ量など，ロールバック不可能な要因によっても発生する．そのため，単純にロールバックを行うだけでは全ての性能異常からは回復できない．

2.5　まとめ

本章では，本研究と関連のある既存研究についてまとめた．まず，ウェブアプリケーションにおける性能異常の検出と原因究明を行う手法についてまとめた．そうすることで，既存手法が検出，原因究明の能力か導入の容易さどちらか一方を重視したものであることを確認した．次に，性能異常を未然に防ぐ手法についてまとめた．重要性について確認しながらも，必ずしも全ての性能異常を完璧に回避できないことも確認し，検出と原因究明を行う手法の必要性について述べた．

動作異常やシステム停止など，性能異常以外の障害への対策についてもまとめた．そうすることで，性能異常がその他の異常と異なることを確認し，別途対策が必要であることを述べた．最後に，ウェブアプリケーション以外のシステムでの障害対策についてもまとめた．対象が異なっても応用可能な点，対象が異なることで対策法が異なる点について述べた．

第3章　性能異常検出手法

本章では提案手法の要素技術の一つである性能異常検出手法を詳細に説明する．第 1.3.1 節で述べたように，提案手法ではリクエストの種類ごとに処理時間を計測する．そして，管理図を用いて，計測した処理時間を統計的に監視していく．管理図は正常時と運用時それぞれで計測した処理時間の分布を比較する．そして，分布に違いが現れた場合，性能異常として警告を発する．

本章ではまず，リクエストを種類ごとに分類する方法を述べる．そうすることで，提案手法を用いた処理時間の監視が実現可能であることを示す．次に，処理時間の監視に管理図を用いる目的を述べ，さらに管理図の説明を行う．その後で，提案手法における管理図の適用方法を説明する．

3.1　リクエストの種類ごとの分類

リクエストの種類の分類は管理者が人手で行うことを想定している．クライアントは目的に応じて，ウェブアプリケーションにさまざまなリクエストを送信する．オークションサイト RUBiS [72] の例では，トップページを取得する `Home` やユーザの登録情報を確認する `AboutMe`，出品中の商品に対して入札を行う `StoreBid` などがある．リクエストを種類ごとに分類するためには，URL を通常は用いる．リクエストは種類ごとに，個別の HTML ファイルや Servlet を利用することが多いので，通常はこれを判別すれば分類することが可能である．上記の三つのリクエストの種類ではそれぞれ `index.html`，`AboutMe`，`StoreBid` を URL で指定する．提案手法では，管理者は最低限，どのような HTML ファイルや Servlet があるかを知ることができると考えている．

ただし，同じ Servlet にまずアクセスする場合でも，受け渡すパラメータによってその後サーバ側で行う処理が異なる場合がある．RUBiS においては `BrowseCategories` と `BrowseCategoriesInRegion` というリクエストの種類がこれに当たる．これら二つのリクエストの種類は同じ Servlet の `BrowseCategories` を URL で指定してアクセスする．しかし，`region` と

いうパラメータを URL で指定するかどうかでその後の処理は異なってくる．`region` パラメータを指定すれば `BrowseCategoriesInRegion`，指定しなければ `BrowseCategories` となる．この場合も，URL を用いてリクエストの種類の分類を行うことができる．

さらに複雑なウェブアプリケーションでは，URL だけではリクエストを種類ごとに分類することが困難な場合があるかもしれない．しかし，この場合でも，提案手法が全く動作しないわけではない．分類が粗くなってしまった箇所では，性能異常の早期検出と原因究明支援の能力は落ちるが，それ以外の箇所では通常通り異常の検出と原因究明支援を行うことができる．検出，原因究明の能力と導入の容易さにはやはりトレードオフがあり，本研究では導入の容易さを保つために，ある程度の能力の低下は仕方ないと考えている．

3.2 管理図を用いる目的

3.2.1 性能異常検出の難しさ

ウェブアプリケーションのリクエスト処理時間はスケジューリングや資源競合などの不確定要素により正常時でも揺らいでいる．図 3.1 がそれを表している．図 3.1 はオークションサイト RUBiS のトップページを取得するためのリクエストの種類 `Home` においてリクエスト処理時間を計測し，その平均値を打点したグラフである．計測の際，第 5 章で報告するケーススタディと同じ実験環境を用いた．平均値の計算は 10 分ごとに行った．処理時間の平均値が 30 ミリ秒から 120 ミリ秒と大きく揺らいでいることがわかる．このため，性能異常の兆候を早期にかつ正確に検出するには，異常の兆候と正常時の揺らぎを見分ける必要がある．特に提案手法では監視粒度がリクエストの種類ごとと粗いため，その見極めが難しくなる．

しかし現在，性能異常兆候の検出は人手を介したアドホックな手法が主流になっている．例えば，リクエスト処理時間の平均値が事前に定めた閾値を超えたことをもって性能異常兆候と検出したり，直近の何分かの平均値が増加傾向にあることをもって性能異常兆候と検出したりといったようなことを行う．このような手法では閾値を何秒にするのか，また増加傾向が何分継続したことをもって性能異常兆候だと検出するかを適切に定めるのが難しく，経験則に頼っているのが実状である．

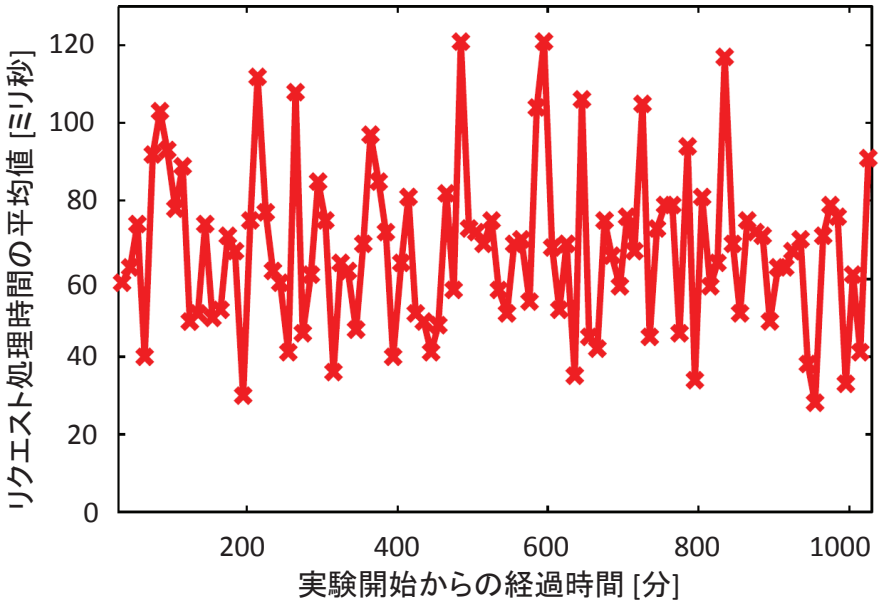


図 3.1: 正常時におけるウェブアプリケーションのリクエスト処理時間の平均値

3.2.2 管理図の利用

そこで，本論文では管理図 [25] という統計的手法を利用して，ウェブアプリケーションの性能異常の兆候検出を行う．管理図は元々，製品製造工程や経営工程を管理するためのものであり，工程が統計的に正常か異常かを判定するものである [73]. 過去の管理対象の値（例：ネジの長さ，企業利益，コレステロール値）の分布と現在の管理対象の値の分布を統計的に比較し，二つの分布が同じか異なるかを判定する．過去のデータを用いて基準線を計算しておき，現在の管理対象の値と基準線の位置関係とを比較することで，管理対象の値に統計的な変化が見られるかどうかを判断する．そして，管理図が管理対象の値に統計的に変化が生じたと判断し警告を発した場合は，工程に変化が生じた（例：工具の摩耗，取引先の減少，食生活の変化）はずであると考え原因究明を始める必要がある．

管理図は過去の品質と現在の品質のずれを統計的に判定するものであるため，ウェブアプリケーションの性能異常兆候検出への適用が期待できる．管理図を用いてウェブアプリケーションの過去の処理時間と現在の処理時間を比較することで，処理時間に現れる性能異常の兆候を経験則に頼らずシステマティックに検出できる．本論文でリクエスト処理時間とはサーバがリクエストを受信してから処理結果を送信するまでの時間を指し，人間のユーザとインタラクティブに処理を行

うウェブアプリケーションでは，このリクエスト処理時間が重要な性能指標となる．本論文でも性能指標としてリクエスト処理時間を用いる．

管理図を用いてウェブアプリケーションを監視する既存研究は Ye ら [74] による研究がある．この研究の目的はウェブアプリケーションへの侵入を検知することである．そのために，一定時間ごとに発行されたイベント数を管理図を用いて監視する．侵入が行われた時には，正常時に比べてイベントの発行数が変化するという前提に基づいている．実験では DoS（Denial of Service）攻撃の検知を行っている．

3.2.3　管理図の利点

管理図は次に挙げるような利点があり，ウェブアプリケーションの性能異常の兆候を検出するのに適している．管理図は人間が決定しなければならないパラメータが少なく，計算量も少ないため導入が容易である．精度に大きく影響を及ぼすパラメータを人間が決定する必要がある場合，利用者がその手法の統計的意味を正確に理解する必要がある．そのため，導入が困難になる．また，計算量が多い手法は適用範囲が限られてしまう．例えば実運用時に利用できなかったり，実運用時に利用するためには専用マシンを追加する必要があったりする．管理図の作成に求められる計算は平均値や範囲（最大値と最小値の差）の計算だけであり計算量が少ない．

さらに管理図は，管理対象の値に特別な制約がないため，本論文で述べる適用法以外にも，ウェブアプリケーションの特性や管理者の要求に応じてさまざまな適用が可能だと考えられる．管理図にはさまざまな種類があり，管理対象の値に応じて適切な管理図を選ぶことができる．

さらに，管理図により得られる結果は管理者にとって理解しやすい形式である．統計手法によっては，異常である確率が何パーセントといった出力を行うものもあり，最終的な判定を行うには新たに閾値を定める必要がある．管理図の出力は正常であるか異常であるかの 2 通りである．

3.3　管理図

本節では管理図の説明を行う．

3.3.1 概要

管理図は元々，製品製造工程や経営工程を管理するためのものであり，工程が統計的に正常か異常かを判定するものである．管理図は過去の品質と現在の品質のずれを統計的に比較し，現在の工程が正常か異常かの判定を行う．

管理図では管理対象の値を**特性値**と呼び，その特性値を監視する．そして，その特性値が統計的に異常な値となった場合に管理図は警告を発する．工場でのネジ製造工程を管理する場合を例にとって説明する．このネジ製造工程では特性値をネジの長さと定め管理する．平常時においては，ネジの長さの誤差はある一定の値に留まり，管理図は警告を発しない．逆に，もし管理図が警告を発した場合には，ネジ製造工程に何かしらの変化が生じたと考えることができ，工具や材料の検査など原因究明を始める必要がある．このように，管理図を用いることで，不良品になってしまうほどネジの長さの誤差が大きくなってしまう前に異常を検出することができる．

管理図には多くの種類があり，特性値の性質と活用目的により適切なものを選択することが必要である．まず，特性値が計量値か計数値かといったように特性値の性質によって利用する管理図の種類は異なってくる．さらに，特性値が計量値の場合でも，異常を早期に検出したいのか，それとも検出の精度を高くしたいのかといったように活用目的によって利用する管理図の種類は異なってくる．このように，管理図は，ウェブアプリケーションの特性や管理者の要求に応じてさまざまな形で性能異常兆候の検出に適用可能だと考えられる．

図 3.2 が管理図の例である．横軸に日や時刻やロット番号などをとり，縦軸に特性値の統計値（例：平均値，中央値，不良率）をとり，打点していく．図 3.2 は特に上で述べた，工場でのネジ製造工程を管理する場合の例であり，ロットごとにネジの長さ（特性値）の中央値（統計値）を計算し打点していく．その際，工程が正常か異常かを判定するために，図 3.2 にある通りあらかじめ過去に測定した特性値のデータから**中心線**，**上方管理限界線**，**下方管理限界線**の 3 本の基準線を引いておく．

3.3.2 作成手順

本論文では**メディアン管理図**を用いるので，ここではメディアン管理図についてだけ作成手順を説明する．本論文でメディアン管理図を選択した理由は第 3.5.1

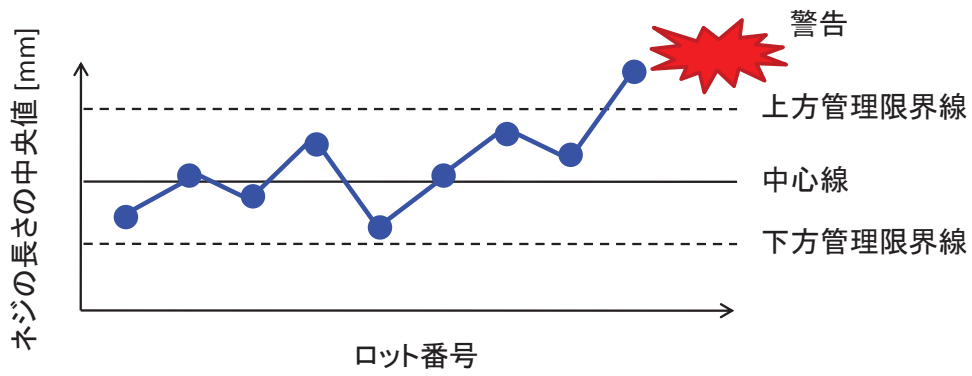


図 3.2: 管理図の例

節で述べる．説明では上で述べた，工場でのネジ製造工程を管理する場合の例も付随させる．

まず初めに基準線を決定するために正常時に 100 個の特性値を測定し，その特性値を測定した順番に 5 個ずつ 20 の群に分類する．ネジ製造工程の例では，ロット番号順に並べた 20 ロットから各 5 個ずつネジを選択し，その長さを測定する．ここで，特性値は 100 個測定するのが慣例となっている．また，群の大きさは管理図の数学的背景により 4 以上を選択する必要があり，中央値の計算を容易にするために通常は奇数である 5 を選ぶ．

そして，各群について中央値 Me と範囲 R（最大値と最小値の差）を計算する．この 20 個の中央値 Me の平均値 $\overline{Me}$ が中心線となる．さらに 20 個の範囲 R の平均値 $\overline{R}$ を計算し，2 本の管理限界線を求める式

$$UCL,\ LCL = \overline{Me} \pm A_2\overline{R}$$

に代入する．UCL が上方管理限界線（Upper Control Limit），LCL が下方管理限界線（Lower Control Limit）である．ここで A_2 は群の大きさによって一意に定まる値であり，群の大きさが 5 のとき A_2 は 0.69 になる．この 2 本の管理限界線を求める式の導出は第 3.4.3 節を参照してほしい．

そして，工程を管理する際には特性値を 5 個測定するごとにその中央値を計算し，管理図の上に打点していく．ネジ製造工程の例では，ネジが 1 ロット完成するごとに 5 個ずつネジの長さを測定し，中央値を計算し，管理図に打点していく．ここで，特性値そのものではなく，特性値の統計値（メディアン管理図では中央値）を打点していくことに注意して欲しい．

3.3.3 警告の発生

打点した特性値の統計値が管理限界線の内側にプロットされ，かつ点の並びにクセがなければ工程は統計的に正常であると判断される．逆に，図 3.2 のように管理限界線を超えて打点されるか，または管理限界線の内側でも打点の並びにクセがあると，工程は異常であるという警告になる．打点の並びのクセとは，中心線とどちらか一方の管理限界線の間に 9 点の連があったり，2 本の管理限界線の内側で 6 点が増加または減少傾向で並ぶことなどである．ただし，本論文のケーススタディで観測した警告は全て管理限界線を超える打点により発せられたものであり，管理限界線の内側での打点の並びのクセによる警告は観測していない．しかし，管理限界線を超えずに少しずつリクエストの処理時間が増大し続けるような性能異常も発生する可能性はあると考えている．

ここで，ウェブアプリケーションのリクエスト処理時間を特性値とした場合に，下方管理限界線を超える打点を性能異常兆候だと検出することについて議論する．通常，リクエストの処理時間は小さいほど性能は良いと判断される．そのため，下方管理限界線を超えた打点が観測された場合に，性能異常の兆候と判断して原因究明を始める必要はないように思える．しかし，異常に処理時間が小さい場合，性能異常の兆候を示している可能性がある．それは，通常行わなければならない処理を行わなかったために，処理時間が小さくなっている可能性があるからである．例えば，本来解放されるべきキャッシュの解放が適切に行われておらず，参照すべきではないキャッシュを参照してしまった場合がこれにあたる．このため，本論文では通常の管理図同様，下方管理限界線を超える打点も警告と判断する．

3.4 管理図の数理

本節では，管理図の数理について簡単に説明する．より詳細な数理に興味のある読者は参考文献 [73, 75] 等を参考にしてほしい．

3.4.1 概要

一般的な管理図では，特性値の統計値の期待値を中心線とし，その上下に特性値の統計値の標準偏差の 3 倍の幅で管理限界線をひくことに定められている [25].

よって，管理限界線は，

$$UCL,\ LCL = E[X] \pm 3D[X]$$

となる．ここで，X は特性値の統計値（例：平均値，中央値，不良率），$E[X]$ は X の期待値，$D[X]$ は X の標準偏差である．

このとき，特性値の統計値 X が正規分布にしたがい，工程が管理状態にあるならば，特性値の統計値 X は管理限界線内に 99.7%，管理限界線外に 0.3%が分布する．よって，特性値の統計値が管理限界線外に打点されれば，ほとんど間違いなく工程に異常が発生していると結論づけることができる．さらに，第 3.3.3 節で述べたように管理限界線の内側に特性値の統計値が打点されていたとしても，その打点のされ方に特定のクセが見受けられた場合は異常が発生していると結論づけ管理図は警告を発する．本論文ではこのクセを判定するルールの数理については詳細に説明せず，説明は参考文献 [73, 75] 等にまかせる．

管理図は特性値の統計値 X に応じてさまざまな種類が存在するが，ここでは最も一般的な管理図である $\overline{X}$ 管理図の数理を説明する．その後で，メディアン管理図の数理についても補足する．その他の管理図の数理については参考文献 [73, 75] 等を参照してほしい．

3.4.2 $\overline{X}$ 管理図

$\overline{X}$ 管理図は特性値 X の平均値 $\overline{X}$ に注目して管理限界線を定める管理図であるため，管理限界線は，

$$UCL,\ LCL = E[\overline{X}] \pm 3D[\overline{X}]$$

となる．

ここで，平均 μ，標準偏差 σ の特性値 X の分布からとった，大きさ n の群の平均値 $\overline{X}$ の分布は平均 μ，標準偏差 $\sigma/\sqrt{n}$ であり，中心極限定理により $\overline{X}$ の分布の形は n が大きくなるほど正規分布に近くなる．実用的には n が 4 くらい以上なら正規分布であるとしても差し支えない [73]．ここで，特性値 X の分布の形によらず，特性値の平均値 $\overline{X}$ の分布の形が正規分布に近くなることに注意して欲しい．これらの値を代入すると，管理限界線は，

$$UCL,\ LCL = \mu \pm 3\frac{\sigma}{\sqrt{n}}$$

となる．

さらに，一般に μ と σ は未知であるため，大きさ n の群の平均値 $\overline{X}$ と範囲 R を用いて推定すると，

$$\mu \leftarrow \overline{\overline{X}}$$
$$\sigma \leftarrow \frac{\overline{R}}{d}$$

となる．ただし，$\overline{\overline{X}}$ は（第 3.3.2 節で述べたように，管理図では通常 20 個の）$\overline{X}$ の平均値，$\overline{R}$ は（第 3.3.2 節で述べたように，管理図では通常 20 個の）R の平均値，d は群の大きさ n によって一意に定まる値である．

よって，$\overline{X}$ 管理図の管理限界線は，

$$UCL,\ LCL = \overline{\overline{X}} \pm 3\frac{\overline{R}}{d\sqrt{n}} = \overline{\overline{X}} \pm A_1\overline{R}$$

となる．ただし，群の大きさ n によって一意に定まる値を $3/d\sqrt{n} = A_1$ とした．

3.4.3 メディアン管理図

本論文では $\overline{X}$ 管理図ではなくメディアン管理図を用いる．$\overline{X}$ 管理図が特性値 X の平均値 $\overline{X}$ に注目して管理限界線を定める管理図なのに対し，メディアン管理図は特性値 X の中央値 Me に注目して管理限界線を定める管理図である．特性値 X の分布が平均 μ，標準偏差 σ だとすると，大きさ n の群の中央値 Me の期待値と標準偏差はそれぞれ，

$$E[Me] = \mu$$
$$D[Me] = \frac{m\sigma}{\sqrt{n}}$$

となる．ただし，m は群の大きさ n によって一意に定まる値である．さらに，μ と σ は大きさ n の群の中央値 Me と範囲 R を用いて推定し，

$$\mu \leftarrow \overline{Me}$$
$$\sigma \leftarrow \frac{\overline{R}}{d}$$

となる．ただし，$\overline{Me}$ は（第 3.3.2 節で述べたように，管理図では通常 20 個の）Me の平均値，$\overline{R}$ は（第 3.3.2 節で述べたように，管理図では通常 20 個の）R の平均値，d は群の大きさ n によって一意に定まる値である．

よって，メディアン管理図の管理限界線は，

$$UCL,\ LCL = \overline{Me} \pm 3\frac{m\overline{R}}{d\sqrt{n}} = \overline{Me} \pm A_2\overline{R}$$

となる．ただし，群の大きさ n によって一意に定まる値を $3m/d\sqrt{n} = A_2$ とした．

3.5 管理図の本研究における適用

3.5.1 メディアン管理図の選択

本論文ではメディアン管理図を用いる．その理由は，メディアン管理図が揺らぎに強いからである．ウェブアプリケーションの処理時間は平常時でも揺らぐことがある．原因はさまざま考えられるが，例えばスケジューリングやロックの挙動に影響を受けている可能性がある．図 3.1 は 10 分間ごとにあるリクエストの種類の処理時間の平均値を計算したグラフである．図 3.1 を見ると，平均値は約 30 ミリ秒から約 120 ミリ秒と大きく揺らいでいることがわかる．ずれに敏感な管理図を用いた場合，これらの揺らぎまで異常と検出してしまい，警告が多発してしまう可能性がある．そこで，個々の管理図の特性をふまえ，測定値のゆらぎに強いメディアン管理図を選択した．

3.5.2 特性値の選択

提案手法では，管理図をウェブアプリケーションの性能異常兆候の検出に適用するに当たって，特性値の選び方を工夫している．その結果，提案手法を利用した場合に作成する管理図は図 3.3 のようになる．その工夫は特性値として個々のリクエスト処理時間ではなく，一定時間ごとの処理時間の統計値を用いるということである．一定時間とは例えば 10 分といった値である．統計値として本論文では平均値，最大値，中央値，最小値の 4 種類を選択した．提案手法ではメディアン管理図を用いるので，つまり，四つの管理図を用意し，平均値の中央値，最大値の中央値，中央値の中央値，最小値の中央値を打点していくことになる．RUBiS の例では統計値が 4 種類でリクエストが 27 種類のため，管理図は 4×27 で 108 作成することになる．

特性値として処理時間の統計値を選択した理由は大きく三つある．一つ目に，統計値を管理対象とすることで，提案手法が揺らぎに対して堅牢になる．個々のリ

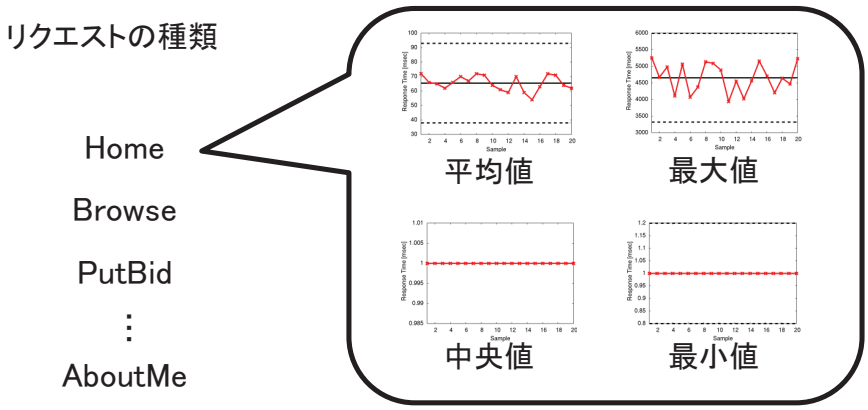


図 3.3: 提案手法で作成する管理図の例

クエスト処理時間では揺らぎが大きく，ある程度まとまった数のリクエスト処理時間の統計値の方が管理対象として適している．二つ目に，統計値を管理対象とすることで，さまざまな種類の性能異常を検出することができる．性能異常はリクエスト処理時間にさまざまな形で現れる．さらに，この工夫により，ウェブアプリケーションの管理者が個々のウェブアプリケーションの性質に応じてリクエストの処理時間を管理できる．アプリケーションによって，管理すべき統計値は異なる．次では，それぞれについて詳しく説明する．

一つめに，統計値を管理対象とすることで，提案手法が揺らぎに対して堅牢になる．現在，ウェブアプリケーションとその提供基盤は非常に複雑であり，全てのリクエストの処理を理想的な時間内に完了させることは現実的ではない．実際にSLA（Service Level Agreement）として処理時間の 95%値に対して閾値の取り決めを行うこともある．例えば，“処理時間の 95%値が 500 ミリ秒以下であることが求められる” といった SLA が見受けられる [76–78]．つまり，残りの 5%のリクエストの処理が理想的に行われなかったとしても，それを許容する取り決めになっている．そこで，提案手法では個々のリクエスト処理時間ではなく，処理時間の統計値が異常な値を取ったときに性能異常として検出することにしている．

二つめに，性能異常の兆候はその原因によってさまざまな形でリクエストの処理時間に表れる．そのため，一般的な性能指標である平均値だけではなく，数種類の統計値を特性値とする．そうすることで，小さな変化を見逃さずに性能異常の兆候を検出する．例えば，同時に処理できる上限数を超える数のリクエストがサーバに到達した場合，性能異常は処理時間の最大値に顕著に現れる．このよう

な事態が発生した場合，一部のリクエストは待ち行列に入れられ，処理時間が異常に大きくなる．一方，待ち行列に入れられなかった大部分のリクエストの処理は平常時と同様に行われるため，処理時間に変化はない．よって，リクエスト処理時間の最大値に異常が顕著に現れる．また，中央値を特性値として用いることにより，多数のリクエストの処理時間がわずかずつ増大もしくは減小したことを検出することができる．平均値のみを特性値としていたのでは，少数の，処理時間が異常に大きなリクエストの影響で，このわずかな変化が隠蔽されてしまう可能性がある．さらに，最小値を特性値として用いることにより，リクエストの処理内容が変化したことによる性能異常の兆候を検出することができるようになる．最小値は通常それほど揺らがない．例えば，資源不足により処理が滞った場合でも，ある程度まとまった数のリクエストの内，少なくとも一つのリクエストは滞りなく処理が行われるのが通常である．最小値に変化が生じたときは，例えばデータベースの肥大化など，リクエストの処理内容に影響を及ぼすような原因により発生した性能異常の兆候の可能性がある．

三つめに，各々のウェブアプリケーションの性質に応じてリクエスト処理時間を管理できる．ウェブアプリケーションの修復を行う際にはリクエストの処理時間を異常発生前と同じ値に回復できることが望ましい．しかし，常にそれを行えるとは限らず，原因によっては困難な場合がある．例えば，クライアント数の増加に伴ってリクエストの処理時間が増大した場合を考える．この場合，サーバの最大同時接続数を増加させることでリクエスト処理時間の最大値は回復できたとしても，中央値は逆に増大してしまうことがある．そのような場合，ウェブアプリケーションの性質に応じて，どちらを優先的に管理するか定める必要がある．例えば，全てのユーザへサービスの最低品質を保証したい場合は最大値を最優先して管理しなければならない．そのため，中央値を犠牲にして最大同時接続数を増加させる必要があると考えられる．個々のリクエスト処理時間を管理していたのでは，この調整を行うのは難しい．提案手法のように統計値を管理対象とする必要がある．また，本論文では管理する統計値を上に挙げた 4 種類に定めているが，ウェブアプリケーションの性質によっては他の特性値を選択できる可能性もあると考えている．例えば，リクエスト処理時間の 95%値や 90%値などがこれに当たる．

リクエスト処理時間の統計値はシステムへの変更を行うことなく計算可能である．そのため，提案手法は既存システムへの導入が容易であるということができる．第 1.3.1 節で述べたように，提案手法ではリクエストごとに URL と処理時間を記録する．この情報を利用し，一定時間ごとにリクエスト処理時間の統計値を

計算することができる．

3.5.3　適用シナリオ

本小節では管理図を用いた性能異常検出の適用シナリオを述べる．第 3.2.2 節で述べたように，管理図で異常の検出を行うためには，正常時のリクエスト処理時間から基準線を計算する必要がある．どの時点で計測した処理時間を “正常” とみなすか，本小節では考えられる典型的なシナリオを挙げる．ここで挙げるのは一例であり，各管理者が，適用するシステムの特徴に応じて基準線を計算することができる．

管理図はウェブアプリケーションの試運用時と実運用時ともに適用できる．図 3.4 を用いてこれを説明していく．まず，試運用時に暫定的な基準線を計算する．もしこの際に管理図が警告を発したならば，原因究明を行い性能異常からの回復と基準線の再計算を行わなければならない．逆に管理図が警告を発しなければ，その時点で計算した基準線を利用し，実運用時にリクエスト処理時間の統計値を管理していくことになる．

実運用時には，試運用時に計算した管理図の基準線を用いてリクエスト処理時間の統計値を管理していく．そして管理図が警告を発した場合は，システムになんらかの変化が生じたと考え，原因究明と回復を行う．しかし，原因究明を行った結果，リクエスト処理時間の変化が避けられず，かつ性能異常がサービスには影響を及ぼさないものと分かることがある．そう判断した場合は，その時点での処理時間を利用して管理図の基準線を再計算することになる．

サービス運用時に発生した処理時間の増大は常に避けなければならないとは限らない．例えば，クライアント数が増加しマシンの計算資源が絶対的に不足してしまった場合，設定の見直しやソフトウェアの最適化だけでは処理時間を回復することができない．そうした場合，新たにマシンを追加する必要がある．しかし，マシンの追加には費用がかかるため，管理者としてはあまり好ましくない．この場合，もし現在の処理時間増大がユーザに悪影響を与えない程度のものであれば，無理にマシンを追加せず現在のシステムでそのままサービスを運用することが可能である．このように，管理者が処理時間の増大を原因究明した上で，サービス運用には問題ないと判断した場合，無理に性能異常を回復させる必要はない．

しかし，処理時間増大前の基準線を用いていたのでは，管理図が無駄な警告を発し続けてしまう．そうすることで，管理者が大事な警告を見逃してしまう可能

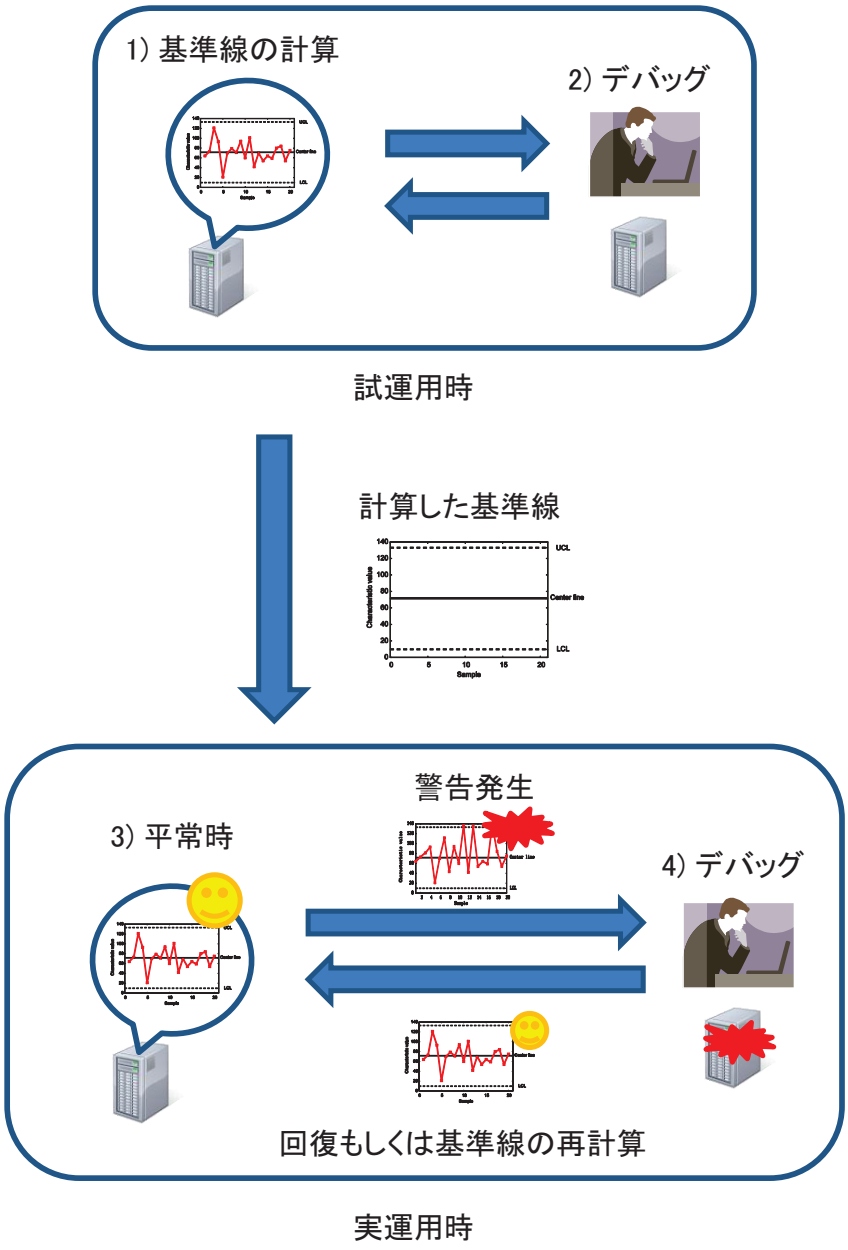


図 3.4: 管理図を用いた性能異常検出の実環境での利用例

性がある．これを避けるために，処理時間増大後のデータを用いて管理図の基準線を再計算し，正常時とみなすデータの分布を更新する必要がある．

上述したように，提案手法では，管理図の基準線を計算するための正常な稼働期間を，最終的には管理者が決定する必要がある．管理図は，現在の特性値が過去の正常時の特性値と比べて統計的に異なっているかどうかを判定する．したがっ

て，もし管理者が正常だと判断した期間に実はすでに性能異常が潜んでいたとしても，それを管理図が検出することはできない．

しかし，本論文で対象とする性能異常は，実運用時にワークロードの変化やサーバ稼働時間の経過に伴って発生する性能異常である．ウェブアプリケーションの管理者は実運用前に十分に試運用を行っているはずである．したがって，その時点での性能は，何らかの理由により理想的な値に比べて劣っていたとしても，管理者が望んだ値になっているはずであると考えている．

また，管理図では，基準線計算時であっても，その期間内に性能が変化するような異常は検出できる．検出することができないのは，その期間内の処理時間が常に大きかったり，常に小さかったりするような性能異常である．実際に，第 5.5 節のケーススタディでは，基準線計算時に発見した性能異常から回復したケースを報告する．

3.6 まとめ

本章では，提案手法の要素技術の一つである，性能異常検出手法を詳細に説明した．提案手法ではリクエストの種類ごとにリクエスト処理時間を監視するため，まずその分類を管理者が行う必要があることを述べた．リクエストは URL で，利用する HTML ファイルや Servlet を指定するため，URL からリクエストの種類を分類できる．

次に，管理図を用いて性能異常を検出することを述べた．まず，管理図を適用する目的が，正常時の揺らぎと異常な変化の兆候を見極めることであることを説明した．そして，管理図の説明を行い，さらに管理図の数学的背景について少し触れた．その後，本研究における管理図の適用方法として，メディアン管理図を用いることと，特性値に処理時間から一定時間ごとに計算した統計値を選んだことについて説明した．最後に適用シナリオを説明し，試運用時に得たデータを正常時の処理時間として与え，実運用時のデータを監視していくことを述べた．

第4章　性能異常分類手法

提案手法では，管理図を用いて性能異常を検出した後，性能異常の分類を行う．そうすることで，管理者が原因究明を行う上で有用な情報を提供する．本章では，この性能異常分類手法という要素技術について説明する．まず性能異常を分類する目的について詳しく説明し，その後，分類を行う上での着目点と分類手順について説明する．さらに，分類手順の各ステップをそれぞれ節に分けて説明する．本章の最後では，分類結果から原因究明を行う際に考慮すべき点をいくつか述べる．

4.1　目的

4.1.1　性能異常の分類

管理図を用いて一度に複数の性能異常を検出した場合，その性能異常の原因が同じか異なるかが分かれば，その後の原因究明をより迅速に行うことができる．このことを図 4.1 を用いて説明する．図 4.1 では，提案手法のように，リクエストの種類（R_1，R_2）ごとに性能異常を検出していると考える．リクエストの種類 R_1 はコンポーネント A と C を利用し，R_2 は B と C を利用する．ここでいうコンポーネントとはマシンやサーバソフトウェア，EJB（Enterprise Java Beans）などであり，特に明確な制限はない．性能異常検出後，原因として怪しいのは異常を検出したリクエストの種類が利用するコンポーネントである．その結果，例えば R_1 で異常を検出した場合はコンポーネント A と C が原因である可能性が高いと分かる．

ここで，リクエストの種類 R_1 と R_2 両方で性能異常が発生した場合，原因として少なくとも 2 通りの場合が考えられる．一つめはコンポーネント C のみが原因で結果としてリクエストの種類 R_1 と R_2 両方で性能異常を引き起こしている場合である．二つめはコンポーネント A と B それぞれが原因で，結果としてリクエストの種類 R_1 と R_2 両方で性能異常を引き起こしている場合である．

上記のような場合，R_1 と R_2 で発生している性能異常の原因が同じか異なるかが分かれば，異常を引き起こしているコンポーネントを素早く絞り込むことができ

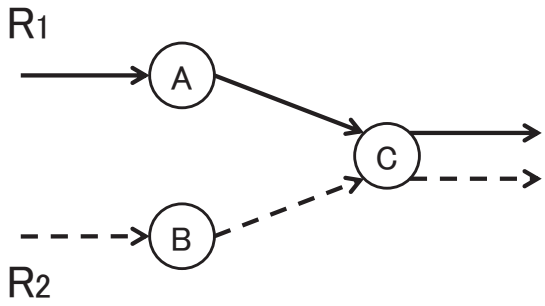


図 4.1: 性能異常分類手法が有効な例

る．原因が同じであれば，コンポーネント C だけが原因である可能性が高い．逆に，原因が異なれば，コンポーネント A と B 両方が原因である可能性が高い．

本論文では性能異常を検出した後の原因究明支援手法として **性能異常分類手法**を提案する．上で述べたように性能異常の原因が同じか異なるかを判定することはその後の原因究明に有益な情報である．提案手法はその判定を自動で行う．性能異常を検出したリクエストの種類を原因ごとに分類し，管理者に分類結果を提供する．

提案手法が保証するのは，次の 2 点である．1 点が，同じクラスタに分類されたリクエストの種類に発生している性能異常の原因が同じであることである．もう 1 点が，異なるクラスタに分類されたリクエストの種類に発生している性能異常の原因が異なることである．

提案手法の貢献は性能異常を原因ごとに分類し，管理者に提示することである．分類結果からシステマティックに原因究明を行う手法は本論文では提案しない．ただし，原因究明時の基本的な考え方として，同じクラスタに分類されたリクエストの種類だけで共通して利用しているコンポーネントを性能異常の原因と考え探索することが有効である．リクエストの種類が同じクラスタに分類されたということは，同じ原因により性能異常が発生しているということを示している．そのため，共通して利用しているコンポーネントが原因と考えられる．また，他のリクエストの種類が異なるクラスタに分類されたということは，異なる原因により性能異常が発生していることを示している．よって，他のクラスタに分類されたリクエストの種類と共通して利用しているコンポーネントは性能異常の原因から除外できる．第 5 章で報告するケーススタディにおいても，同じクラスタに分類されたリクエストの種類だけで共通して利用しているコンポーネントの探索を行っている．

ここで，図 4.1 の例において，リクエストの種類 R_1 と R_2 両方で性能異常が発生した場合に考えられる原因コンポーネントの組み合わせが，実際は 5 種類あることについてふれる．これまで，コンポーネント C のみが原因の場合とコンポーネント A と B 両方が原因の場合の 2 種類の組み合わせについてのみ述べてきた．実際は，これら以外にも次の 3 種類の組み合わせが考えられる．1) コンポーネント A と C 両方が原因の場合，2) コンポーネント B と C 両方が原因の場合，3) コンポーネント A と B と C 全てが原因の場合，の 3 種類である．しかし，提案手法により得られる分類結果と，この 3 種類の原因コンポーネントの組み合わせの関係を説明するには，先に提案手法についての詳細な説明を行う必要がある．したがって，詳細な説明は第 4.3.2 節で行う．

4.1.2 監視を細粒度に行う場合

図 4.1 の例において，コンポーネント単位で処理時間を監視すれば性能異常の分類は不要に見える．性能異常の検出がコンポーネント単位で行われるため，即座に原因コンポーネントを絞り込むことができるからである．一方，提案手法ではシステムへの導入を容易にするため，処理時間の監視をリクエストの種類ごとに行っている．このため，複数のリクエストの種類で性能異常を検出した際には，まず性能異常を原因ごとに分類する必要がある．

しかし実際は，処理時間を細粒度で監視した場合でも，性能異常の分類は必要である．このため性能異常分類手法は，単体で他の手法にも適用可能である．

図 4.1 の例において，コンポーネント A，B，C をより細かく見ていくと，このことがわかる．それを示すのが図 4.2 である．メソッド x はクラス A 内で定義されており，メソッド y と z はクラス C 内で定義されている．さらにメソッド x と y がクラス A で呼び出され，メソッド y と z がクラス B で呼び出されている．

図 4.2 において A，B，C，全てのコンポーネントで性能異常を検出した場合を考える．この時，原因メソッドの組み合わせは 2 種類考えられる．一つめはメソッド y のみが原因となっている場合である．この場合，メソッドを定義しているクラス C だけでなく，メソッド y を呼び出したクラス A とクラス B でも性能異常が検出される．もう一つはメソッド x とメソッド z 両方が原因となっている場合である．この場合，それぞれのメソッドを定義しているクラス A とクラス C だけでなく，メソッド z を呼び出したクラス B でも性能異常が検出される．

本論文で提案する性能異常分類手法を，コンポーネント A，B，C に適用する

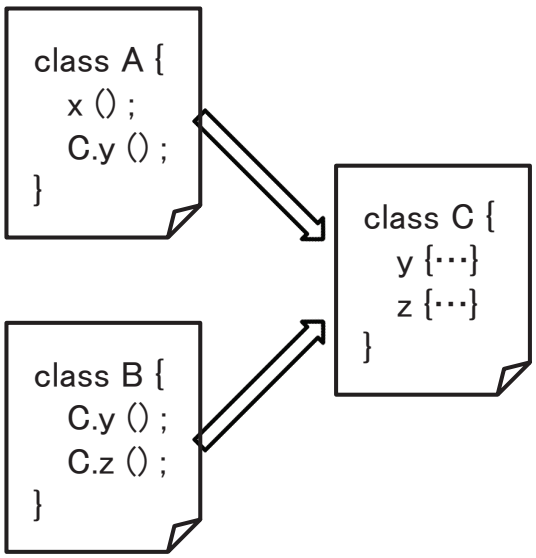


図 4.2: 細粒度に監視を行った場合でも性能異常分類手法が必要であることを示す例

ことで，上記の場合においても原因究明支援を行うことができる．提案手法によりクラス A，クラス B，クラス C 全てのコンポーネントが同じクラスタに分類されれば，共通メソッドであるメソッド y が原因だと分かる．一方，クラス B とクラス C が同じクラスタに分類され，クラス A のみが異なるクラスタに分類されれば，メソッド x とメソッド z がともに原因だと分かる．

性能異常の分類を必要としないためには，監視対象同士で共通して利用する部分が一つもできないほど監視粒度を細かくする必要がある．そのためには監視粒度をクラス，メソッド，ブロック，文と細かくしていくことが求められる．しかし，それは実装コストと実行時オーバーヘッドの観点から現実的ではない．監視対象が細かくなるほど，実装箇所は増加し，実装コストが大きくなる．監視対象が増加するほど，処理時間の計測にかかる処理も増加し，実行時オーバーヘッドは増大する．

ここまで図 4.1 と図 4.2 のように，本研究で提案する性能異常分類手法が有用なケースのみに着目して議論を行ってきた．しかし，性能異常分類手法が常に原因究明に有益な情報を提供できるわけではなく，その適用効果が小さい可能性もある．例えば，性能異常を検出したリクエストの種類同士が，共通して利用するコンポーネントを一つももたない場合がこれに当たる．性能異常の分類を行ったとしてもその後の原因究明に有用な情報は得られない．しかしこの場合でも，有益な情報を得られないまでも原因究明を妨げることにはならず，提案手法を適用す

る際の障害にはならない．

4.2 概要

本研究では，性能異常の分類をリクエスト処理時間の分布の変化傾向に着目して行う．リクエスト処理時間の分布の変化傾向が似ていれば，同じ原因により性能異常が発生していると判断する．逆にリクエスト処理時間の分布の変化傾向が異なれば，異なる原因により性能異常が発生していると判断する．

性能異常は原因によって処理時間の分布に異なる形で現れる．このことを図 4.3 を用いて説明する．図 4.3 は三つのリクエストの種類それぞれのリクエスト処理時間の累積分布関数（CDF）を模式的に表している．左からそれぞれリクエストの種類 1，2，3 の処理時間の分布を表している．グラフは横軸が処理時間，縦軸が割合である．青い CDF が性能異常検出前の処理時間の CDF，赤い CDF が性能異常検出後の処理時間の CDF である．また，黄色の矢印が性能異常検出前後での処理時間の増大を表している．リクエスト 1 と 2 で発生している性能異常は全てのリクエストの処理時間を少しずつ増大させ，リクエスト 3 で発生している性能異常は一部のリクエストの処理時間を大きく増大させていることが見て取れる．この場合，リクエスト 1 と 2 には同じ原因により性能異常が発生し，リクエスト 3 にはそれとは異なる原因により性能異常が発生していると考えることができる．

図 4.1 の例に当てはめれば，性能異常発生前後で R_1 と R_2 の処理時間の分布が同様の変化傾向を示していれば，同じ原因，つまりコンポーネント C により性能異常が引き起こされている可能性が高いと結論づけることができる．逆に R_1 と R_2 の処理時間の分布が異なる変化傾向を示していれば，異なる原因，つまりコンポーネント A とコンポーネント B それぞれにより別の性能異常が引き起こされている可能性が高いと結論づけることができる．

提案手法では三つのステップを設けて，リクエストの種類を性能異常の原因ごとに分類する．まず初めに，各リクエスト処理時間の分布の変化傾向を**性能異常シグネチャ**として抽出する．次に，全てのリクエストの種類同士で性能異常シグネチャの類似度を計算する．例えば，性能異常を 5 種類のリクエストで検出した場合，${}_5C_2 = 10$ 通りの組み合わせで性能異常シグネチャの類似度を計算する．最後に，計算した性能異常シグネチャの類似度を基にリクエストの種類をクラスタリングする．クラスタリングには**階層的クラスタリング**の**群平均法**を用いる．リクエストの種類同士が同じクラスタに分類されれば，それらに発生している性能

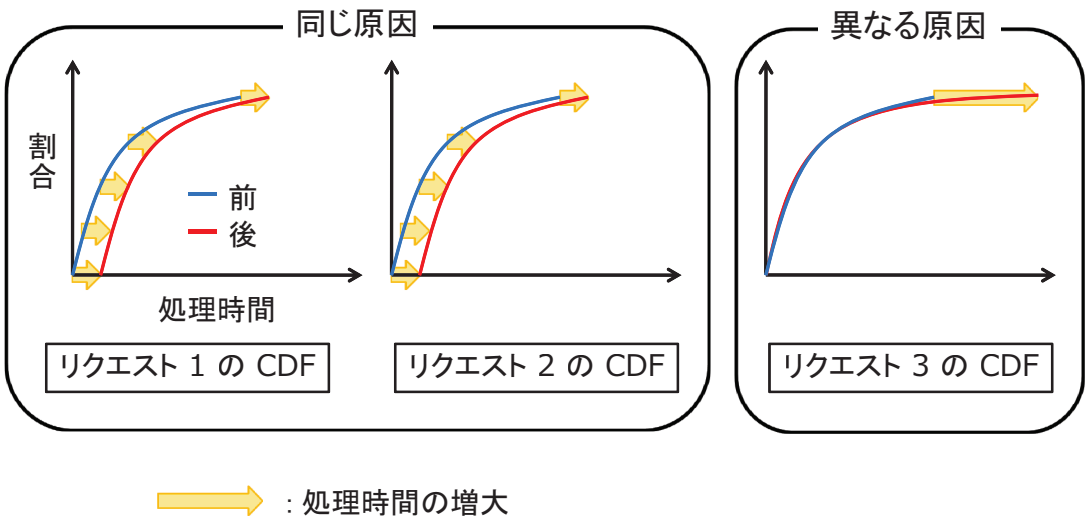


図 4.3: 性能異常が原因によってリクエスト処理時間の分布を異なる形で変化させる例

異常の原因は同じだと考えられる．逆に，リクエストの種類同士が異なるクラスタに分類されれば，それらに発生している性能異常の原因は異なると考えられる．本章では，この三つのステップをそれぞれ節に分けて説明する．

性能異常シグネチャには，処理時間以外の性能指標の変化傾向を用いることができる可能性がある．本論文では，実験で観測した性能異常を調査した結果，処理時間の分布の変化傾向を性能異常シグネチャとして用いることにした．しかし，リクエスト処理時間が常に性能異常シグネチャに最適な指標だと主張しているわけではない．処理時間以外にも，一定時間ごとのアクセス数や一定時間ごとの CPU 使用率に対して CDF を作成し，変化傾向をシグネチャとして抽出することも可能だと考えている．

また，処理時間の分布の変化傾向を抽出するために，CDF の代わりにヒストグラムなど他の方法を用いることも可能であると考えている．提案手法では CDF を選択したが，重要なのは処理時間の分布の変化傾向を抽出することであり，ウェブアプリケーションの特性に応じては他の方法を利用することも考察すべきであると考えている．

4.3　性能異常シグネチャ

提案手法で重要となる性能異常シグネチャは処理時間の “分布” の変化傾向を抽出する．処理時間の分布に着目することで，ある程度変化傾向を詳細にとらえつつ，かつ無駄な情報を排除することを目標にしている．単純に，平均値や最大値など統計値の変化量をシグネチャにしたのでは，重要な変化傾向が失われてしまう可能性がある．平均値の増大量が同じだった場合でも，全てのリクエスト処理時間が少しずつ増大している場合もあれば，一部のリクエスト処理時間のみ大きく増大している場合もある．逆に，より詳細な情報を得ようと，性能異常発生前後で得られた処理時間を計測順に並べ，差を計算したものを性能異常シグネチャとしたのでは，余分な情報を含んでしまう．結果として，性能異常を特徴付ける変化が余分な情報により隠蔽されてしまう可能性がある．具体的には，スケジューリングなどの影響がシグネチャに含まれてしまう．

提案手法では性能異常シグネチャを図 4.4 のように棒グラフの形式で表す．この棒グラフが，処理時間の分布の変化傾向を抽出したものである．各クラスにおける増大分を計算し，紫の棒グラフで表している．横軸はリクエスト処理時間の増大分であり，単位はミリ秒である．縦軸はクラス番号である．グラフを見やすくするために，図 4.5，図 4.6 とは横軸のスケールが異なっている．

このシグネチャは，性能異常発生前後のリクエスト処理時間の累積分布関数（CDF）を基に作成する．CDF に生じた変化に着目することで，分布の変化傾向を抽出する．CDF は統計値と異なり分布を表すため，分布の変化傾向を抽出するのに適している．また，処理時間の大きさでソートを行うため，スケジューリングの影響など無駄な情報を排除するのにも適している．

4.3.1　作成手順

ここでは，図 4.5 と図 4.6 を用いて，図 4.4 の性能異常シグネチャを導出する過程を説明する．

1. 図 4.5 のように，性能異常発生前後それぞれのリクエスト処理時間で CDF を作成する．横軸はリクエスト処理時間であり，単位はミリ秒である．縦軸はパーセントである．青い線が性能異常発生前の CDF，赤い線が性能異常発生後の CDF である．

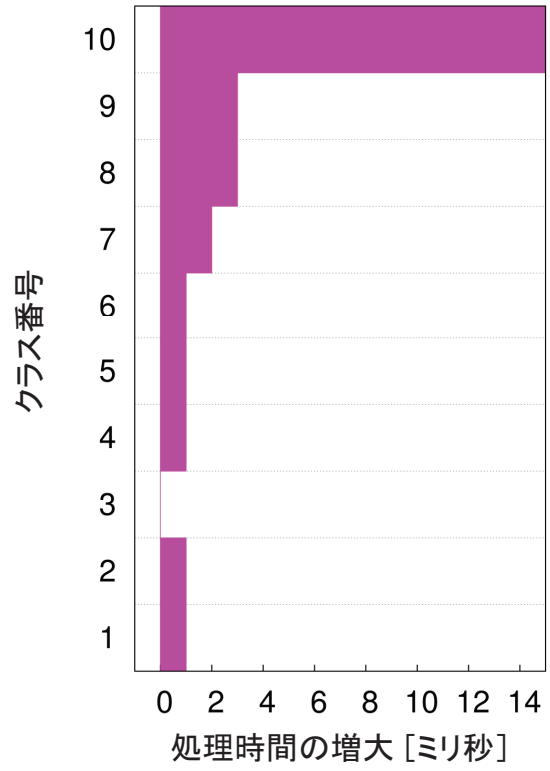


図 4.4: 性能異常シグネチャ

2. 図 4.6 のように，図 4.5 で得た CDF を単純化する．縦軸を n クラスに分割し，各クラスに含まれるリクエスト処理時間の平均値を，各クラスの代表値とする．横軸はリクエスト処理時間の平均値であり，単位はミリ秒である．縦軸はクラス番号である．青い打点が，性能異常発生前の CDF から計算した平均値である．赤い打点が，性能異常発生後の CDF から計算した平均値である．こうすることで，次のステップで CDF の変化傾向を抽出できるようにしている．図 4.6 は n を 10 としたときの例である．ケーススタディでは n を 100 に設定した．

3. 図 4.6 のように単純化した CDF の各クラスの増大量を計算し，図 4.4 に示すシグネチャ棒グラフの各クラスの高さとする．つまり，各クラスにおいて，性能異常発生後の処理時間の平均値から性能異常発生前の処理時間の平均値を引く．そして，その値を，各クラスにおける棒グラフの高さとする．この処理を行うために，前のステップで CDF を単純化している．CDF の各標本の増大量を計算するためには，二つの CDF の標本数が同じである必要があ

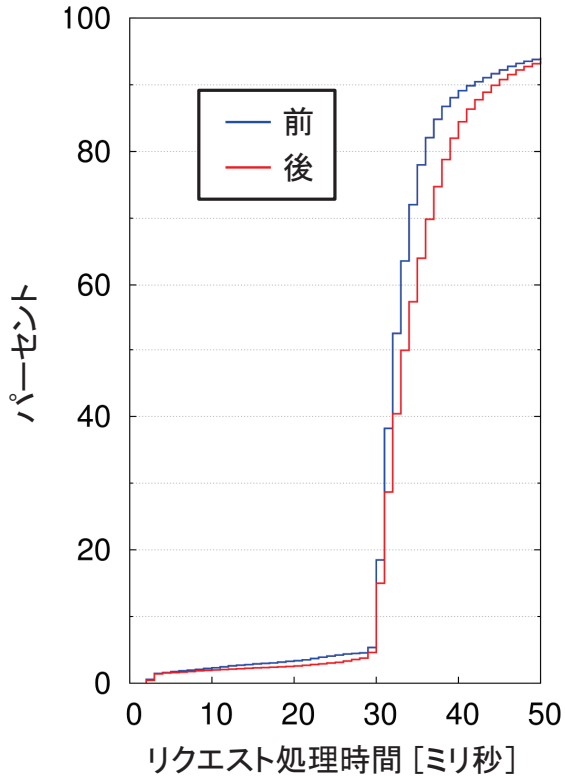


図 4.5: 性能異常発生前後の CDF

る．しかし，通常，性能異常発生前後それぞれで処理されるリクエスト数は異なる．そのため，両方の CDF の標本数を n に変更している．

以上のようにして得たシグネチャ棒グラフは，性能異常発生前後での処理時間の分布の変化傾向を表している．この棒グラフを見れば，図 4.3 で示した例のように，全てのリクエストの処理時間が増大していることや，一部のリクエストのみ処理時間が増大していることがわかる．よって，このシグネチャが類似していれば，それは処理時間の変化傾向が似ているということである．つまり，それらのリクエストの種類に発生している性能異常の原因が同じである可能性が高いといえる．

4.3.2　正しい分類を行えない可能性

ただし，本論文で提案する性能異常シグネチャは，常に正しい分類結果のみを出力できるとは限らない．次の三つのどれかに当てはまる場合，誤った分類結果

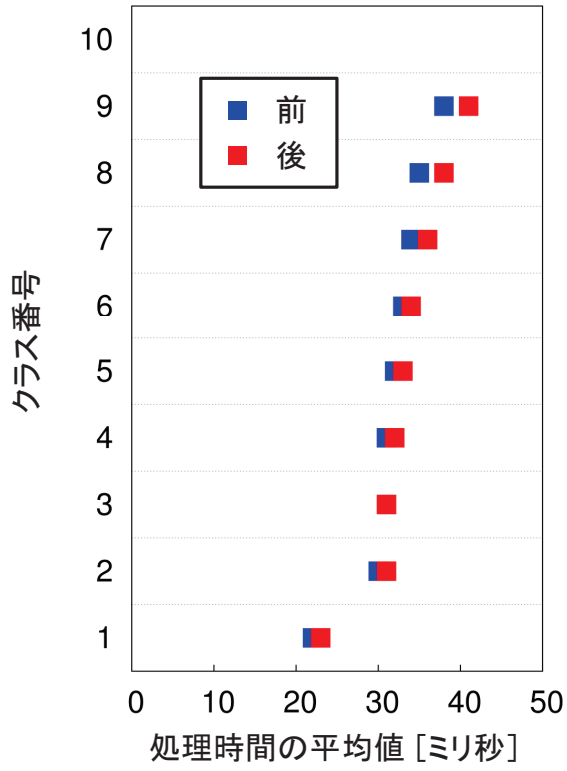


図 4.6: 単純化後の CDF

を出力してしまう可能性がある．次では，それぞれについて述べる．

まず一つ目は，性能異常の原因が同じにも関わらず，処理時間の分布の変化傾向が異なる場合である．われわれが行った実験では，このタイプの性能異常は発生しなかった．しかし，その可能性が全くないとは言い切れない．もしこのようなタイプの性能異常が発生した場合，後の第 4.5 節で述べるように，クラスタリングの閾値を変更する必要がある．閾値を小さくすることで，性能異常シグネチャの類似度がそれほど高くないリクエストの種類同士を，同じクラスタに分類することができる．

本論文で利用するクラスタリング手法では，類似度が事前に定めた閾値以上の場合にシグネチャ同士を同じクラスタに分類する．閾値を小さな値に定めれば，類似度があまり高くないシグネチャ同士が同じクラスタに分類される．閾値を大きな値に定めれば，類似度が非常に高いシグネチャ同士のみ同じクラスタに分類される．

二つ目は，性能異常の原因が異なるにも関わらず，偶然，処理時間の分布の変

化傾向が似通った場合である．この場合，上記の場合とは逆に，クラスタリングの閾値を大きくする必要がある．そうすることで，性能異常シグネチャ同士の類似度が高いリクエストの種類のみを同じクラスタに分類できる．

このように，提案手法による分類結果を利用して原因究明が行えなかった場合，状況によっては，クラスタリングの閾値を変更しながら原因究明とクラスタリングを繰り返し行う必要がある．ただし，第 5 章のケーススタディで報告するように，本論文で報告する三つのケーススタディでは，これを行う必要はなかった．全てのケーススタディで閾値を 60 に定め，適切に性能異常の分類を行うことができた．よって，上述した二つの事態が発生する可能性はそれほど高くなく，提案手法の有用性を否定するものではないと考えている．さらに，もし最初に出力された性能異常の分類結果が誤っていたとしても，提案手法による支援が全くない場合に比べると，原因究明の難易度は軽減するとも考えている．閾値を変更後，再度クラスタリングを行うことになったとしても，原因究明に利用できる情報が全くない場合に比べると，円滑に原因究明を行える可能性が高い．

三つ目は，一つのリクエストの種類に発生している性能異常の原因が複数存在する場合である．この場合，あるリクエストの種類の性能異常シグネチャには複数の性能異常が混在した形で現れてしまう．そのため，それが一つの原因によるものなのか，それとも複数の原因によるものなのかを見極めることはできない．図 4.1 の例を用いると，1) コンポーネント A と C が原因で性能異常が発生，2) コンポーネント B と C が原因で性能異常が発生，3) コンポーネント A と B と C 全てが原因で性能異常が発生，の三つの場合がこれに当たる．全ての場合において共通コンポーネント C が原因の一つであるにも関わらず，R_1 と R_2 は異なるクラスタに分類されてしまう．

このような事態が発生した場合は，原因究明とクラスタリングを繰り返し行うことで，最終的に全ての原因を見つけ出すことができる．まず R_1 と R_2 が異なるクラスタに分類されたということで，少なくとも A か B どちらかが原因であることが分かる．そして，原因を究明し処理を再開後，もう一度提案手法を用いてクラスタリングを行う．ここで 1) と 2) の場合であれば，R_1 と R_2 が同じクラスタに分類され，C の原因究明に進むことができる．もし 3) の場合であれば，R_1 と R_2 がもう一度異なるクラスタに分類され，コンポーネント A もしくは B に着目して原因究明を進め，再度クラスタリングを行うという手順になる．このように，一度に全ての原因を究明することはできないものの，提案手法による支援が無い場合に比べ，原因究明の難易度を軽減することができる．

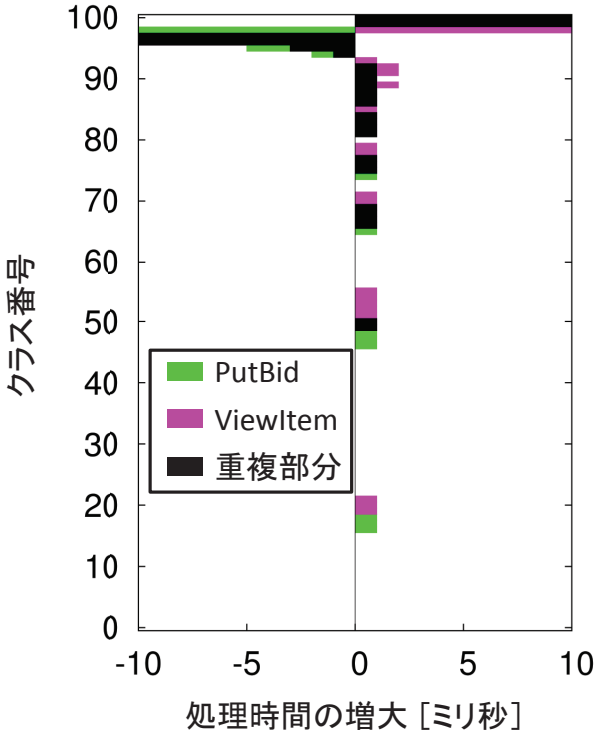


図 4.7: 二つの性能異常シグネチャを重ね合わせた例

4.4 シグネチャ間類似度の計算

4.4.1 概要

提案手法では二つのシグネチャ棒グラフを重ね合わせ，重なる面積が大きいほど二つのシグネチャが類似していると判断する．図 4.7 が二つのシグネチャを重ね合わせた例である．横軸はリクエスト処理時間の増大分であり，単位はミリ秒である．縦軸はクラス番号である．緑の棒グラフは `PutBid` というリクエストの種類で得たシグネチャである．紫の棒グラフは `ViewItem` というリクエストの種類で得たシグネチャである．二つのシグネチャが重なった部分を黒く表示している．黒い領域の面積が大きいほど二つのシグネチャが類似していると判断する．

これを，性能異常が発生している全てのリクエストの種類で行う．例えば，五つのリクエストの種類で性能異常を検出した場合，${}_5C_2 = 10$ 通りの組み合わせでシグネチャ同士の類似度を計算することになる．

ただし，シグネチャの類似度は重なる面積を 0 以上 100 以下に正規化して得る．なぜならば，単純に重なる面積を計算しただけでは，類似度を正しく測ることが

できないからである．重なる面積は，個々のシグネチャ棒グラフの面積が大きいほど大きくなる．元の棒グラフの面積が大きいか小さいかに関わらず，“類似度”を計算する必要がある．

提案手法では，重なる面積を正規化する方法を 2 種類用意した．そうすることで，シグネチャに現れるスパイクを柔軟に扱うことができる．2 種類の正規化はそれぞれ，**大域的な正規化**と**局所的な正規化**と呼ぶ．前者は大域的に正規化することで，異常に大きな値をもつクラスの影響が大きく反映され，スパイクにより注目することができる．一方，後者は局所的に正規化することで，全てのクラスから受ける影響を等しくし，スパイクに大きく影響されるのを避ける．そうすることで，小さな変化にも注目することができる．これら二つを効果的に組み合わせることで，性能異常をより効果的に分類することができる．

4.4.2 大域的な正規化

図 4.8 を用いて，大域的な正規化がスパイクに注目できることを示す．図 4.8 の左側には三つのシグネチャA，B，C が並んでいる．そして，右側にはシグネチャ A と B を重ね合わせたグラフとシグネチャC と B を重ね合わせたグラフが上から並んでいる．横軸はリクエスト処理時間の増大分であり，単位はミリ秒である．縦軸はクラス番号である．左側は，三つのシグネチャである．水色のグラフがシグネチャA，紫のグラフがシグネチャB，緑のグラフがシグネチャC である．A のシグネチャは 10 番目のクラスのみ処理時間が 20 ミリ秒増大している．B のシグネチャは 1〜9 番目のクラスで処理時間が 1 ミリ秒増大，10 番目のクラスのみ処理時間が 20 ミリ秒増大している．C のシグネチャは全てのクラスで処理時間が 1 ミリ秒増大している．右側はシグネチャ同士を重ね合わせたグラフであり，重なった部分を黒く表示している．二つの正規化を用いて得られる類似度を図中に示した．

シグネチャA と B には 10 番目のクラスにスパイクが存在する．大域的な正規化を用いて計算したシグネチャの類似度はこのスパイクによる影響が大きく反映される．

大域的な正規化を用いる場合，シグネチャX と Y の類似度 $G(X,Y)$ を次のように計算する．

$$G(X,Y) = \frac{\sum_{1 \leq i \leq n} X_i \cap Y_i}{\sum_{1 \leq i \leq n} X_i \cup Y_i} \times 100$$

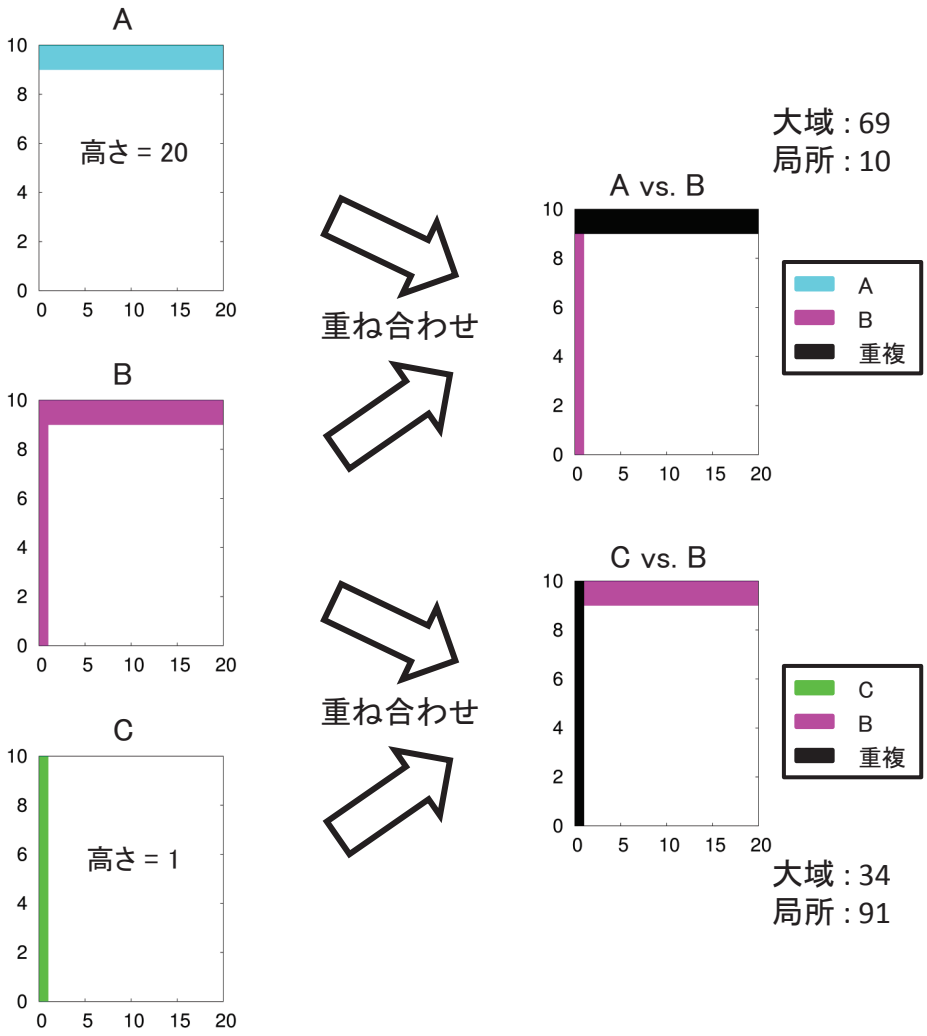


図 4.8: 大域的な正規化と局所的な正規化

ここで n はクラス数である（図 4.8 では 10）．$X_i \cap Y_i$ はシグネチャ X と Y の i 番目のクラスにある二つの棒の共通部分（重なる面積）を表し，$X_i \cup Y_i$ は二つの棒を合併させてできる棒の面積である．図 4.8 の例ではそれぞれ，$G(A,B) = 20/29 \times 100 = 69$, $G(B,C) = 10/29 \times 100 = 34$ となる．

シグネチャ A と B には同様のスパイクが存在するので，A と B の類似度は B と C の類似度に比べて大きくなる．異常に大きな値をもつ 10 番目のクラスの影響が，分母と分子両方に大きく反映される．その結果，$G(X,Y)$ は 10 番目のクラスの影響を大きく受けた値となる．

4.4.3 局所的な正規化

一方，局所的な正規化を用いて計算することで，少しずつ，たくさんのクラスにまたがる変化の影響を大きく反映することができる．図 4.8 でシグネチャBとCは 1 番目から 9 番目のクラスの棒の高さが 1 である．つまり，わずかな変化がたくさんのクラスにまたがっている．この変化に，局所的な正規化を利用して着目する．

局所的な正規化を用いる場合，シグネチャXとYの類似度$L(X,Y)$を以下のように計算する．

$$L(X,Y) = \sum_{1 \leq i \leq n, \neg(X_i = Y_i = 0)} \frac{X_i \cap Y_i}{X_i \cup Y_i} \times \frac{100}{n - \#z}$$

ここで$\#z$は両方の棒の高さが 0，つまり$X_i = Y_i = 0$となるクラスの数である．図 4.8 ではそれぞれ，$L(A,B) = (0/1 \times 9 + 20/20 \times 1) \times 10 = 10$，$L(B,C) = (1/1 \times 9 + 1/20 \times 1) \times 10 = 91$となる．

わずかな変化がたくさんのクラスにまたがっているため，$L(B,C)$が$L(A,B)$に比べて大きくなっている．局所的に正規化を行うことで，全てのクラスから受ける影響を等しくできる．そうすることで，スパイクによる影響を抑えられていることがわかる．

ここで，$\#z$について詳しく説明する．X_iとY_iの値がともに 0 のときは注意が必要である．$X_i = Y_i = 0$のときは$X_i \cup Y_i = 0$となり，$X_i \cap Y_i / X_i \cup Y_i$が定義できない．この場合，クラス$i$の値はともに 0 であり等しいので，直感的には$X_i \cap Y_i / X_i \cup Y_i$を 1 と定義するのが正しく思える．しかし，この定義では類似度を正しく計算することができない．

二つのシグネチャSとTを考えてほしい．ここで，Sは全てのクラスの値が 0，Tは半分のクラスの値が 0，残り半分のクラスの値が 1 とする．このとき，値がともに 0 のクラスiにおいて$X_i \cap Y_i / X_i \cup Y_i$を 1 とすると，$S$と$T$の類似度は 50 となり，性能異常の発生していないリクエストの種類のシグネチャSと性能異常が発生しているリクエストの種類のシグネチャTの類似度が大きな値になってしまう．この事態が発生するのを避けるために，提案手法では，ともに値が 0 になるクラスを計算から排除している．さらに類似度が 0 以上 100 以下であるのを保つために，$100/(n - \#z)$を掛けている．

局所的な正規化には，処理時間に現れる自然のゆらぎに類似度が大きく影響されてしまうのを避ける効果もある．リクエスト処理時間は CPU や I/O のスケジュー

リングによって正常時もゆらぐ．このような影響を受け，一部のリクエスト処理時間は正常時も他のリクエスト処理時間に比べて異常に大きな値となる．もちろん，この一部のリクエスト処理時間は，CDF の上側に集まることになる．さらに，この一部のリクエスト処理時間は他のリクエスト処理時間に比べて大きくゆらぐ．そのため，性能異常シグネチャの上側の数クラスの値を異常に大きくする傾向がある．つまり，棒グラフに現れたスパイクを無視することは，自然のゆらぎによる影響を避けることにもつながる．例えば，図 4.8 においてシグネチャ A と B がゆらぎによる影響を受け，10 番目のクラスの棒の高さが異常に大きな値になっているとする．この場合でも，$L(A, B)$ は 10 にとどまっている．

4.4.4　シグネチャ同士の位置合わせ

これまでに説明した方法でシグネチャ同士の類似度を計算するだけでは，類似度が求めるよりも小さくなってしまうことがある．図 4.7 の 20 番目付近と 50 番目付近のクラスに注目してほしい．二つのシグネチャは似ているが，重なる領域は小さい．よって，この付近の類似度は小さな値になってしまう．同じ原因による性能異常だとしても，処理時間の分布の変化傾向が全く同じになるとは限らず，このようにわずかにずれが生じる可能性がある．この場合は，シグネチャを縦方向にずらすことで重なる面積を大きくし，類似度を大きくすることが求められる．

しかし，その一方で単純にずらして重なった領域をそのまま足し合わせたのでは，逆に類似度が大きくなりすぎてしまう．これを避けるために，提案手法では二つの棒を重ねるためにずらした数を用いてペナルティを与える．また，片方のシグネチャのあるクラスの棒グラフに他方のシグネチャの複数の棒グラフを重ねた場合も，重なった数を用いてペナルティを与える必要がある．

4.5　クラスタリング

提案手法では，シグネチャ間の類似度を基に**階層的クラスタリング**の**群平均法**を用いて性能異常を分類する．この手法は，クラスタリングのために広く一般的に用いられている手法である．

前のステップですでに，2 種類の正規化それぞれを用いて，性能異常が発生しているリクエストの種類全てのシグネチャ同士の類似度を得た．ここでは，クラスタリング手法を大域的な正規化により得た集まり，局所的な正規化により得た集

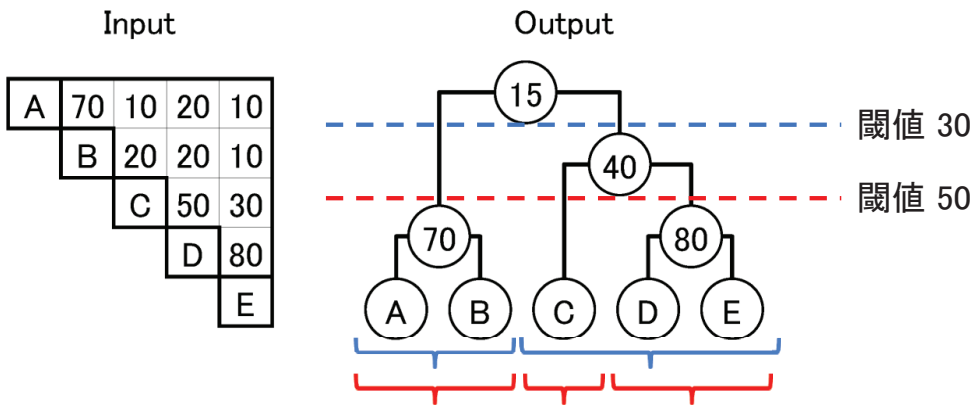


図 4.9: クラスタリングの入力と出力

まりのそれぞれに対して別々に適用する．つまり，利用者は分類結果を 2 種類得ることになり，目的に合わせてそれぞれを参考に性能異常発生箇所の特定を行う．

4.5.1 手順

階層的クラスタリングでは，図 4.9 の左側に示した入力となる表から右側に示した出力木が得られる．表は各シグネチャ同士の類似度を表しており，例えば A と B の類似度は 70，B と D の類似度は 20 である．出力は全二分木の形式を取る．葉は各リクエストの種類であり，ノードは自分の子孫である葉を含むクラスタを示す．

出力木は以下のようにして得られる．まず始めに，全ての葉は自分自身のみを含むクラスタとみなす．その後，最も類似度の大きい二つのクラスタを結合する．図 4.9 の例では D と E の類似度が 80 で一番大きいので D と E が結合され，$[A]$，$[B]$，$[C]$，$[D,\ E]$ と四つのクラスタになる．ここで，クラスタ間の類似度はそれぞれのクラスタに含まれる監視対象のシグネチャ同士の類似度を全ての組み合わせについて計算し，それらの平均となる．例えば，$[A]$ と $[D,\ E]$ の類似度は $(20 + 10)/2 = 15$ である．以後，クラスタが一つになるまでこの処理を繰り返す．図 4.9 の例では $[A,\ B]$，$[C]$，$[D,\ E]$ から $[A,\ B]$，$[C,\ D,\ E]$ となり，最後に $[A,\ B,\ C,\ D,\ E]$ となりクラスタの結合は終了する．

最後にクラスタ結合の閾値を定めることで出力木からクラスタを得ることができる．閾値を 50 に定めれば，得るクラスタは $[A,\ B]$，$[C]$，$[D,\ E]$ の三つとなり，30 に定めれば，$[A,\ B]$，$[C,\ D,\ E]$ の二つとなる．第 5 章では閾値を 60 に定めて実験を行った．

4.5.2　閾値の決定

第 4.3.2 節で述べたように，閾値によってクラスタリングの厳密さを変更することができる．閾値を大きくすれば，シグネチャの類似度が非常に高いリクエストの種類同士のみが同じクラスタに分類される．つまり，処理時間の分布の変化傾向が非常に似たリクエストの種類同士のみが同じクラスタに分類される．逆に閾値を小さくすれば，シグネチャの類似度がそれほど大きくないリクエストの種類同士も同じクラスタに分類されるようになる．つまり，処理時間の分布の変化傾向がそれほど厳密に似ていなくても同じクラスタに分類されることになる．

閾値は提案手法の精度に関わる重要な値である．しかし，閾値は必ずしも固定する必要はなく，閾値の調整とクラスタリングを繰り返しながら原因究明を進めることができる．一つの選択肢として，閾値を大きな値から徐々に小さくしていきながら原因究明を行う方法がある．閾値が大きいほど，発生している性能異常の類似度が大きいリクエストの種類のみが同じクラスタに分類され，結果として個々のクラスタは小さくなる．そのため，クラスタに含まれるリクエストの種類同士の共通部分や，クラスタに含まれないリクエストの種類との排反部分を探すのがいくぶん容易である．このような使い方をすることで，閾値決定の問題を回避することができる．

4.5.3　分類対象のリクエストの種類

ここで注意したいのは，性能異常の分類を，性能異常を検出したリクエストの種類に対してだけでなく，全てのリクエストの種類に対して行うということである．ここまでは便宜的に，2 番目のステップで性能異常を検出したリクエストの種類だけをクラスタリングするように説明してきた．しかし実際は，性能異常を検出しなかったリクエストの種類も含めてクラスタリングを行う．これによって，原因究明支援の能力を高められる可能性がある．

性能異常の原因となっているコンポーネントを利用していたとしても，そのリクエストの種類に必ずしも異常が明確に現れるとは限らない．例として，軽度なコネクション不足による性能異常が挙げられる．軽度なコネクション不足が発生した場合，大部分のリクエストは通常通りに処理が行われ，一部の，コネクションが偶然不足していたときにサーバに到達したリクエストのみ処理に時間がかかる．この場合，コネクションが不足しているときに偶然多くのリクエストが発行

されたリクエストの種類と，そうでないリクエストの種類では処理時間の増大量が異なる．前者では大きく処理時間が増大し，後者では処理時間の増大はそれほど大きくない．よって，性能異常検出手法は前者のみで異常を検出する可能性がある．この状態で性能異常の分類を行ってしまうと，同じ原因を共有しているリクエストの種類同士にも関わらず同じクラスタに分類されないことになり，管理者へ提供する情報の有用性が低下してしまう．

しかし，この事態を分類手法が補うことができる可能性がある．性能異常検出手法は処理時間の増大量に注目して異常を検出する．それに対し，性能異常分類手法は処理時間の分布の変化傾向に着目する．増大量が小さくて異常を検出できなかったとしても，分布の変化傾向は似る可能性がある．そのため，性能異常を検出できなかったリクエストの種類が，性能異常を検出したリクエストの種類と同じクラスタに分類される可能性がある．上記のコネクション不足の例に当てはめれば，コネクションが不足していたときにあまりリクエストが発行されなかったリクエストの種類も，コネクションが不足していたときに多くリクエストが発行されたリクエストの種類と同じクラスタに分類される可能性がある．

そして最後に，性能異常を検出しなかったリクエストの種類のみが含まれるクラスタを取り除き，性能異常分類手法の出力とする．上述の理由により，性能異常を検出しなかったリクエストの種類も含めてクラスタリングを行う．しかし，正常なリクエストの種類同士が同じクラスタに分類された場合は，偶然処理時間の分布の変化傾向が似ただけであり，特別な意味を持たない．

4.6　分類結果を利用した原因究明

提案手法を利用してリクエストの種類を性能異常の原因ごとに分類した後，管理者は人手で原因究明を進めていく．分類結果が保証するのは次の 2 点である．

1. 同じクラスタに分類されたリクエストの種類同士に発生している性能異常の原因は同じ
2. 異なるクラスタに分類されたリクエストの種類同士に発生している性能異常の原因は異なる

管理者は先の 2 点に注目して，性能異常の原因究明を進めていくことができる．

ただし，本論文の提案手法により行えるのは原因の絞込みである．巨大なシステムの内，原因と考えられるコンポーネント群を絞り込み，その後の原因究明に

かかる手間を削減することが目的である．最終的な原因の特定は，他の手法やデバッガを用いて行う必要がある．

4.6.1 共用コンポーネントの探索

分類結果を得た後，まず始めに管理者が行うべきことは，同クラスタに含まれたリクエストの種類が共通して利用しているコンポーネントを探索することである．発生している性能異常の原因が同じであるため，共用コンポーネントが原因だと考えられる．

性能異常を検出しなかったリクエストの種類に注目することで，原因の可能性があるコンポーネントをさらに絞り込むことができる．同クラスタに含まれたリクエストの種類の共用コンポーネントの内，性能異常を検出しなかったリクエストの種類からも共用されているコンポーネントは原因とは考えられない．なぜならば，性能異常を検出しなかったリクエストの種類が利用しているコンポーネントは正常だと考えられるからである．

本来，異なるクラスタに分類されたリクエストの種類によって共用されているコンポーネントも，原因とは考えられないはずである．異なるクラスタに分類されたため，性能異常の原因は異なると考えられるからである．しかし，第 4.3.2 節で述べたように，性能異常の原因が複数存在する場合，異なるクラスタに分類されたリクエストの種類同士が性能異常の原因を共有している可能性がある．そのため，機械的に原因の可能性から排除してしまうのではなく，注意深く調査する必要がある．

共用コンポーネントは，性能異常によって粒度が異なる．Enterprise Bean やメソッドなど細かい粒度の場合もあれば，マシン単位など粗い粒度の場合もある．探索の際には，同クラスタに分類されたリクエストの種類が多いか少ないかに留意しながら共用コンポーネントを探すことも重要である．同クラスタに含まれるリクエストの種類が多い場合は，多数のリクエストの種類に共用されているコンポーネントが原因だと考えられるため，マシン単位など粗い粒度で原因究明を進めるべきである．逆に同クラスタに含まれるリクエストの種類が少ない場合は，少数のリクエストの種類に共用されているコンポーネントが原因だと考えられるため，Enterprise Bean やメソッド単位など細かい粒度で原因究明を進めるべきである．

提案手法の分類結果を利用して管理者が行えるのは，原因の絞込みである．必ずしも，原因を一意に特定できるとは限らない．同じクラスタに分類されたリク

エストの種類が共通して利用するコンポーネントは，複数存在する可能性がある．この場合，その複数コンポーネントの内どのコンポーネントが原因であるかは，提案手法の分類結果のみから知ることはできない．導入の容易さと原因究明能力にはトレードオフがあり，より強力な原因究明支援を行うにはさらに細かい粒度で処理時間の監視を行う必要がある．ただし，このような要求があった場合でも，性能異常分類手法単体であれば，監視粒度に関わらず適用可能である．このことは，第 4.1.2 節で述べた．そのため，提案手法の有用性が全くないわけではない．

また，同じクラスタに分類されたリクエストの種類同士が共通して利用しているソフトウェアコンポーネントを探索することは必ずしも容易だとは限らない．第 4.1 節の図 4.1 と図 4.2 で示した例は，ウェブアプリケーション内の一部を簡略化した例である．実際のウェブアプリケーションはより複雑なため，特定のリクエストの種類同士が共通して利用しているソフトウェアコンポーネントを探索することは容易ではない．しかし，この探索を自動化する手法が Chen ら [17] などによって提案されつつあり，管理者はこれらの手法を用いることができる．

4.6.2　管理図が発した警告への留意点

原因究明時には，性能異常検出時に管理図が発した警告に対しても注意を払う必要がある．そうすることで，原因究明にかかる時間を短縮したり，優先的に原因究明を進めるべきリクエストの種類を見分けたりすることができる．

まず，性能異常を検出した統計値の種類に着目する必要がある．中央値や最小値が正常にも関わらず最大値を監視する管理図のみが警告を発した場合，コネクションなど，資源の不足が性能異常を引き起こしている可能性が高い．最大値のみが異常な値となっている場合，少数のリクエストのみ処理に異常に大きな時間がかかっているということになる．この場合，資源が不足し一部のリクエストのみが待ち行列に入れられている可能性が高い．資源不足が発生し始めるとき，多数のリクエストは待ち行列に入れられることなく，処理が通常通り行われる．一方，偶然多くのリクエストがサーバに到達した場合のみ，処理能力を超えてしまった少数のリクエストが待ち行列に入れられ，処理時間が異常に大きくなる．

最小値を監視する管理図が警告を発した場合も注意が必要である．最小値が異常な値になった場合，全てのリクエストの処理に影響を及ぼす性能異常が発生している可能性が高い．資源不足による性能異常が発生した場合，ある程度まとまった数のリクエストの内，全てのリクエストの処理が滞ることはほとんどない．し

たがって，全てのリクエストの処理時間が増大するわけではなく，最小値の分布に変化は生じない．アプリケーションサーバ内でのデータキャッシュの保持を取りやめるなど，全てのリクエストの処理に影響を及ぼす要因が性能異常の原因だと考えられる．

性能異常を検出したリクエストの種類が複数のクラスタに分類され，複数の原因により性能異常が同時に発生していることが分かった場合，以下に述べることを考慮に入れ，原因究明時の優先順位を定める必要がある．まず，性能異常検出時に管理図が多数の警告を発したリクエストの種類について優先的に原因究明を始める必要がある．警告が多数発生しているということは，性能異常が発生している期間が長いということになる．よって，被害が大きくなる可能性が高いため，優先して回復する必要がある．

また，警告がより早く発っせられたリクエストの種類から優先的に原因究明を行う必要がある．伝播して発生している複数の性能異常の内，大元の原因を見つけ出すことができるからである．性能異常は伝播して発生することがあり，大元の異常を回復することで複数の異常を同時に回復することができる場合がある．例えば，アプリケーションサーバとデータベースサーバの間のコネクションが不足した場合，アプリケーションサーバに処理が滞ったリクエストが滞積していくことがある．その結果，ウェブサーバとアプリケーションサーバの間のコネクションも不足するといった事態に陥ることがある．

4.7 まとめ

本章では，提案手法の要素技術の一つである性能異常分類手法の説明を行った．まず性能異常を分類する目的として，同時に複数発生した性能異常の原因が同じか異なるかを判定し，その後の原因究明を支援することを説明した．次に概要では，クラスタリングを処理時間の変化傾向に着目して行うことと，分類手法が三つのステップで構成されていることを説明した．

第 1 ステップでは，累積分布関数のグラフから，処理時間の分布の変化傾向をシグネチャとして抽出する．第 2 ステップでは，二つのシグネチャ同士を重ね合わせたときに重なる面積を工夫して計算し，類似度を得る．最後に第 3 ステップでは，既存のクラスタリング手法を第 2 ステップで得た類似度の表に適用し，クラスタリングを行う．本章の最後では，提案手法の出力結果を利用して原因究明を行う際の留意点を述べた．

第5章　ケーススタディ

提案手法の有用性を示すために，提案手法を用いて性能異常の早期検出と原因究明支援を行うことができたケーススタディを報告する．本章では，まず実験手順を説明する．その後，本実験で利用したウェブアプリケーションやサーバソフトウェアなど実験環境について説明を行う．その上で，ケーススタディを三つ報告する．さらに，ケーススタディを踏まえて，性能異常分類手法に関する議論を行う．

5.1　実験手順

本実験は次の五つの手順からなる．

1. 性能異常を発生させる．
2. 性能異常検出手法を用いて性能異常を検出する．
3. 性能異常分類手法を用いて性能異常を原因ごとに分類する．
4. 提案手法が出力した結果を利用しながら，人手により詳しい原因究明を行う．
5. システム修復後，性能異常から回復していることを確認する．

5.1.1　性能異常の発生

まず 1 番目のステップで，実験を行うために，性能異常を発生させる必要がある．本論文では，クライアント数を増加させたり，システムの設定を変更したりしながら，性能異常が発生する環境を構築する．現在，ウェブアプリケーションを提供する環境は非常に複雑であり，さまざまな要素が絡み合って性能異常は発生する．

ケーススタディで行ったシステム設定の変更は，コネクションの最大同時接続数変更やデータベース内のテーブルへのインデックス追加などである．コネクションの最大同時接続数は大きく設定すれば良いというものではなく，個々のシステムの特徴に合わせて調整する必要がある．小さく設定しすぎれば，計算資源が余っているにも関わらずリクエストを受け付けることができない．逆に大きく設定しすぎれば，マシンの能力を超える数のリクエストを同時に処理しようと試み，過負荷でマシンの停止を引き起こす可能性がある．また，テーブルへのインデックス追加を行うと，検索時間の短縮を行えることがある．しかしその一方で，データの更新などにかかる時間は増大する．このため，テーブルの内容と発行する SQL 文に応じて適切なインデックスを作成する必要がある．

ここで注意してほしいのは，本論文ではフォールトインジェクションを行っているわけではないということである．性能異常を発生させる一つの手段としてフォールトインジェクションがある．フォールトインジェクションとは，あらかじめ性能異常の原因をシステムへ挿入し，性能異常を発生させる手段である．そして，提案手法を用いて挿入したフォールトを見つけ出し，提案手法の有用性を示す．

これに対し，本論文では，未知の原因を，提案手法による支援のもと特定できたケーススタディを報告する．本論文で行うクライアント数の増加やシステム設定の変更は，性能異常の原因そのものを挿入しているわけではない．これらを変化させることによって，隠れていた性能異常の原因を顕在化させ，性能異常を発生させている．実際に，第 5.3 節，第 5.4 節，第 5.5 節で行っている設定変更は，それ以前の実験で発見した，性能を**向上**させるための変更である．

5.1.2 性能異常の検出

次に 2 番目のステップで，提案手法の要素技術の一つである性能異常検出手法を用いて性能異常を検出する．性能異常を検出したときの処理時間の変化がどの程度であるかを提示し，提案手法で早期検出が行えているかを判断する．本実験では，統計値として平均値，最大値，中央値，最小値の 4 種類を選択した．それぞれの統計値の計算は 10 分ごとに行った．

5.1.3　性能異常の分類

性能異常検出後，3 番目のステップでもう一つの要素技術である性能異常分類手法を用いて，リクエストの種類を性能異常の原因ごとに分類する．本論文では，性能異常を検出した統計値の種類に応じて，類似度計算時の正規化方法を選択する．最大値に異常が発生した際は大域的な正規化，それ以外の統計値（平均値，中央値，最小値）に異常が発生した際は局所的な正規化を用いる．

第 4.4.3 節で述べたとおり，局所的な正規化を用いることで自然のゆらぎの影響を無視できる．そのため，通常はこちらを用いる．それに対し，最大値に性能異常が発生した場合は，自然のゆらぎを超える大きなスパイクを生み出すため，大域的な正規化を用い，スパイクに注目して類似度を計算する．また，第 4.5.1 節で述べたように，クラスタリングの閾値は全てのケーススタディを通じて 60 に定めている．

5.1.4　原因究明

性能異常分類後，4 番目のステップで，分類結果を利用しながら人手で原因究明を行う．本論文では，システマティックに原因の特定を行う手法は提案しないが，第 4.6 節で述べたことに留意しながら原因究明を行うことができる．

提案手法が保証するのは 2 点である．1 点目は，同じクラスタに含まれるリクエストの種類には同じ原因により性能異常が発生しているということである．そして 2 点目は，逆に，異なるクラスタに含まれるリクエストの種類には異なる原因により性能異常が発生しているということである．原因究明の基本的な方針としては，個々のクラスタに含まれるリクエストの種類同士のみで共通して利用しているコンポーネントを探すということが挙げられる．そして，それを全てのクラスタに対して行う．

5.1.5　システム修復

最後に 5 番目のステップで，提案手法の検出，原因究明結果が正しいことを，システムを性能異常から回復させることで示す．そのために，4 番目のステップで得た原因究明結果を実際にシステムに適用して再度実験を行う．実際に性能異常が回復され異常が検出されなければ，提案手法が有用だということがわかる．

表 5.1: 実験で用いたマシンのスペック

CPU	3.00 GHz × 4
主記憶	2 GB
ネットワーク	Gigabit Ethernet

5.2 実験環境

5.2.1 概要

実験では，対象のウェブアプリケーションとして RUBiS [72] を用いた．RUBiS はオークションサイト eBay.com [79] を模して作成されたベンチマークウェブアプリケーションである．実際にさまざまな研究で，各研究の提案手法の有用性を示すために利用されている．

この RUBiS は，さまざまな環境に対してプログラムが用意されている．本実験では Java Enterprise Edition を用いて構築されているものを選択した．さらに，永続的なデータの管理を CMP（Container Managed Persistence）を用いて行うものを選択した．

本論文では，実験環境を一般的な 3 層構造で構築した．そして，ウェブ層に Apache HTTP サーバ [12] 2.2.11，アプリケーション層に JBoss アプリケーションサーバ [13] 5.0.1，データベース層に MySQL [80] 5.1.34 をそれぞれインストールした．ウェブサーバ，アプリケーションサーバ，データベースサーバはそれぞれ別のマシンで動作させ，各層 1 台ずつマシンを用意した．さらに，クライアントエミュレータを動作させるために，もう 1 台マシンを用意した．クライアントエミュレータではサーバへリクエストを送信する．このため，マシンは合計で 4 台になる．全てのマシンで動作しているオペレーティングシステムは Linux であり，カーネルのバージョンは 2.6.27 である．マシンのスペックは表 5.1 に示す．

RUBiS にはサーバへリクエストを送信するためのクライアントエミュレータが付属している．個々のクライアントは個別のスレッドとして動作し，HTTP リクエストを送信する．全てのクライアントはまずトップページを訪れ，そこから各ページへと遷移を行いながら商品をオークションに出品したり，出品中の商品へ入札したりといった人間のユーザを模した処理を行う．ページ間の遷移を行う際にも人間のユーザを模しており，あるページから他のページへの遷移は定めた確率に

従ってランダムに行われる．さらに，ページ間の遷移を行った後は，人間のユーザが考慮する時間を模して遅延が挿入されており，こちらも定めた分布に従ってランダムに遅延を挿入することができる．また，一連の遷移が終了した場合には，もう一度トップページからの遷移を再開する．さらに，このクライアントエミュレータを用いてデータベース内のデータを初期化することができる．本実験では登録しているユーザの数，商品の数をそれぞれ 100,000 人，32,667 個に設定した．

5.2.2　クライアントエミュレータの変更

本論文のケーススタディでは，このクライアントエミュレータにいくつかの変更を加えた．これらの変更は，クライアントスレッドの動作をより人間に近づけるためのものである．一つは，クライアントスレッドに 8 秒間のタイムアウトを設けたことである．始めの実装では，クライアントスレッドにタイムアウトが設定されていなかったため，サーバからの応答を，コネクションが切断されるまで待ち続ける実装になっていた．しかし，実際の人間は，ある時間経過して応答がなければ再実行を繰り返すのが通常である．その動作をエミュレータへ実装した．実装には Apache Commons の HTTP Client コンポーネント [81] を利用した．タイムアウトを 8 秒に設定した理由は，通常人間がウェブアプリケーションの応答を待つ時間は 8 秒と言われているからである．他の研究 [62] でもこの値を研究の有用性を示すための閾値として利用している．本論文では性能異常を発生させるために故意にサーバの高負荷を引き起こしたため，このような変更が必要になった．

もう一つは，クライアントスレッドの起動間隔を 5 秒設けるように実装を変更した．当初の実装では，クライアントスレッドを一斉に起動させ，ウェブアプリケーションへのリクエストを一斉に送信し始めるようになっていた．この実装では，大量のリクエストが同時にサーバへ送信され，一時的にシステムが過負荷状態に陥ってしまう．さらに悪いことに，この一時的な過負荷は他の異常も引き起こし，実験を通じてその影響が残ってしまうことがあった．通常のウェブアプリケーションに対しては，このような状況はほとんど起こらないため，クライアントスレッドの起動間隔を設けることでこの問題を回避した．スレッド起動時は一定間隔でサーバへリクエストが送信されることになるが，時間が経過すればリクエストがサーバへ送信される間隔はランダムになり，実環境と近くなる．起動後，ページ間の遷移時間がランダムに決定されるためである．

5.2.3　リクエスト処理時間の計測

Apache では各リクエストの URL と処理時間をログへ記録する機能を備えているため，本実験ではシステムへの変更をせずに提案手法を適用することができる．必要なのは，ログに記録された URL を用いてリクエストを種類ごとに分類するプログラムのみである．これは提案手法がシステムへの導入を容易に行えることを表している．

ただし，今回の実験ではクライアントエミュレータ側でリクエストの種類ごとに処理時間を計測し，ファイルへ書き出している．これは，RUBiS のクライアントエミュレータが処理時間を計測する機能を備えていたため，こちらの機能を用いた方がリクエストを種類ごとに分類するのが容易だったためである．

今回の実験ではクライアント側で取得した応答時間を，サーバ側で計測した処理時間と同等に扱ったとしても問題はない．通常，クライアント側で測定した応答時間はネットワークによる遅延を含んでいるため，サーバ側で計測した処理時間との差が大きく，しかもその差は一定ではないという問題がある．しかし，今回の実験ではクライアントエミュレータとサーバが動作しているマシン全てを同一の LAN（Local Area Network）内に配置し行ったため，ネットワークによる遅延は小さく，しかもその揺らぎも小さい．

5.2.4　リクエストの種類

RUBiS ではプログラムの開発者によってリクエストの種類は 27 種類に定められている．表 5.2 にその一覧を示す．表 5.2 にあるように，リクエストは種類ごとに処理が及ぶサーバが異なる．全てのリクエストはまずウェブサーバに送信され，必要があればアプリケーションサーバに転送，さらに必要があればデータベースサーバへ転送される．また，アプリケーションサーバ内でデータベース内のデータをキャッシュとして保持するために，Entity Bean というソフトウェアコンポーネントを利用しており，表 5.2 に示したように，それぞれのリクエストの種類が必要な Entity Bean へアクセスを行う．Entity Bean 以外にも，各リクエストはさまざまなソフトウェアコンポーネントにアクセスを行う．ウェブページを動的に生成するための Servlet やタスクの処理を行うための Session Bean がこれに当たる．このように，ウェブアプリケーションはさまざまなコンポーネントを複雑に呼び出して処理が行われていることが分かる．

表 5.2: RUBiS における 27 のリクエストの種類

リクエストの種類	処理が及ぶサーバ	利用するソフトウェアキャッシュ
AboutMe (auth form)	ウェブ	
Browse	ウェブ	
Home	ウェブ	
Register	ウェブ	
Sell	ウェブ	
BuyNowAuth	アプリケーション	
PutBidAuth	アプリケーション	
PutCommentAuth	アプリケーション	
SellItemForm	アプリケーション	
AboutMe	データベース	BidBean, BuyNowBean, CommentBean, ItemBean, UserBean
BrowseCategories	データベース	CategoryBean
BrowseCategoriesInRegion	データベース	CategoryBean, RegionBean
BrowseRegions	データベース	RegionBean
BuyNow	データベース	ItemBean, UserBean
PutBid	データベース	BidBean, ItemBean, UserBean
PutComment	データベース	ItemBean, UserBean
RegisterItem	データベース	IDManagerBean, ItemBean
RegisterUser	データベース	IDManagerBean, RegionBean, UserBean
SearchItemsInCategory	データベース	ItemBean
SearchItemsInRegion	データベース	ItemBean
SelectCategoryToSellItem	データベース	CategoryBean, UserBean
StoreBid	データベース	BidBean, IDManagerBean, ItemBean
StoreBuyNow	データベース	BuyNowBean, IDManagerBean, ItenBean
StoreComment	データベース	CommentBean, IDManagerBean, UserBean
ViewBidHistory	データベース	BidBean, ItemBean, UserBean
ViewItem	データベース	BidBean, ItemBean, UserBean
ViewUserInfo	データベース	CommentBean, UserBean

5.3 ケース 1

ケース 1 では，提案手法が性能異常を早期に検出し，さらに異常が発生しているサーバの特定に役立ったケースを示す.

5.3.1 性能異常の発生

まずサーバの設定をデフォルトにしたまま，試運用を模してクライアント数 200 の負荷をサーバにかけた．そのときに測定したリクエスト処理時間を用いて管理図の基準線を計算した．その後，実運用を模してクライアント数を 200 から 300 へ増加させながらサーバへ負荷をかける．そのときに測定した処理時間を用いて管理図へ打点していく．実験時間は，クライアント数が 200 と 300 の期間をともに 3 時間 20 分に指定した．つまり，各期間で四つずつ管理図への打点が行われることになる．統計値の計算を 10 分ごとに行い，さらに五つの統計値ごとに中央値を計算して打点していくので，$10 \times 5 \times 4 = 200$ 分である．

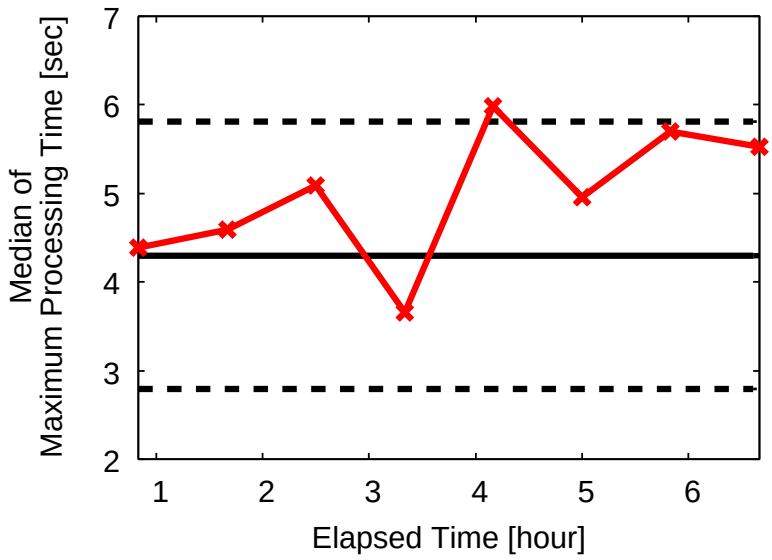
5.3.2 性能異常の検出 (1)

クライアント数が 300 に増加した時点で 27 のリクエストの種類の内，26 のリクエストの種類で性能異常を検出した．警告を発したのは最大値を監視している管理図であった．このときに観測した管理図の例が図 5.1 である．1〜4 番目の打点がクライアント数が 200 のときのもの，5〜8 番目の打点がクライアント数が 300 のときのものである．図 5.1(a) と図 5.1(b) ともに，上方管理限界線を超えて打点が行われているのがわかる．このとき，リクエスト処理時間の平均値の増大は 30〜110 ミリ秒であり，性能異常を早期に検出できたといえる．

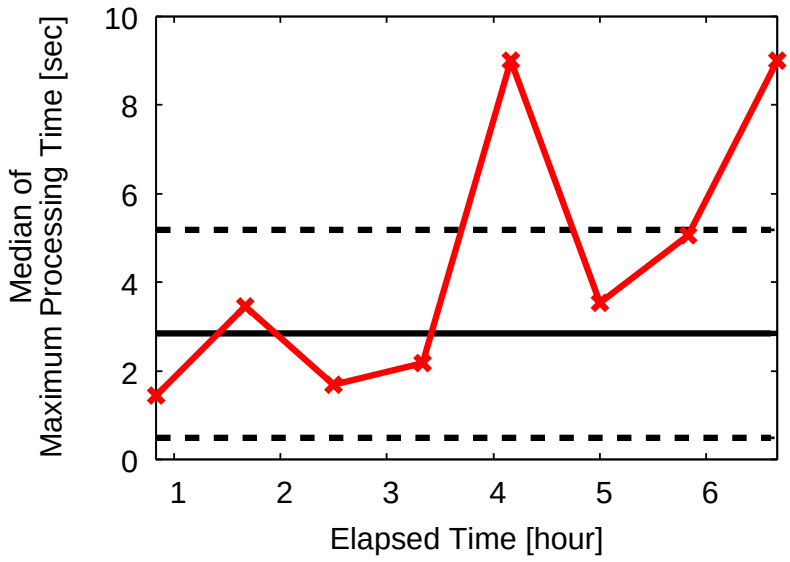
5.3.3 性能異常の分類 (1)

性能異常検出後，提案手法を適用しリクエストの種類の分類を行った．最大値を監視する管理図が警告を発したため，前述の通り大域的な正規化を用いてシグネチャ同士の類似度を計算する．

提案手法を適用した結果，リクエストの種類は C_a と C_b の 2 種類のクラスタに分類された．C_a は 5 つのリクエストの種類を含み（[`Home`, `Browse`, `Register`, `Sell`, `AboutMe (auth form)`]），C_b は残り 22 のリクエストの種類を含む（[`RegisterUser`, `AboutMe`, `SelectCategoryToSellItem`, ...]）．C_a と C_b それぞれに含まれるリクエストの種類のシグネチャを重ねた例を図 5.2 と図 5.3 にそれぞれ示す．図 5.2 は `Home` で作成した緑のシグネチャと `Browse` で作成した紫のシグネチャを重ね合わせており，図 5.3 は `RegisterUser` で作成した緑のシ



(a) リクエストの種類：`Home`



(b) リクエストの種類：`RegisterItem`

図 5.1: ケース 1 においてサーバ設定がデフォルトのときに警告を発した管理図の例

グネチャと `AboutMe` で作成した紫のシグネチャを重ね合わせている．ともに，黒い部分が重なった領域である．C_a に含まれたリクエストの種類同士，C_b に含まれたリクエストの種類同士のシグネチャは重なっていることを表す黒い部分が，シグネチャの大部分を占めており，類似度が高いことが見て取れる．一方，C_b に含

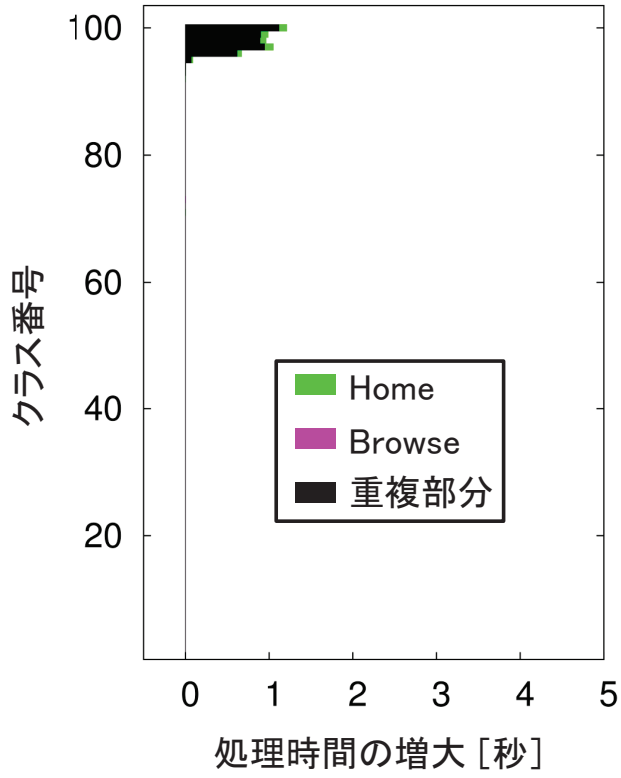


図 5.2: C_a に含まれるリクエストの種類のシグネチャを重ねた例

まれるリクエストの種類のシグネチャと C_a に含まれるリクエストの種類のシグネチャを比べると，C_a に比べて C_b の方が 100 番目のクラスの値が大きくなっているのが分かる．

5.3.4 原因究明 (1)

分類結果を利用して性能異常発生箇所の特定を行った．RUBiS のソースコードを調査したところ，クラスタ C_a と C_b に含まれるリクエストの種類では処理が及ぶサーバが異なることがわかった．図 5.4 に示すように，C_a に含まれた 5 種類のリクエストは処理がウェブサーバのみで行われるのに対し，C_b に含まれた 22 種類のリクエストは処理にアプリケーションサーバを必要とするものだった．

このことから，性能異常はウェブサーバとアプリケーションサーバの両方で発生している可能性が高い．まず，C_a に含まれた 5 種類のリクエストに性能異常が発生したことから，ウェブサーバに性能異常が発生していると考えられる．C_b に含まれたリクエストの種類もウェブサーバに発生している性能異常に影響を受けて

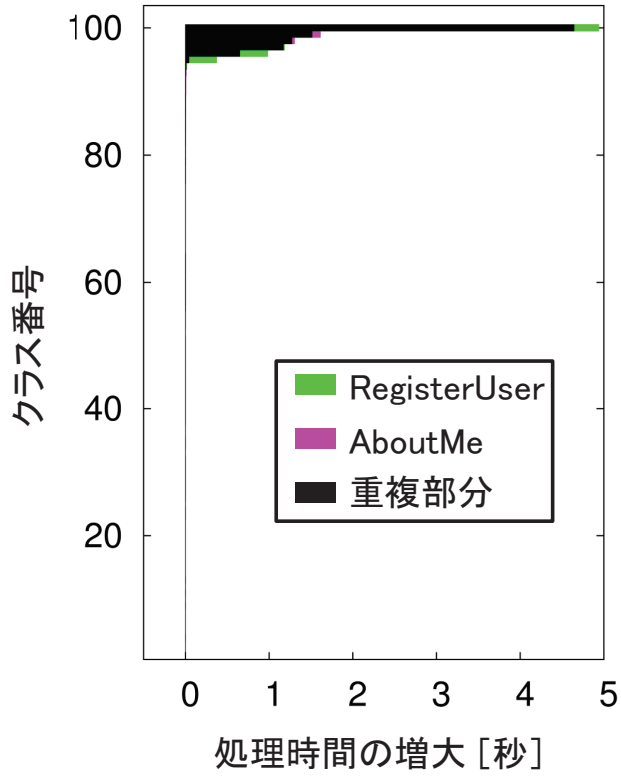


図 5.3: C_b に含まれるリクエストの種類のシグネチャを重ねた例

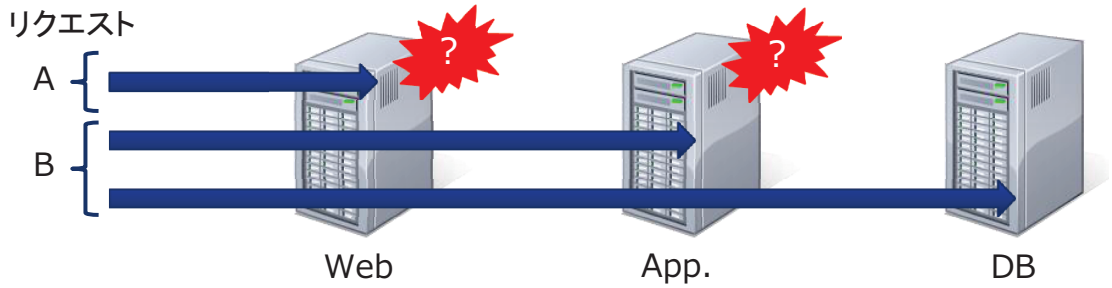


図 5.4: ケース 1 における原因の予想

いると考えられるが，C_a とは異なるクラスタに分類されたため，影響を受けている性能異常はそれだけではないと考えられる．ここで，処理がデータベースサーバに及ばないリクエストの種類も C_b に分類されたことから，もう一つの性能異常はアプリケーションサーバに発生していると考えられる．

5.3.5　システム修復 (1)

まずアプリケーションサーバを調査したところ，ウェブサーバとアプリケーションサーバの間のコネクション数が上限に達していることがログの出力から分かった．これを受け，JBoss の設定である maxThreads をデフォルトの 200 から 250 に増加させ，もう一度実験を行った．maxThreads は JBoss が Apache から転送されてくるリクエストを受け付ける際に生成するスレッド数の上限を定めるための設定である．すると，11 のリクエストの種類で管理図により警告が発せられた．このときに観測した管理図が図 5.5 である．1〜4 番目の打点がクライアント数が 200 のときのもの，5〜8 番目の打点がクライアント数が 300 のときのものである．

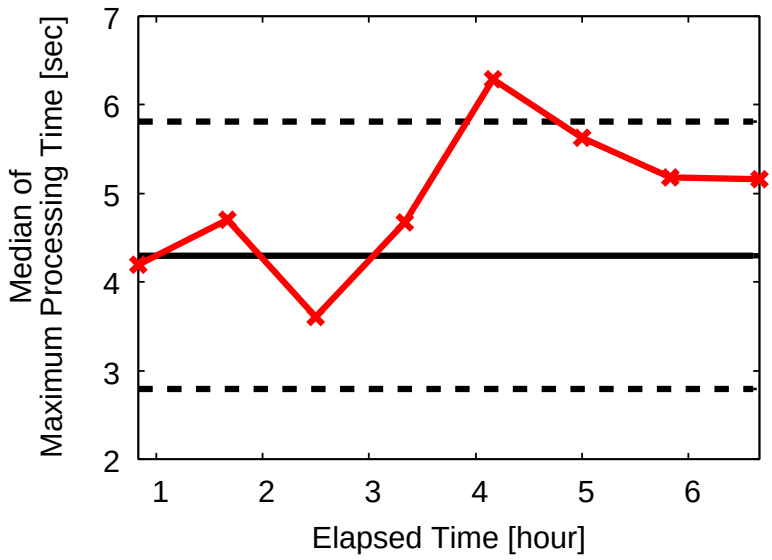
5.3.6　性能異常の検出 (2)

この実験では，先ほどに比べて処理時間の増大は小さくなったものの，まだ警告が発せられている．図 5.5(a) と図 5.5(b) で共に上方管理限界線を超えた打点が見て取れる．これはアプリケーションサーバにおける性能異常は解消したものの，ウェブサーバにおける性能異常がまだ残っているためだと考えられる．

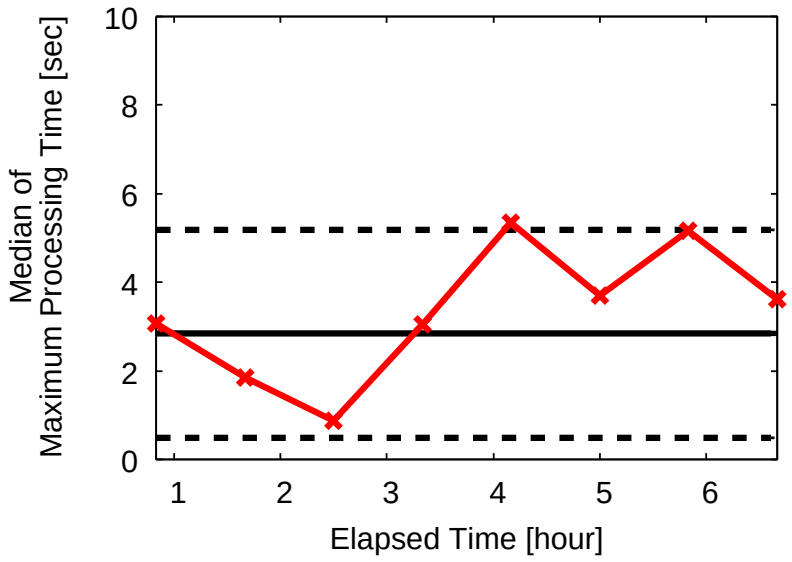
maxThreads を 250 より大きな値に設定し再実験を行っても，性能異常の発生を防ぐことはできなかった．クライアント数が 300 の場合でも，必要なスレッド数が 250 を上回らないということがわかる．maxThreads を必要以上に大きな値に設定しスレッドの上限数を増やしても，生成されるスレッド数に変化がないため，処理時間を短縮することはできない．さらに，処理能力を上回るほど大きな値に設定してしまうと過負荷時にサーバが停止してしまう．このため，maxThreads は最低限の値に定める必要がある．

5.3.7　性能異常の分類 (2)

性能異常を検出したため，もう一度提案手法を用いて性能異常を分類した．すると，今度は全てのリクエストの種類が一つのクラスタに分類された．第 5.1 節で述べたように，性能異常を検出しなかった 16 のリクエストの種類も含めてクラスタリングを行った．図 5.4 を用いて説明したように，こちらはウェブサーバに発生している性能異常であると考えられる．



(a) リクエストの種類：`Home`



(b) リクエストの種類：`RegisterItem`

図 5.5: ケース 1 において `maxThreads` を 250 に増加させたときに警告を発した管理図の例

5.3.8 原因究明 (2)

分類結果を受けてウェブサーバを調査した．最大値が増大して性能異常が発生している場合，その原因として軽微なコネクション不足が考えられる．軽微なコ

ネクション不足とは，コネクションが不足していたとしても常に不足しているわけではなく，偶然多数のリクエストを受信したときだけ利用可能なコネクションを確保できない状況である．この場合，一部のリクエストの処理時間だけ大幅に増大し，最大値に性能異常が現れる．ほとんどのリクエストに対してはコネクションの確保がスムーズに行えるので，処理時間の増大はない．しかし，偶然多数の他のリクエストと競合してしまった場合のみ，コネクションの確保に時間がかかり，処理時間が大きく増大する．

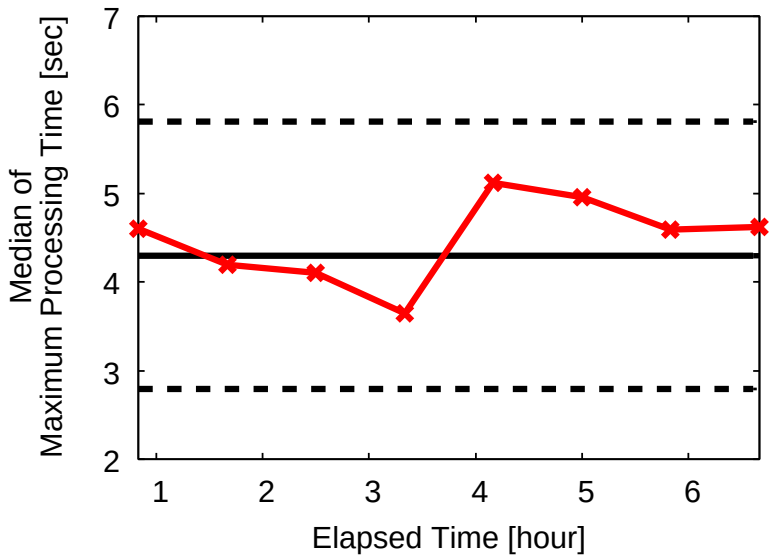
5.3.9 システム修復 (2)

上記のことを踏まえて，コネクションに関する設定をいくつか変更しながら実験を繰り返したところ，`KeepAliveTimeout` をデフォルトの 5 秒から 2 秒に減少させることで性能異常から回復できた．このときの管理図が図 5.6 である．1〜4 番目の打点がクライアント数が 200 のときのもの，5〜8 番目の打点がクライアント数が 300 のときのものである．全ての打点が 2 本の管理限界線の内側にされている．

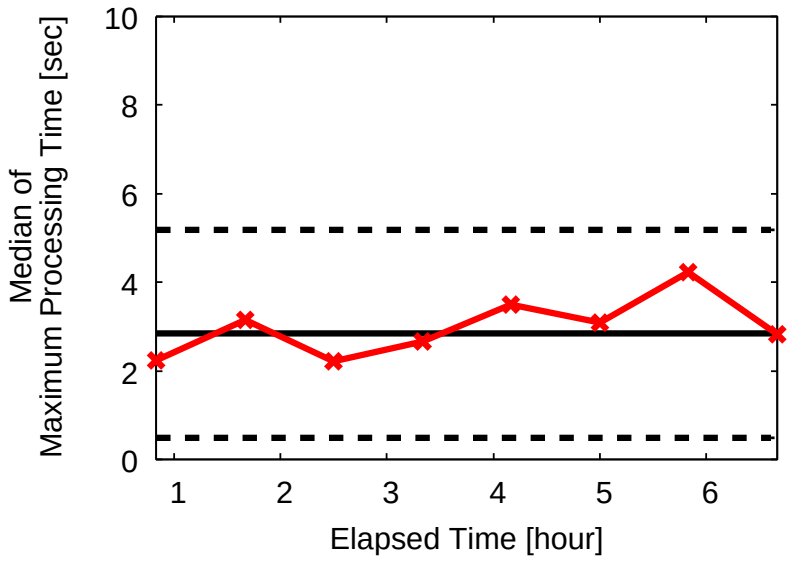
`KeepAliveTimeout` はサーバがレスポンス送信後，コネクションを維持する時間を設定するパラメータである．コネクションを維持することで，次に同じクライアントからリクエストが送信されてきた際にコネクションを確立するための時間が削減でき，処理時間の短縮につながる．しかし，コネクションが不足している場合に無駄にコネクションを維持してしまうことで，他のコネクションの確立を妨げてしまうことがある．そのため，無駄なコネクションは切断する必要がある．この設定はクライアント数やウェブアプリケーションの性質，クライアントとサーバの位置関係を考慮に入れて適切に設定する必要がある．今回の実験環境においては，クライアント数が増加した場合はタイムアウトを短く設定する方が性能を向上できることが分かった．このように，提案手法を用いて性能異常を分類することで発生箇所特定を速やかに行うことができた．

5.4 ケース 2

ケース 2 では，提案手法が異常を引き起こしたメソッドの特定に役立ったケースを示す．



(a) リクエストの種類：Home



(b) リクエストの種類：RegisterItem

図 5.6: ケース 1 においてさらに KeepAliveTimeout を 2 秒に減少させたときの管理図の例

5.4.1 性能異常の発生

本ケーススタディでは KeepAliveTimeout を 1，maxThreads を 400 にそれぞれ設定している．これらの設定変更は第 5.3 節で述べたように，RUBiS の性

能を向上するものである．第 5.3 節の原因究明結果から，本論文の実験環境では KeepAliveTimeout は小さく設定し直した方が良く，maxThreads は大きく設定し直した方が良いという結果が得られた．さらに，データベース内の items テーブルと users テーブルにそれぞれ新たにインデックスを加えた．これらもさまざまな環境を試す上で発見した性能向上のための変更である．本節で述べるように，発生した性能異常の原因はこれらの設定変更によるものではない．

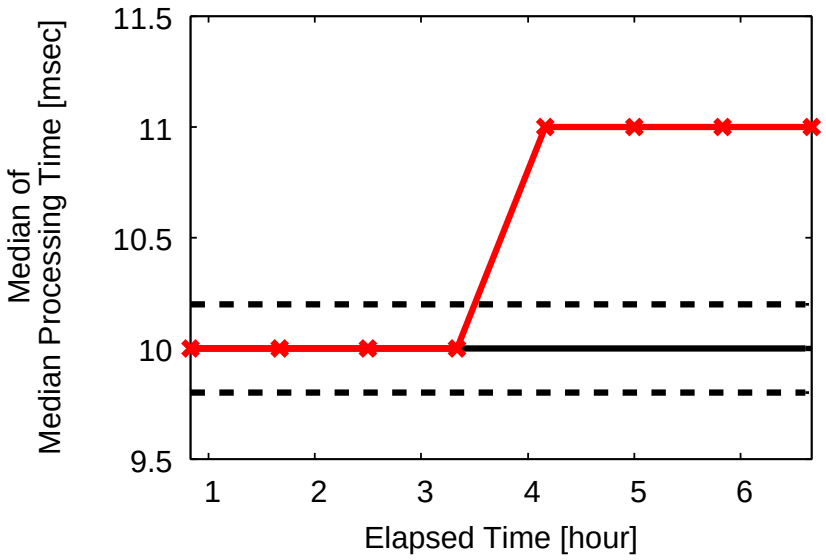
5.4.2 性能異常の検出

先の設定変更後，実験を行った．まず，試運用時を模してクライアント数が 200 の負荷をサーバへかけ，その時に計測した処理時間を用いて管理図の基準線を計算する．さらに実運用時を模してクライアント数を 200 から 300 に増加させながら計測した処理時間を用いて管理図に打点を行った．そうしたところ，5 種類のリクエストの種類（SearchItemsInCateogry，SearchItemsInRegion，ViewItem，ViewUserInfo，ViewBidHistory）の処理時間の中央値を監視する管理図が警告を発した．このときの管理図の例が図 5.7 である．1〜4 番目の打点がクライアント数が 200 のときのもの，5〜8 番目の打点がクライアント数が 300 のときのものである．クライアント数が増加した五つ目以降の打点に，上方管理限界線を超えるものが見受けられる．このときリクエスト処理時間の変化は 4〜10 ミリ秒であったため，性能異常を早期に検出できたということができる．

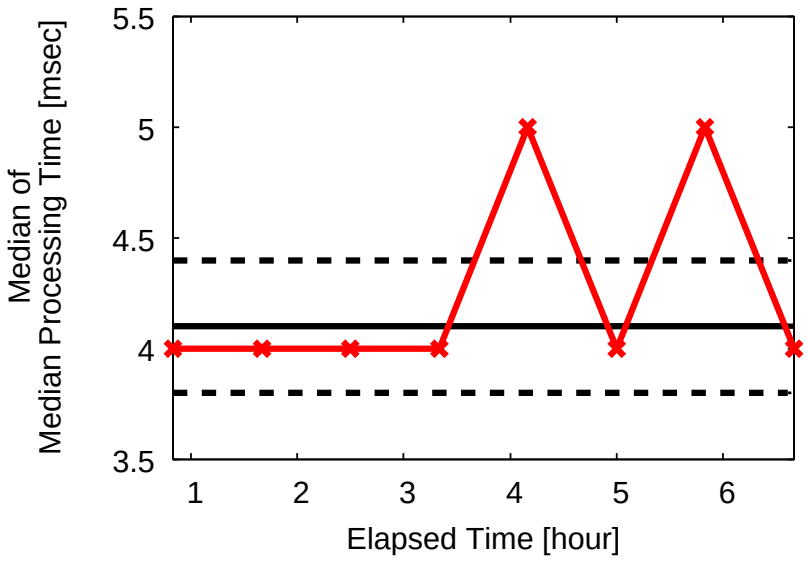
5.4.3 性能異常の分類

次に，原因究明を行うために性能異常の分類を行った．ケース 2 では中央値を監視する管理図が警告を発したため，局所的な正規化を用いて計算したシグネチャの類似度にクラスタリングを適用した．そうしたところ，異常を発した五つのリクエストの種類はそれぞれ別々のクラスタに分類された．それぞれのクラスタは［SearchItemsInCateogry］,［SearchItemsInRegion］,［ViewItem，PutBid］,［ViewUserInfo］,［ViewBidHistory］である．ここで注目してほしいのがクラスタ［ViewItem，PutBid］である．この二つのリクエストの種類のシグネチャを重ね合わせた図はすでに図 4.7 で示している．

PutBid では警告が発生しなかったにも関わらず，ViewItem と同じクラスタに分類された．このときの PutBid の管理図が図 5.8 である．処理時間の増大は



(a) リクエストの種類：`ViewItem`



(b) リクエストの種類：`ViewUserInfo`

図 5.7: ケース 2 において警告を発した管理図の例

見られるものの，管理限界線を超える打点は見られない．第 4.5.3 節で述べたように，提案手法で性能異常を検出しなかったとしても，性能異常分類手法がそれを補うことができる場合もある．

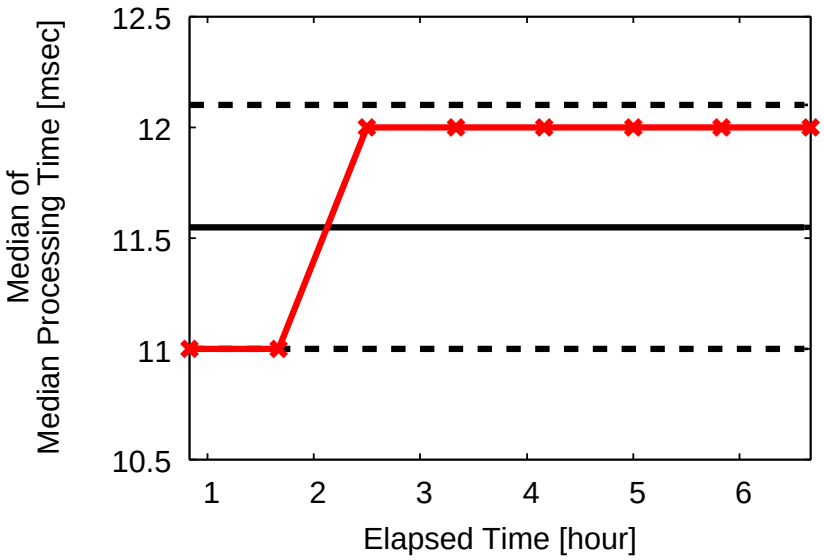


図 5.8: ケース 2 において原因究明前の PutBid の管理図

5.4.4 原因究明

ここでは，このクラスタに分類された二つのリクエストの種類に発生している性能異常の原因が同じであることを示す．そのために，分類結果に注目しながら性能異常の発生箇所を特定した．今回のケースでは，クラスタに含まれた 2 種類のリクエストのみに使用されるコンポーネントを探した．そのため，多数のリクエストの種類において警告が発せられたケース 1 よりも細かい粒度で調査した．具体的には，リクエストの入り口である Servlet からプログラムを少しずつ調査し，共通の箇所を探した．すると，これら 2 種類のリクエストのみで共通して使用される SB_ViewItemBean.getItemDescription というメソッドを発見した．このメソッドはリクエストされた商品の詳細情報を取得するメソッドである．詳細情報とは，その商品の個数や出品者，現在の入札数などである．

さらに getItemDescription メソッドを詳しく調査したところ，このメソッドはリクエストされた商品の在庫数に応じて 2 種類のパスのうち片方を通過することが分かった．このメソッドは商品に対する入札の最低金額を表示し，その計算方法は在庫数が 1 かそれより大きいかによって異なる．在庫数が 1 の時には現時点で最後に入札された金額より高ければ入札が可能になる．一方，在庫数が n の時には n 番目に高い入札額より高ければ入札が可能になる．後者の方が複雑な処理を要するため処理時間は大きくなる傾向がある．

この特性に注目して原因究明を行った．本ケースで処理時間の中央値が増大し

た原因は，在庫が複数個の商品がリクエストされる割合が徐々に大きくなったことであった．クライアントエミュレータの設定を調査したところ，在庫が複数個の商品に対して`getItemDescription`メソッドがリクエストされるべき割合は本来常に 20%であるべきだった．よって，処理時間も本来は常に一定となるはずである．しかし原因究明を進めたところ，RUBiS に混入していた二つのバグにより，本実験ではこの割合が約 10%から始まり徐々に 20%へと近づくように変化していたことが分かった．クライアント数の増加によりこの変化が加速され，管理図が処理時間の異常を検出したと考えられる．

まず，データベースの初期化時に，在庫が複数個の商品に対して`getItemDescription`メソッドがリクエストされるべき割合が約 10%であった原因を述べる．この原因は RUBiS の二つのバグによるものだった．まず，一つ目のバグにより，初期化したデータベース内に存在する商品の内，在庫が複数個の商品の割合が実際は 20%より小さかった．RUBiS ではデータベースを初期化する際に，登録する商品の内，在庫が複数個の商品の割合を設定することができる．デフォルトの設定ではこれが 20%に設定されていた．ただし，商品の在庫に関して設定できるのは在庫が複数個の商品の割合のみである．よって，その商品の在庫数を個別に設定することはできず，クライアントエミュレータが 10 以下の値を無作為に設定する．ここで，無作為に設定する文にバグが混入していた．在庫が複数個の商品の場合，在庫数は 2〜10 に設定されるべきである．しかし，バグにより 1〜10 から無作為に個数を設定してしまっていた．結果として，在庫数が複数個の商品の割合が設定した値よりも小さくなっていたのである．理論上，在庫が複数個の商品の割合は約 18%であったと考えられる．

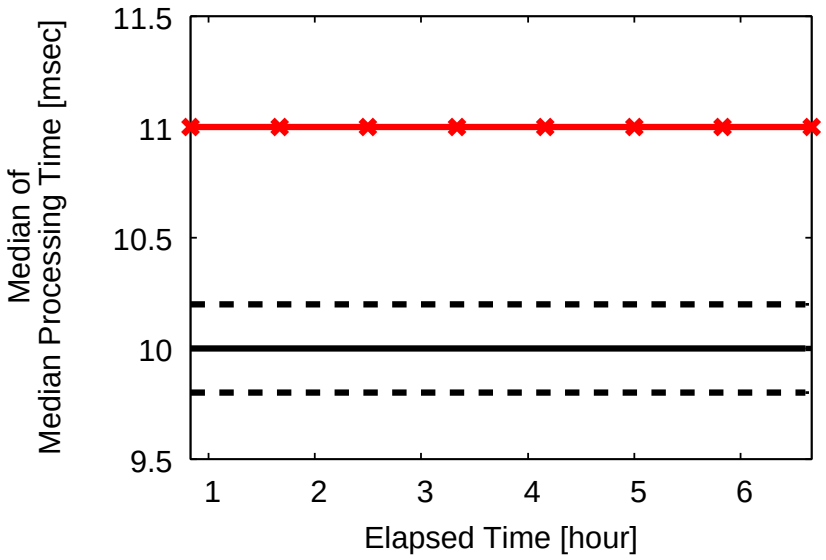
また，もう一つのバグにより，`getItemDescription`メソッドが在庫が複数個の商品に対してリクエストされる割合がさらに小さくなっていた．このバグは在庫数が複数個の商品を各カテゴリに均等に配分しなかったことによるものである．RUBiS では，商品を書籍や家電などのようにカテゴリごとに検索することが可能である．本来，在庫が複数個の商品は各カテゴリに均等に振り分けられるべきである．しかし，バグにより，データベースへ商品を登録する際，在庫が複数個の商品は少数のカテゴリのみに登録されていた．RUBiS ではデータベース初期化として商品を登録する際，その商品のカテゴリは設定ファイルに記述してあるカテゴリからラウンドロビンで選択していた．さらに，その設定ファイルには各カテゴリに所属すべき商品の数も記述してある．したがって，商品数の少ないものから順に登録が終了し，後半には商品数が多いカテゴリのみに商品が登録され

ることになる．一方，在庫数の設定は，先に登録する商品の在庫数を 1 に，残りの商品の在庫数を複数に設定していた．このため，在庫が一つの商品は全てのカテゴリに登録されるのに比べ，在庫が複数個の商品は一部のカテゴリにしか登録されないということが起こる．しかし，`getItemDescription` メソッドは全てのカテゴリに均等にリクエストされるため，在庫が複数個の商品に対してリクエストされる割合が，データベース内に存在している割合である約 18%よりさらに小さい約 10%になっていた．

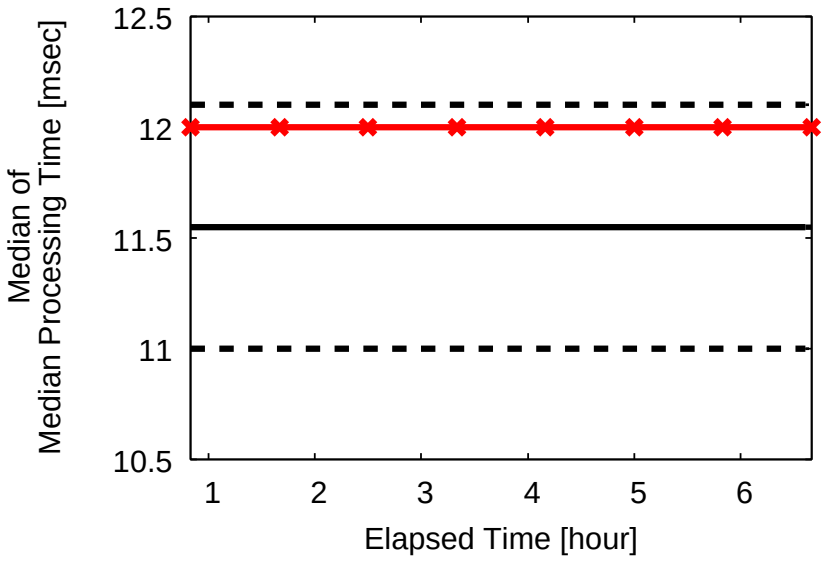
さらに，実験の経過により，在庫が複数個の商品の割合が増大した原因について述べる．クライアントエミュレータによりサーバへリクエストを送信し続けると，商品の購入と商品の出品が行われる．RUBiS では，ある商品への入札が 1 件もない時点で，出品者が定めた金額を支払えば，オークションに参加せずにその商品を即時購入することができる．この `StoreBuyNow` というリクエストは，性能異常が発生した `ViewItem` 同様に，無作為にカテゴリを選択して処理が行われる．したがって，在庫が 1 つの商品へリクエストされる割合が 20%より大きく，在庫が 1 つの商品の割合を小さくする効果をもたらしていた．さらに，`RegisterItem` というリクエストにより，商品を新たに出品することができる．こちらのリクエストも，前述の二つのリクエストの種類同様，全てのカテゴリに対して均等にリクエストされる．このリクエストは商品の在庫数を定める設定ファイルにしたがって，約 20%の割合で在庫数を複数に設定する．よって，全てのカテゴリに均等に在庫が複数個の商品が登録されていく効果をもたらしていた．

5.4.5 システム修復

上記の二つのバグを修正することで性能異常から回復できた．これを示すのが図 5.9 の二つの管理図である．1〜4 番目の打点がクライアント数が 200 のときのもの，5〜8 番目の打点がクライアント数が 300 のときのものである．図 5.9(a) と図 5.9(b) 共に，リクエスト処理時間の中央値の増大から回復できていることがわかる．ここで，図 5.9(a) においては全ての打点が上方管理限界線を超えてしまっている．しかし，これは元々の基準線計算時のリクエスト処理時間の中央値が異常に小さな値になっていたことによるものである．したがって，処理時間の大小よりも，一定であることが正常な状態である．この場合，システムの変更により処理時間の分布が変わってしまったので，第 3.5.3 節で述べたように管理図の基準線を再計算する必要がある．



(a) リクエストの種類：`ViewItem`



(b) リクエストの種類：`PutBid`

図 5.9: ケース 2 において初期化のデバッグを行った後の管理図

5.5 ケース 3

ケース 3 では，発生した性能異常の原因が全て異なるものだったため，ケース 1 と 2 に比べると原因の絞込みを速やかに行うことはできなかった．しかし，原因が異なることを知ることができたのは，原因究明に全く無意味だったわけではな

く，ある程度原因究明を支援することができた．

5.5.1 性能異常の発生

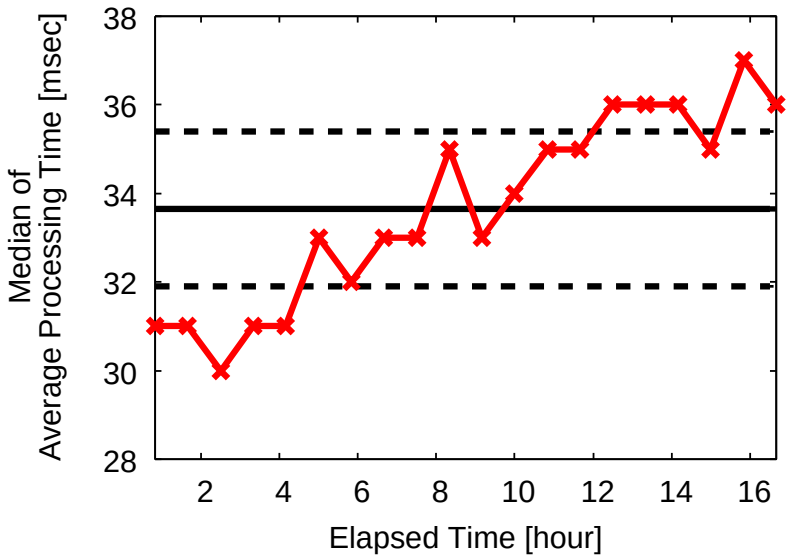
今回の実験では，まず Apache HTTP サーバの `KeepAlive` 機能をオフに設定した．第 5.3 節のケース 1 で述べた通り，この設定変更は RUBiS の性能を向上するためのものである．

また，実験を繰り返す内に発見した RUBiS のバグを修正した．このバグは，`BuyNowAuth` というリクエストの種類を処理するためのプログラムに混入していたものである．RUBiS では，ある商品への入札が 1 件も行われていない場合，出品者が定めた金額を支払うことでオークションに参加せずに即時購入することができる．このような処理を行う際にまず必要なのが `BuyNowAuth` という，購入者の認証を行うリクエストである．このリクエストを発行するためのリンクは，ある商品への入札が 1 件も行われていないときのみ，購入者に対して表示されるべきである．しかしバグにより，全ての商品においてこのリンクが表示されてしまっていた．本実験を行う前に，このバグを修正した．

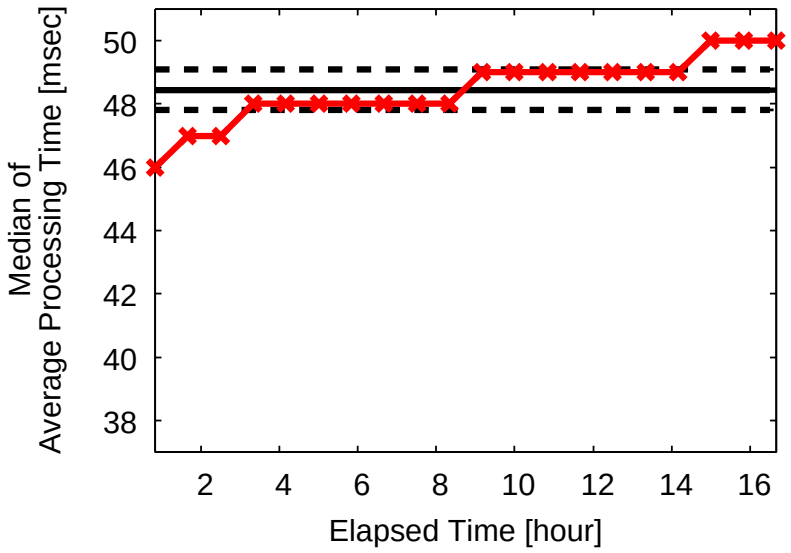
5.5.2 性能異常の検出

今回の実験では，試運用時を模してサーバに対してクライアント数 200 の負荷をかけ，管理図の基準線を計算しているときに管理図が警告を発した．第 3.5.3 節で述べたように，管理図は基準線計算時においても，計算した基準線に対して打点の位置が統計的に優れないと判断すると警告を発する．今回のケーススタディはこの状況に当てはまる．

平均値を管理する 27 の管理図の内，9 のリクエストの種類において管理図が警告を発した．そのリクエストの種類は，
`RegisterUser`，`SearchItemsInCategory`，`BrowseRegions`，`SearchItemsInRegion`，`ViewUserInfo`，`ViewBidHistory`，`BuyNow`，`PutComment`，`AboutMe`
であった．その管理図の例を図 5.10 に示した．実験が経過するにしたがって処理時間の平均値が増大していることがわかる．実験開始付近と実験終了付近で，それぞれ下方，上方管理限界線を超えた打点が行われていることがわかる．



(a) リクエストの種類：`AboutMe`



(b) リクエストの種類：`SearchItemsInCategory`

図 5.10: ケース 3 において基準線計算時に警告を発した管理図の例

5.5.3 性能異常の分類

次に提案手法ではリクエストの種類を，発生している性能異常の原因ごとに分類する．ただし，本ケースではケース 1 と 2 に比べて気をつけるべき点がある．それは，管理図の基準線計算時に性能異常を検出したことである．このため，性能

異常の発生時刻を知ることができない．ケース 1 と 2 のように適切な基準線を計算することができていれば，正しい基準線に対して打点がおかしくなった時点を性能異常の発生時刻と知ることができる．しかし，本ケースのように基準線計算時に性能異常を検出した場合，その基準線自体が正常時に得られたものではないため，各打点が適切か不適切かを判断することができない．得られる情報は，監視しているデータの分布が刻々と変化してしまっているということだけである．

性能異常の発生時刻が分からなければ，リクエストの種類を性能異常の原因ごとに分類することができない．なぜならば，分類手法では性能異常発生前後の処理時間の CDF を比較する必要があるからである．発生時刻が分からなければ，どの区間を性能異常発生前，発生後としていいか定めることができない．これに対処するために，本ケーススタディでは，実験を行った期間を半分に分け，前半部分を性能異常発生前，後半部分を性能異常発生後としてリクエストの種類を分類した．この対処法は，提案手法を厳密に適用しているとはいえないが，本ケーススタディでは問題なく働いた．

本ケーススタディでは，図 5.10 に示したように，処理時間の平均値が増大傾向を見せた性能異常が対象だったため，応急処置的な対処法の適用で問題なく動作した可能性が高い．将来的に，より複雑な性能異常にも対処していくために，性能異常の発生している期間を見極める必要があると考えている．

上記の対処によりリクエストの種類を提案手法で分類した結果，九つのリクエストの種類はそれぞれ別々のクラスタに分類された．しかも，その全てのクラスタはリクエストの種類を一つずつのみ含むものだった．よって，九つのリクエストの種類に発生している性能異常は全て別々の原因によるものだと考えることができる．このとき，得られた内の二つのシグネチャを重ね合わせた例を図 5.11 に示した．二つの性能異常シグネチャはそれぞれ `AboutMe` と `SearchItemsInCategory` というリクエストの種類において得られたものである．シグネチャ同士が重なっていることを表す黒い部分が，それぞれのシグネチャの内わずかな部分しか占めていないことが分かる．

5.5.4 原因究明の概要

本ケーススタディではまず，九つのリクエストの種類に発生した性能異常に対する原因究明結果を簡単にまとめて説明する．その後で，個々のリクエストの種類について詳しく説明を行う．まず，本ケーススタディでは残念ながら，九つのリ

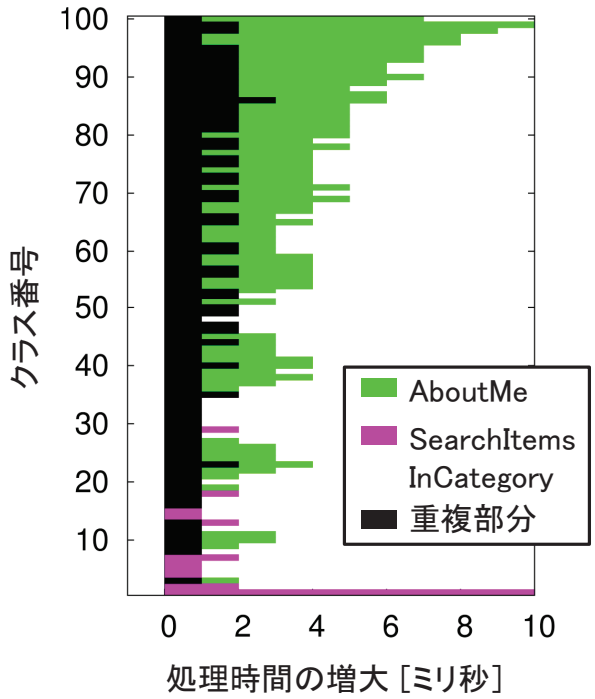
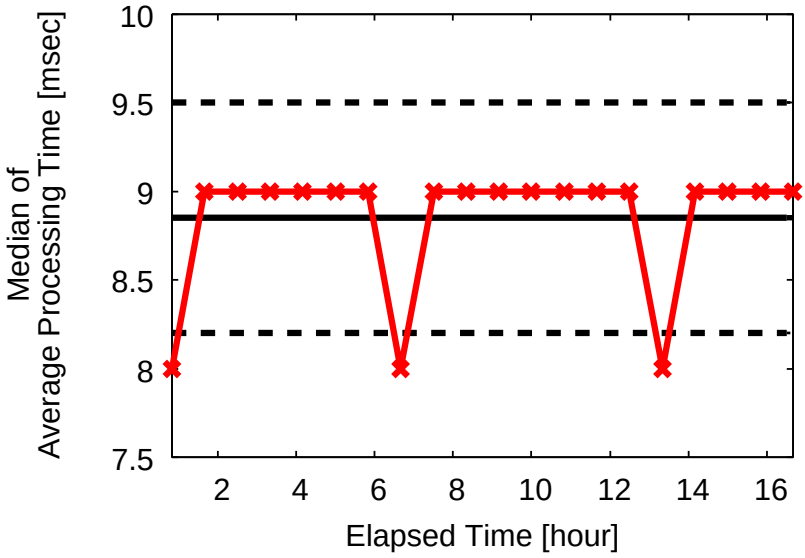


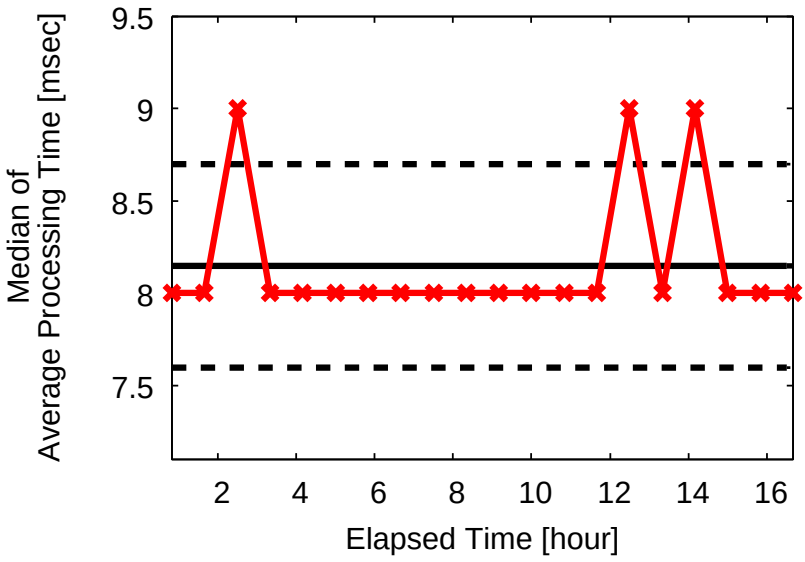
図 5.11: ケース 3 において，性能異常が発生した二つのリクエストの種類の性能異常シグネチャを重ねた例

クエストの種類で発生した性能異常の内，五つのリクエストの種類で発生した性能異常しか原因究明と回復を行うことができなかった．また，回復した五つのリクエストの種類の中でも，管理図の警告発生を完全に防ぐことができたのは 1 種類のみであった．その他の 4 種類では処理時間の平均値の増大傾向を避けることができたり，増大幅を小さくすることができたにとどまっている．提案した性能異常分類手法は，複数のリクエストの種類で同じ原因により性能異常が発生した場合に最も有効であり，一つのリクエストの種類のみで発生した性能異常の原因究明を行うのは，やや難しい場合がある．

まず，四つのリクエストの種類で発生した性能異常の原因究明が行えなかった理由について述べる．この四つのリクエストの種類では，実験を繰り返す中で性能異常が必ず発生するとは限らなかった．また，これら四つのリクエストの種類では図 5.12 に示したように，図 5.10 に示した管理図に比べ，処理時間の平均値の変化に一定の傾向が見られなかった．図 5.12 は四つの内，二つのリクエストの種類の処理時間を用いて作成した管理図である．ところどころ，打点が管理限界線を超えている．したがって，この 2 点から，この四つのリクエストの種類に発生した性能異常は，管理図が正常時の処理時間の揺らぎを性能異常として誤検知し



(a) リクエストの種類：`BrowseRegions`



(b) リクエストの種類：`RegisterUser`

図 5.12: ケース 3 において原因究明を行えなかったリクエストの種類の管理図の例

てしまった可能性が高い．

提案手法では，三つのケーススタディを通じ，管理図を利用してウェブアプリケーションの性能異常を検出することが概ねできることを確認した．しかし，その精度は 100%というわけではなく，上記の四つのリクエストの種類のように，誤

検知を行ってしまう可能性もある．リクエストはウェブアプリケーション内で複雑に処理されているので，正常時においてもある程度の揺らぎを避けることができないのが原因だと考えられる．とはいえ，三つのケーススタディで報告したように，管理図を用いて適切に性能異常を検出できている場合の方が多く，提案手法の有用性も十分あると考えている．もし，正常時の揺らぎによる誤検知を完璧に避けたいのであれば，スパイクが発生しているような警告を無視するといったようなフィルタを通した後，管理者へ警告を通知するといった対処が考えられる．スパイクが発生しているだけであれば，その後性能異常が急激に悪化する可能性は低く，原因究明を始めるのが少し遅れたとしても，大きな問題にはならない可能性が高い．

ここから，残り五つのリクエストの種類それぞれで発生した性能異常について原因究明結果を詳細に説明する．ここでいう五つのリクエストの種類とは，`SearchItemsInCategory`，`SearchItemsInRegion`，`AboutMe`，`ViewUserInfo`，`ViewBidHistory`の五つである．本ケーススタディではこの五つのリクエストの種類はそれぞれ別々のクラスタに分類されたので，原因が異なると考えて原因究明を行った．よって，原因究明時には個々のリクエストの種類のみで利用するコンポーネントに対象を絞って，詳細に調査を行った．今回は個々のリクエストの種類のみで利用するコンポーネントが怪しいということで，ケース 2 と同じく，粒度の細かいコンポーネントを対象に調査した．リクエストの入り口である Servlet からソースコードを追い，Session Bean や Entity Bean にも調査は及んだ．

5.5.5 原因究明とシステム修復 (`SearchItemsInCategory`)

まず，`SearchItemsInCategory`について説明する．このリクエストの種類に発生した性能異常は，`items`という，商品を管理するテーブルの肥大化によって引き起こされたものだった．RUBiS では，その商品に対するオークションが終了したとしても，`items`テーブルから該当データの削除を行わない．よって，実験が経過するにつれて，テーブル内のデータが増加していき，テーブルが肥大していく．このため，`SearchItemsInCategory`というリクエストの種類が，処理に必要なデータを`items`テーブルから検索するために要する時間が，実験が進むにつれて増大するという現象が発生していた．

性能異常の原因が`items`テーブルの肥大化であると突き止めた後，さらに調査

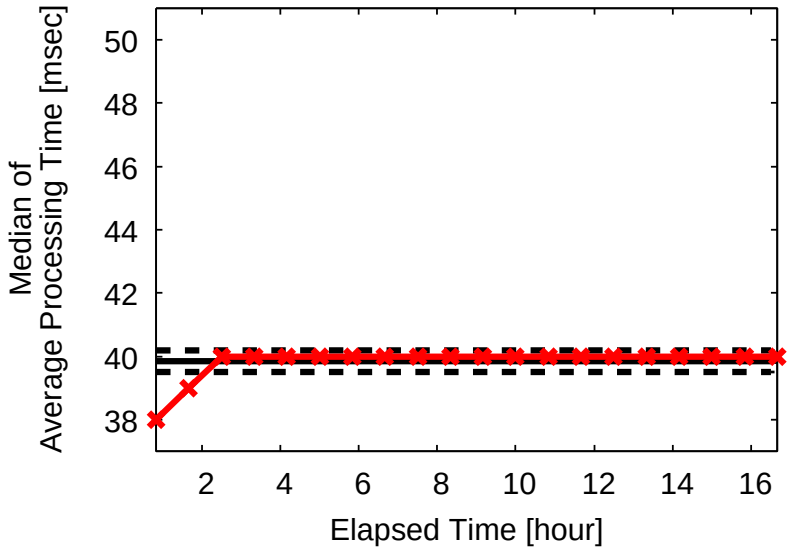
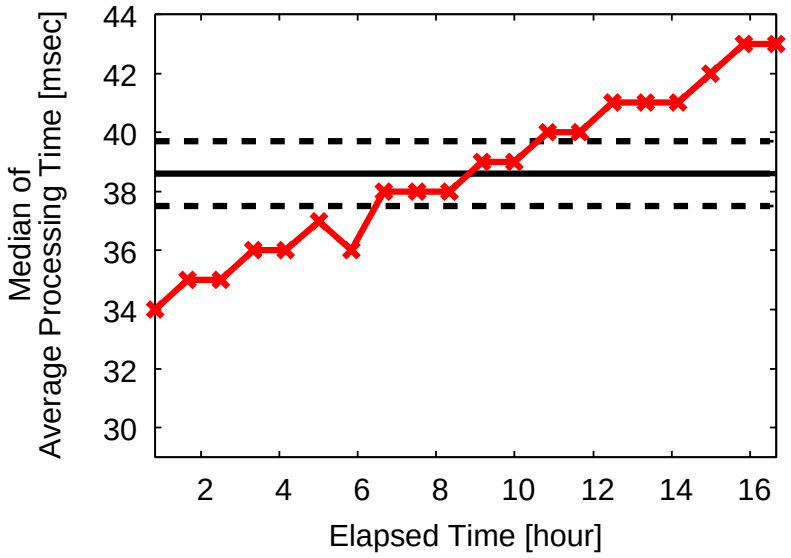


図 5.13: ケース 3 において性能異常から回復した後の SearchItemsInCategory の管理図

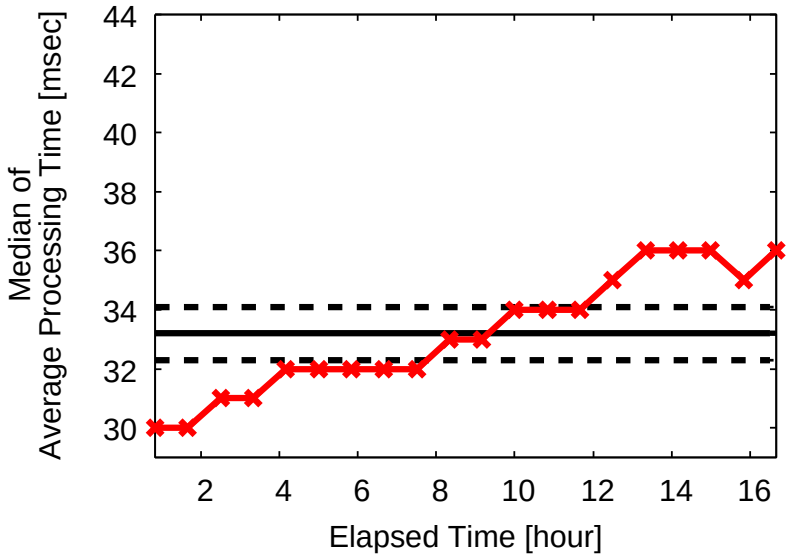
を進め，テーブルへインデックスを追加することで処理時間の増大を避けられる可能性があることを突き止めた．インデックスを追加すれば検索にかかる時間を大幅に短縮できるため，データ量が増加したとしても検索時間をほぼ一定に保つことができる．そこで，category と end_date という二つのエンティティからなるインデックスを作成した．この修正を行った後に再実験をして得られた管理図が図 5.13 である．図 5.10(b) の管理図と縦軸，横軸ともに同じスケールである．図 5.10(b) と比較すると処理時間の増大傾向が避けられていることがわかる．また，全体の処理時間も減少できている．

5.5.6 原因究明とシステム修復 (SearchItemsInRegion)

2 番目に，SearchItemsInRegion に発生した性能異常について説明する．SearchItemsInRegion でも，items テーブルの肥大化により性能異常が発生していた．よって，実験時間が経過するにしたがって処理時間が増大していた．このときの管理図を図 5.14(a) に示す．実験が経過するにつれて処理時間が増大し，実験開始直後と実験終了直前で打点が管理限界線を超えていることがわかる．性能異常を引き起こした要因は SearchItemsInCategory で発生した性能異常と同じだったものの，二つのリクエストの種類は異なる SQL 文を用いて items テーブルにアクセスするため，二つのリクエストの種類に発生した性能異常の原因は



(a) 1 度目の実験で得られた管理図



(b) 再実験後に得られた管理図

図 5.14: ケース 3 において得た `SearchItemsInRegion` の管理図

異なるといえる．

詳細に調査を進めた結果，`SearchItemsInCategory` に発生した性能異常を回復するときに必要だったインデックス以外に，二つのインデックスを追加する必要があることがわかった．`SearchItemsInCategory` と `SearchItemsInRegion`

に発生した性能異常は原因の一つを共有していたものの，全ての原因が同じではなかったため，異なるクラスタに分類された．第 4.3.2 節で述べたように，提案手法により同じクラスタに分類されるのは，原因が全く同じ場合である．

追加する必要のあったインデックスの一つは，region テーブルへの id と users エンティティからなるインデックスである．SearchItemsInCategory で述べたのと同様に，インデックスを追加することで items テーブルが肥大化したとしても，処理時間をほぼ一定に保つことができる．上記二つのインデックスを追加したときに再実験を行い得た管理図が図 5.14(b) である．処理時間の増大を抑制することができた．また，処理時間自体も小さく抑えられていることがわかる．

ただし，性能異常から完全に回復することはできなかった．性能異常を回避するために必要なもう一つのインデックスを追加することができなかったからである．SearchItemsInRegion は，users テーブルへもアクセスを行う必要がある．性能異常から回復するためには，いくつかのエンティティを items テーブルから users テーブルへ複製し，適切なインデックスを追加する必要がある．このような操作をデータベースに対して行うのは，データを保持する上でのオーバーヘッドとなる．

5.5.7 原因究明とシステム修復 (AboutMe)

3 番目に，AboutMe について説明する．AboutMe は，ユーザの登録情報を表示するリクエストの種類である．名前や出品中の商品，購入履歴などを表示する．このリクエストの種類で発生していた性能異常の原因はバグであった．AboutMe は，上述のように，過去に落札した商品を表示する．しかし，バグにより，落札できたかどうかに関わらず，過去に入札した全ての商品を表示するように実装されていた．そのため，取得するデータが急激に増加し，性能異常が発生していた．

ソースコード修正後に作成した管理図が図 5.15 である．図 5.10(a) の管理図と縦軸，横軸ともに同じスケールである．処理時間の平均値は，やや増大傾向にあるものの，全ての点が管理限界線内に打点されている．バグを除去し，性能異常から回復することができた．

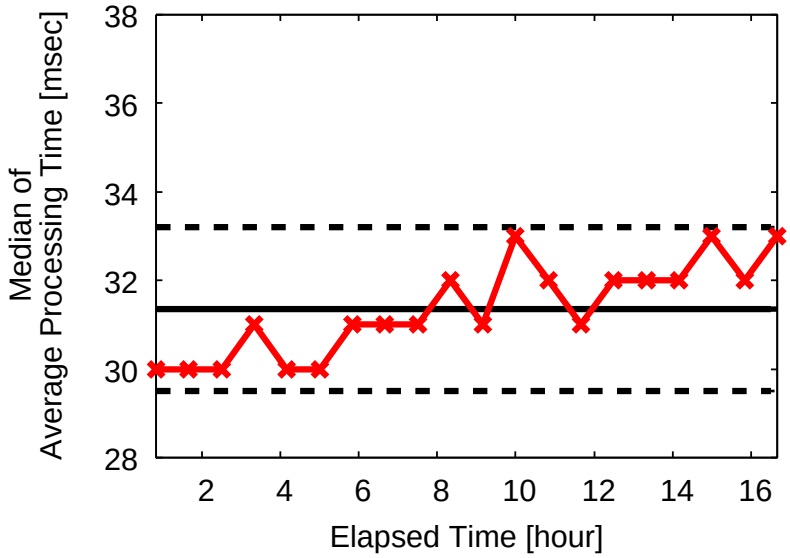


図 5.15: ケース 3 において性能異常から回復した後の AboutMe の管理図

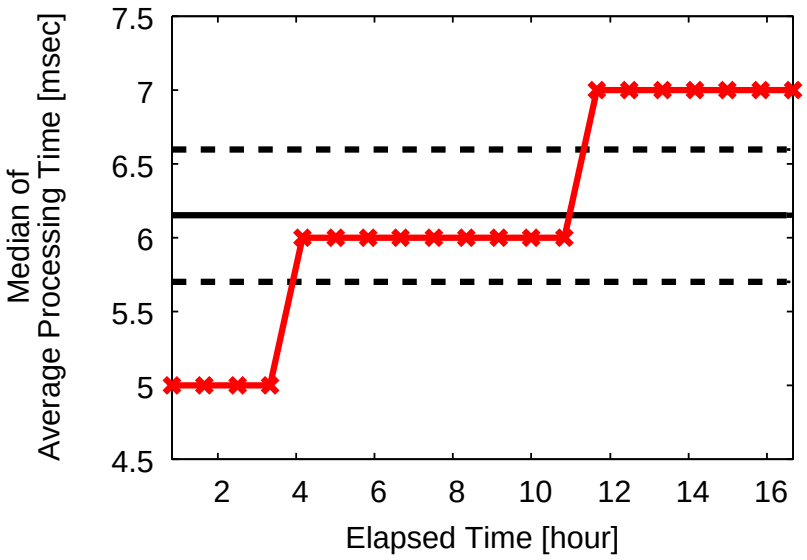
5.5.8 原因究明とシステム修復 (ViewUserInfo)

4 番目に，ViewUserInfo について説明する．性能異常発生前後の管理図を図 5.16 に示す．性能異常発生時に作成した管理図が図 5.16(a) である．処理時間が増大傾向にあり，始めに下方管理限界線を，終わりに上方管理限界線を超えて打点が行われている．

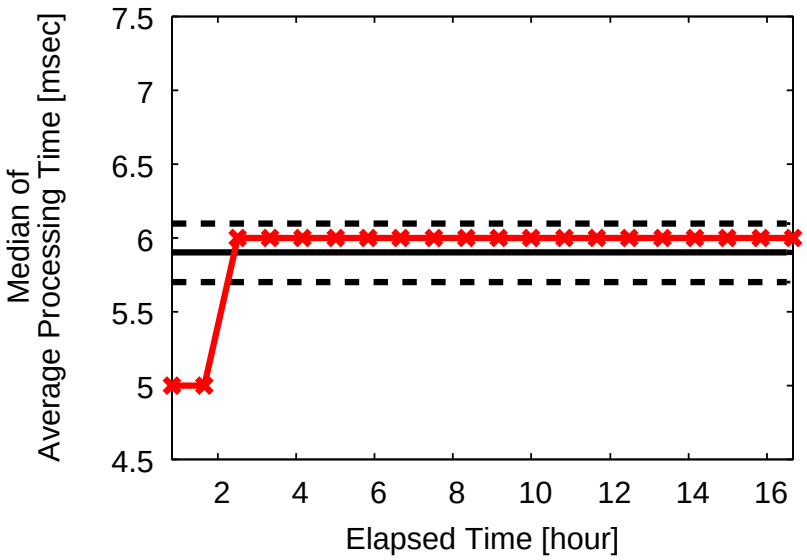
ViewUserInfo は，出品者に対する評価を表示するリクエストの種類である．他のユーザから寄せられたコメントを表示する．このリクエストの種類で発生していた性能異常の原因は，データベースの初期化が適切に行われていないことによるものだった．end_date という，オークションの終了時刻を定めるエンティティが常に，初期化時刻の 7 日後に設定されていた．そのため，実験中にオークションが終了する商品が一つもない状態になっていた．実環境に近づけるためには，ランダムに終了時刻を設定し，商品が入れ替わるようにする必要がある．

商品は，end_date で昇順にソート後，表示される．一方，RUBiS のクライアントエミュレータの実装では，前のページに表示された商品に対し，閲覧や評価を行う確率が高くなる．これらが相まって，データベース初期化時に偶然先に登録された商品の出品者にばかり，評価がつけられるという現象が起こっていた．そのため，実験が経過するにつれて，データベースからのコメントの取得に時間がかかるようになっていた．

本来ならば，実験中に，オークションが終了する商品が現れ，前のページに表



(a) 1 度目の実験で得られた管理図



(b) 再実験後に得られた管理図

図 5.16: ケース 3 において得た `ViewUserInfo` の管理図

示される商品も入れ替わるはずである．そのため，少数の出品者にばかりコメントがつけられることは起こらない．初期化を改良し，再実験を行ったときに作成した管理図が図 5.16(b) である．最初二つの打点のみ，下方管理限界線を越えてしまっているものの，その後は処理時間の増大は見受けられない．

5.5.9 原因究明とシステム修復 (ViewBidHistory)

最後に，ViewBidHistory に発生した性能異常について説明する．性能異常発生前後に ViewBidHistory で作成した管理図は図 5.17 である．性能異常発生時に作成した管理図は図 5.17(a) である．処理時間の平均値が増加し続け，実験の始めで下方管理限界線を，実験の終わりで上方管理限界線を超えて打点が行われている．

ViewBidHistory は，ある商品への入札履歴を表示するリクエストの種類である．このリクエストの種類に発生した性能異常の原因も，ViewUserInfo に発生した性能異常同様，end_date エンティティを常に初期化時刻の 7 日後に設定していたことだった．コメントの代わりに，入札が少数の商品にばかり行われていた．

end_date エンティティを適切に修正後，再実験を行った．その時に作成した管理図が図 5.17(b) である．処理時間の平均値の増大を完璧に回避することはできなかったが，抑制に成功している．

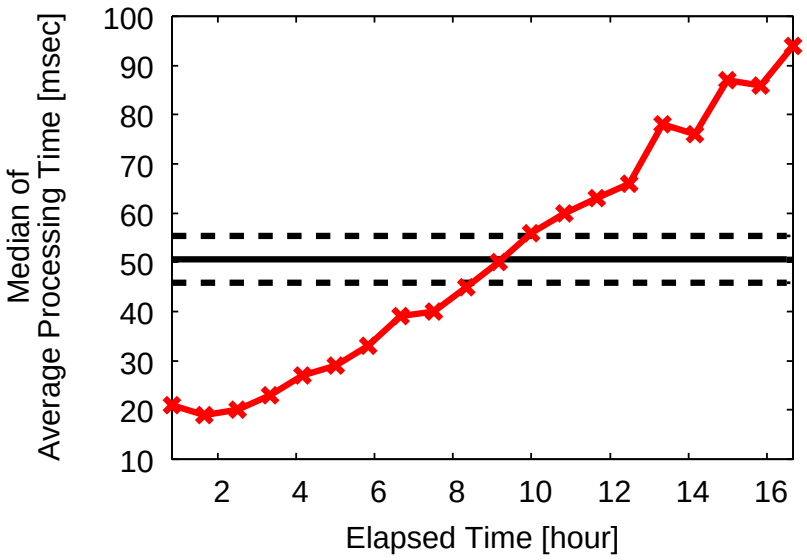
ViewUserInfo と ViewBidHistory で発生していた性能異常の原因は，どちらも end_date エンティティの設定誤りであった．しかし，ViewUserInfo ではコメントが，ViewBidHistory では入札が増加していた．よって，二つのリクエストの種類は，異なるクラスタに分類された．

5.6 性能異常分類手法に関する議論

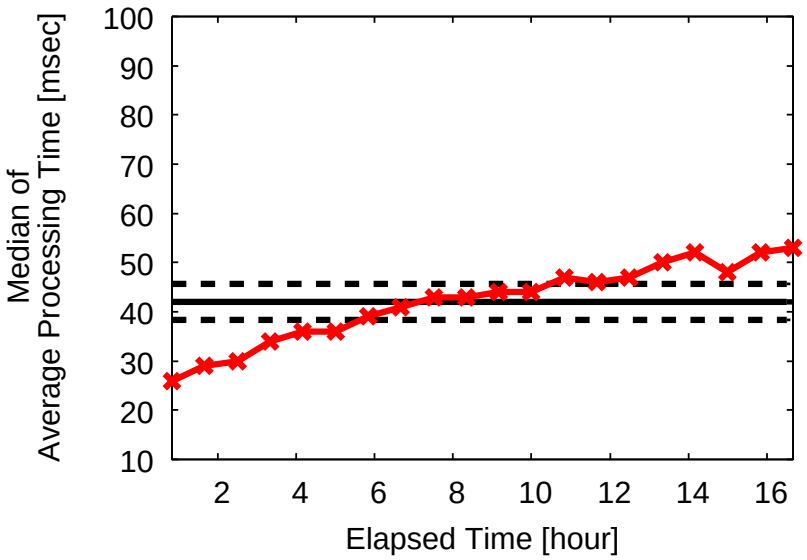
第 5.5 節で説明したケース 3 において，SearchItemsInCategory と SearchItemsInRegion はどちらも，items テーブルの肥大化により，性能異常が発生していた．また，ViewUserInfo と ViewBidHistory はどちらも，end_date エンティティの設定ミスで性能異常が発生していた．

管理者としては，SearchItemsInCategory と SearchItemsInRegion，ViewUserInfo と ViewBidHistory それぞれが同じクラスタに分類された方が，その後の原因究明を円滑に行える可能性が高い．新たな正規化の方法を導入すれば，それが行える可能性がある．

図 5.18 は，ViewUserInfo と ViewBidHistory それぞれの性能異常シグネチャである．ここで，図 5.18(a) と図 5.18(b) では，横軸のスケールが異なることに注意してほしい．



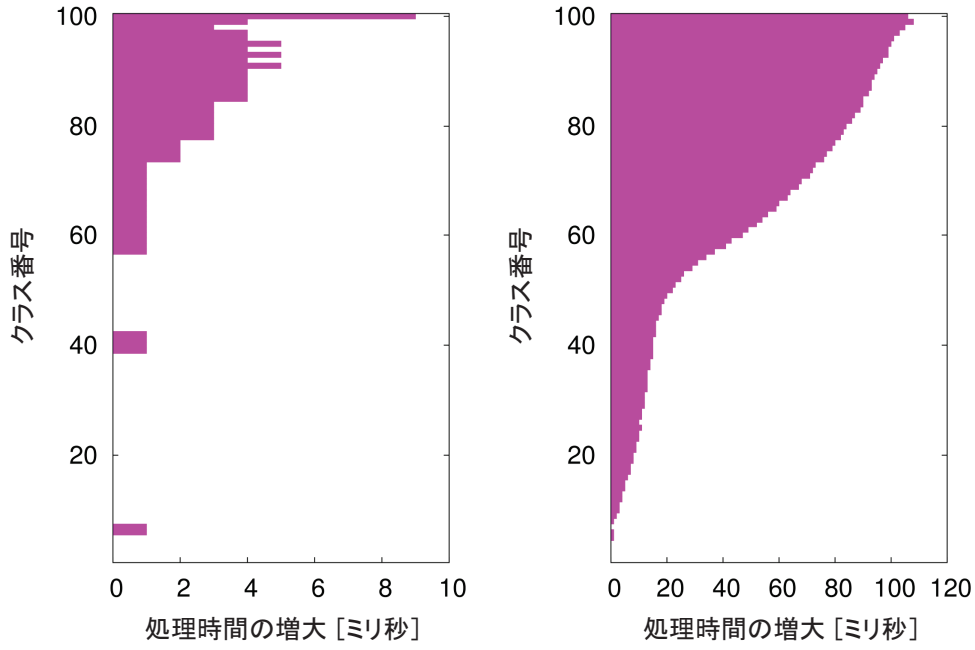
(a) 1 度目の実験で得られた管理図



(b) 再実験後に得られた管理図

図 5.17: ケース 3 において得た `ViewBidHistory` の管理図

このように，図 5.18 の二つの棒グラフは，処理時間の増大量は大きく異なるものの，形は似ている．よって，全体の面積で正規化を行ったり，棒グラフの高さを合わせるような正規化を行えば，二つのリクエストの種類は同じクラスタに分類できる可能性がある．



(a) `ViewUserInfo`　　(b) `ViewBidHistory`

図 5.18: ケース 3 において得た性能異常シグネチャ

しかし，このような正規化を行った場合，異なる性能異常にも関わらず同じクラスタに分類してしまう可能性も増大する．よって，本論文では，現在の正規化方法が適当であると考えている．`SearchItemsInCategory` と `SearchItemsInRegion`，`ViewUserInfo` と `ViewBidHistory` がそれぞれ異なるクラスタに分類されるのは仕方ない．

5.7　まとめ

本章では，提案手法の有用性を示すためにケーススタディを報告した．それぞれのケーススタディでは，提案手法を用いることで性能異常を早期に検出し，かつ原因究明の支援を行うことができた．全てのケーススタディを通じて，性能異常検出時の処理時間の平均値の増大は高々110 ミリ秒程度であった．また，提案手法による分類結果の支援を受け，原因をサーバ単位やメソッド単位で絞り込むことができた．

また，提案手法はシステムを変更せずに導入可能であったため，導入時のコストが低く，導入が容易だといえる．本実験ではクライアントエミュレータの機能

を用いてリクエスト処理時間と URL を記録したが，Apache HTTP サーバのアクセスログを用いて同様のことを行うこともできる．

第6章　結論

本章では，本論文をまとめる．そして，今後の研究の展望についても述べる．

6.1　本研究のまとめ

本論文では，ウェブアプリケーションにおける性能異常の検出と原因究明支援を行う手法を提案した．提案手法は，検出と原因究明の能力だけではなく，導入の容易さも重視している．既存手法では，どちらか一方のみを重視していた．

提案手法では，処理時間の監視をリクエストの種類ごとに行う．そうすることで，ソフトウェアコンポーネントなど，より細かい粒度で監視する手法に比べて導入を容易にしている．ソフトウェアコンポーネント単位で処理時間を監視するためには，そのための機構をシステム内のさまざまな個所に実装する必要がある．一方，リクエストの種類単位であれば，既存の監視機構を用いて処理時間を取得できる．通常のサーバでは，リクエストごとに URL と処理時間を記録している．また，もし仮に監視機構を備えていないシステムに適用する場合でも，実装はリクエストの入り口 1 か所に限定できる．

リクエストの種類ごとに計測した処理時間に統計処理を施し，性能異常の検出と原因究明を行う．管理図を用いて，処理時間の分布に生じる微小な変化を見逃さずに監視するため，異常を早期に検出することができる．さらに，処理時間の分布の変化傾向を性能異常シグネチャとして抽出することで，発生している性能異常を特徴づけることができる．そうすることで，同時に複数の性能異常が発生した際，それらの原因が同じか異なるかを自動で判定することができる．リクエストの種類を性能異常の原因ごとに分類することができれば，各クラスタが共通して利用しているコンポーネントを探すことで，細かく性能異常の原因を絞り込むことができる．

提案手法の有用性を示すために，ケーススタディを報告した．一般的なウェブアプリケーション提供基盤同様，3 層構造で実験環境を構築し，その上でオークションサイト RUBiS を動作させた．性能異常を引き起こすため，クライアント数

やシステム設定をさまざまに変更し，提案手法で処理時間を監視した．性能異常検出時，処理時間の平均値の増大は高々110 ミリ秒程度であり，異常を早期に検出できたといえる．その後，性能異常の分類結果による支援を受け，原因をサーバ単位やメソッド単位で絞り込むことができた．実際にシステムを修復後，再実験し，性能異常から回復できることを確認した．

6.2 今後の展望

今後の研究展望としては，原因究明時の性能異常に優先順位をつける手法の考案が必要だと考えている．本論文のケーススタディでも報告したように，性能異常は同時に複数発生することがある．原因究明には人的，時間的にコストがかかるため，そのような場合には優先度を設け，優先度の高いものから修復を行う必要がある．

優先度の例としては，次のようなものが挙げられる．まず一つ目に，根本的な原因を優先的に解決する必要がある．性能異常は，ある異常が他の異常を引き起こすといった形で伝播することがある．例えば，コネクション不足の場合を考える．データベースサーバとアプリケーションサーバの間のコネクションが不足すると，アプリケーションサーバにリクエストが停滞してしまう．その結果，アプリケーションサーバとウェブサーバ間のコネクションも不足し，合計 2 か所で性能異常が発生することになる．この場合，データベースサーバとアプリケーションサーバ間のコネクションを優先的に修復する必要がある．

二つ目の例として，原因究明の難易度が低い性能異常を優先的に解決する必要がある．難易度が高いほど，原因究明に時間がかかる．二つの性能異常で原因究明にかかる時間が異なる場合，どちらの性能異常から回復したとしても，最終的に全ての性能異常からの回復にかかる時間は同じである．しかし，難易度の低い性能異常から迅速に回復してしまう方が，時間のかかる方を先に原因究明する場合に比べ，両方の性能異常が発生している時間が短くなる．結果として，被害を小さく抑えることができる．

上記のような優先順位を体系的に定め，自動で優先順位を判別するアルゴリズムを考案する必要がある．本論文の実験を行う上で，原因究明時の優先順位に関する知見を得ることができた．しかし，現時点では場当たり的に対処することしかできていない．自然言語ではなく，プログラムとして記述可能な形式にまとめる必要がある．

謝辞

本論文は著者が慶應義塾大学大学院理工学研究科開放環境科学専攻の後期博士課程に在籍中の研究成果をまとめたものです．本研究を行うにあたって，また本論文をまとめるにあたって，多くの方々からご指導とご協力を賜りました．お世話になった全ての方々へこの場を借りて御礼申し上げます．

まず，本論文の主査であり，著者の指導教員である慶應義塾大学理工学部情報工学科 河野健二准教授に深く感謝いたします．河野健二准教授には，慶應義塾大学 4 年生時から修士課程，後期博士課程と 6 年間もの長きにわたってさまざまなことをご教授いただきました．特に，後期博士課程進学という，それまで自分が考えもしなかった道へ進むきっかけを作ってくださり，私の人生に非常に大きな影響を与えてくださいました．それまで人からものを教わることが勉強だと思っていた私に，自分の頭で考えていく研究の楽しさを教えてくださいました．河野健二准教授に出会っていなかったら，おそらく後期博士課程に進学していなかったのではないかと思います．真剣に，私が納得いくまで研究の相談に乗ってくださり，私にとっては理想的な指導教員でした．心より感謝いたします．

次に，本論文の副査を担当していただいた，慶應義塾大学理工学部情報工学科 山本喜一教授，天野英晴教授，高田眞吾准教授に感謝いたします．副査の皆様には貴重なお時間を割いていただき，本論文を丁寧に査読していただきました．副査の皆様との有意義な議論によって，本論文の完成度が向上したと実感しております．深く感謝いたします．

筑波大学システム情報系情報工学域 杉木章義助教に感謝いたします．杉木章義助教には学部 4 年生時に卒業研究のテーマを与えていただき，研究指導もしていただきました．卒業研究時のテーマを発展させていったおかげで，本論文をまとめることができました．研究でご一緒できたのは 1 年間という短い期間でありましたが，杉木章義助教のご指導なしには本論文は完成しなかったと感じています．

慶應義塾大学理工学部情報工学科 河野研究室の皆様に感謝いたします．優秀な仲間に囲まれ刺激を受けることで，時に辛い研究も途中で諦めることなく完成さ

せることができました．また，気の合う仲間たちと研究以外でも楽しく過ごせたのも，私の研究を助けてくださったと感じています．

本論文の研究を進めるにあたって使用した実験機材の一部，および本研究の原著論文の発表にあたっては，科学技術振興機構 CREST による支援をいただきました．また，慶應義塾先端科学技術研究センターの KLL 後期博士課程研究助成金から本研究に対する支援をいただきました．さらに，日本学生支援機構奨学金や藤原奨学金は，著者の研究活動の大きな支えとなりました．ここに感謝いたします．

最後に，経済的支援を惜しまず，著者の博士課程進学を支持し，現在まで暖かく見守っていただきました両親と妹に心より感謝いたします．

参考文献

[1] Java Platform, Enterprise Edition. http://www.oracle.com/technetwork/jp/java/javaee/overview/index.html.

[2] Google. www.google.com/.

[3] This is your pilot speaking. Now, about that holding pattern... http://googleblog.blogspot.com/2009/05/this-is-your-pilot-speaking-now-about.html.

[4] YAHOO! JAPAN. http://www.yahoo.co.jp/.

[5] Amazon Web Services. http://aws.amazon.com/jp/.

[6] YAHOO! JAPAN オークション，システム障害について. http://topic.auctions.yahoo.co.jp/notice/maintenance/post_231/.

[7] 「クラウドは飛行機」？　Amazon の大規模障害で見えた企業の対応. http://cloud.watch.impress.co.jp/docs/column/infostand/20110509_444082.html.

[8] 楽天ネット証券. https://www.rakuten-sec.co.jp/.

[9] 楽天ネット証券，システムの障害や不具合に関する履歴. https://www.rakuten-sec.co.jp/web/company/failure/archive.html.

[10] Greg Linden. Make Data Useful. http://www.scribd.com/doc/4970486/Make-Data-Useful-by-Greg-Linden-Amazoncom.

[11] Amazon.com. http://www.amazon.com/.

[12] Apache HTTP Server Project. http://httpd.apache.org/.

[13] JBoss Application Server. http://www.jboss.org/jbossas.

[14] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UC Berkeley Computer Science Technical Report UCB//CSD-02-1175, Berkeley Computer Science, 2002.

[15] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. Whodunit: Transactional Profiling for Multi-Tier Applications. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '07)*, pp. 17–30, March 2007.

[16] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, pp. 259–272, December 2004.

[17] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-Based Failure and Evolution Management. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, pp. 309–322, March 2004.

[18] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP '03)*, pp. 74–89, October 2003.

[19] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeffrey S. Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, pp. 231–244, December 2004.

[20] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting Large-Scale System Problems by Mining Console Logs. In *Proceedings of*

the ACM symposium on Operating systems principles (SOSP '09), pp. 117–132, October 2009.

[21] Jian-Guang LOU, Qiang FU, Shengqi YANG, Ye XU, and Jiang LI. Mining Invariants from Console Logs for System Problem Detection. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '10)*, pp. 1–14, June 2010.

[22] Project Darkstar. `http://java.net/projects/games-darkstar`.

[23] Apache Hadoop. http://hadoop.apache.org/.

[24] Peter Bodík, Greg Friedman, Lukas Biewald, Helen Levine, George Candea, Kayur Patel, Gilman Tolle, Jon Hui, Armando Fox, Michael I. Jordan, and David Patterson. Combining Visualization and Statistical Analysis to Improve Operator Confidence and Efficiency for Failure Detection and Localization. In *Proceedings of the IEEE International Conference on Autonomic Computing (ICAC '05)*, pp. 89–100, June 2005.

[25] シューハート管理図. JIS Z 9021.

[26] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '02)*, pp. 595–604, June 2002.

[27] Nikolai Joukov, Avishay Traeger, Rakesh Iyer, Charles P. Wright, and Erez Zadok. Operating System Profiling via Latency Analysis. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, pp. 89–102, April 2006.

[28] Avishay Traeger, Ivan Deras, and Erez Zadok. DARC: Dynamic Analysis of Root Causes of Latency Distributions. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '08)*, pp. 277–288, June 2008.

[29] Nate Kushman and Dina Katabi. Enabling Configuration-Independent Automation by Non-Expert Users. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, pp. 1–14, October 2010.

[30] Mike Chen, Emre Kiciman, Anthony Accardi, Armando Fox, and Eric Brewer. Using Runtime Paths for Macroanalysis. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '03)*, pp. 79–84, May 2003.

[31] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Symposium on Operating Systems Principles (SOSP '01)*, pp. 230–243, 2001.

[32] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: online modelling and performance-aware systems. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS '03)*, pp. 85–90, May 2003.

[33] Steve Zhang, Ira Cohen, Moises Goldszmidt, Julie Symons, and Armando Fox. Ensembles of Models for Automated Diagnosis of System Performance Problems. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '05)*, pp. 644–653, June 2005.

[34] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. Capturing, Indexing, Clustering, and Retrieving System History. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP '05)*, pp. 105–118, October 2005.

[35] Peter Bodík, Moises Goldszmidt, Armando Fox, Dawn B. Woodard, and Hans Andersen. Fingerprinting the Datacenter: Automated Classification of Performance Crises. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys 2010)*, pp. 111–124, April 2010.

[36] Sapan Bhatia, Abhishek Kumar, Marc E. Fiuczynski, and Larry Peterson. Lightweight, High-Resolution Monitoring for Troubleshooting Production Systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implimentation (OSDI '08)*, pp. 103–116, 2008.

[37] Christopher Stewart and Kai Shen. Performance Modeling and System Management for Multi-component Online Services. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, pp. 71–84, May 2005.

[38] Wei Zheng, Ricardo Bianchini, G. John Janakiraman, Jose Renato Santos, and Yoshio Turner. JustRunIt: Experiment-Based Management of Virtualized Data Centers. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '09)*, pp. 1–16, June 2009.

[39] Zhichun Li, Ming Zhang, Zhaosheng Zhu, Yan Chen, Albert Greenberg, and Yi-Min Wang. WebProphet: Automating Performance Prediction for Web Services. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI '10)*, pp. 1–16, April 2010.

[40] Christopher Stewart, Terence Kelly, Alex Zhang, and Kai Shen. A Dollar from 15 Cents: Cross-Platform Management for Internet Services. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '08)*, pp. 199–212, June 2008.

[41] Christopher Stewart, Terence Kelly, and Alex Zhang. Exploiting Nonstationarity for Performance Prediction. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys '07)*, pp. 31–44, March 2007.

[42] Christopher Stewart, Ming Zhong, Kai Shen, and Thomas O'Neill. Comprehensive Depiction of Configuration-dependent Performance Anomalies in Distributed Server Systems. In *Proceedings of the USENIX Workshop on Hot Topics in System Dependability (HotDep '06)*, pp. 1–6, November 2006.

[43] Kiran Nagaraja, Fabio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, pp. 61–76, December 2004.

[44] Yan Zhang, Wei Qu, and Anna Liu. Automatic Performance Tuning for J2EE Application Server Systems. In *Proceedings of the International Conference on Web Information Systems Engineering (WISE '05)*, pp. 520–527, November 2005.

[45] Akiyoshi Sugiki, Kenji Kono, and Hideya Iwasaki. Tuning mechanisms for two major parameters of apache web servers. *Software: Practice and Experience*, Vol. 38, No. 12, pp. 1215–1240, October 2008.

[46] Wei Zheng, Ricardo Bianchini, and Thu D. Nguyen. Automatic Configuration of Internet Services. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys '07)*, pp. 219–229, March 2007.

[47] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H. Xia, and Li Zhang. A Smart Hill-Climbing Algorithm for Application Server Configuration. In *Proceedings of the International World Wide Web Conference (WWW '04)*, pp. 287–296, May 2004.

[48] IBM WebSphere Application Server. `http://www-01.ibm.com/software/webservers/appserv/was/`.

[49] Takayuki Osogami and Sei Kato. Optimizing System Configurations Quickly by Guessing at the Performance. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, pp. 145–156, June 2007.

[50] Byung Chul Tak, Chunqiang Tang, Chun Zhang, Sriram Govindan, Bhuvan Urgaonkar, and Rong N. Chang. vPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC '09)*, pp. 1–14, June 2009.

[51] Xu Chen, Ming Zhang, Z. Morley Mao, and Paramvir Bahl. Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pp. 117–130, December 2008.

[52] Kenji Yamanishi and Yuko Maruyama. Dynamic Syslog Mining for Network Failure Monitoring. In *Proceedings of the ACM International Conference on Knowledge Discovery in Data Mining (KDD '05)*, pp. 499–508, August 2005.

[53] Miao Jiang, Mohammad A. Munawar, Thomas Reidemeister, and Paul A.S. Ward. Automatic Fault Detection and Diagnosis in Complex Software Systems by Information-Theoretic Monitoring. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '09)*, pp. 285–294, June 2009.

[54] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, pp. 1–14, October 2010.

[55] Kelvin C. W. So and Emin G´un Sirer. Latency and Bandwidth-Minimizing Failure Detectors. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys '07)*, pp. 89–99, March 2007.

[56] Ya-Yunn Su, Mona Attariyan, and Jason Finn. AutoBash: Improving configuration management with operating system causality analysis. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP '07)*, pp. 237–250, October 2007.

[57] Ya-Yunn Su and Jason Finn. Automatically Generating Predicates and Solutions for Configuration Troubleshooting. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC '09)*, pp. 1–13, June 2009.

[58] Fábio Oliveira, Andrew Tjang, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Barricade: Defending Systems Against Operator Mistakes. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys '10)*, pp. 83–96, April 2010.

[59] Paul D. Marinescu, Radu Banabic, and George Candea. An Extensible Technique for High-Precision Testing of Recovery Code. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '10)*, pp. 1–14, June 2010.

[60] Ming Zhong, Kai Shen, and Joel Seiferas. Replication Degree Customization for High Availability. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys '08)*, pp. 55–68, March 2008.

[61] Siddhartha Sen, Wyatt Lloyd, and Michael J. Freedman. Prophecy: Using History for High-Throughput Fault Tolerance. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI '10)*, pp. 1–16, April 2010.

[62] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot - A Technique for Cheap Recovery. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, pp. 31–44, December 2004.

[63] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amrasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically Patching Errors in Deployed Software. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP '09)*, pp. 87–102, October 2009.

[64] Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. Automated Known Problem Diagnosis with Event Traces. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys '06)*, pp. 375–388, April 2006.

[65] Chinghway Lim, Navjot Singh, and Shalini Yajnik. A Log Mining Approach to Failure Analysis of Enterprise Telephony Systems. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '08)*, pp. 398–403, June 2008.

[66] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, pp. 245–258, December 2004.

[67] 上級ユーザー向けの Windows レジストリ情報. `http://support.microsoft.com/kb/256986`.

[68] Kirk Glerum, Kinshman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP '09)*, pp. 103–116, October 2009.

[69] Alexander V. Mirgorodskiy, Naoya Maruyama, and Barton P. Miller. Problem Diagnosis in Large-Scale Computing Environments. In *Proceedings of the IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC '06)*, pp. 1–13, November 2006.

[70] Adam Oliner and Jon Stearley. What Supercomputers Say: A Study of Five Systems Logs. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*, pp. 575–584, June 2007.

[71] Aaron B. Brown and David A. Patterson. Undo for Operators: Building an Undoable E-mail Store. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '03)*, pp. 1–14, June 2003.

[72] RUBiS: Rice University Bidding System. `http://rubis.ow2.org/`.

[73] 中村達男. 管理図の作り方と活用. 日本規格協会, 1999.

[74] Nong Ye, Connie Borror, and Yebin Zhang. Ewma techniques for computer intrusion detection through anomalous changes in event intensity. *Quality and Reliability Engineering International*, Vol. 18, No. 6, pp. 443–451, August 2002.

[75] 鐵健司. 新版　品質管理のための統計的方法入門. 日科技連出版社, 2000.

[76] Peter Bodík, Rean Griffith, Charles Sutton, Armando Fox, Michael Jordan, and David Patterson. Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '09)*, p. 5 pages, June 2009.

[77] Jeremy Philippe, Noël De Palma, Sara Bouchenak, Fabienne Boyer, and Daniel Hagimont. A Black-Box Approach for Web Application SLA. In *Proceedings of the ACM Symposium on Applied Computing (SAC '06)*, pp. 807–808, April 2006.

[78] Guide to Application Performance-Based Service Level Agreements. `http://communities.quest.com/community/foglight/blog/2011/02/18/apm-service-level-agreements`.

[79] eBay.com. `http://www.ebay.com/`.

[80] MySQL Community Server. http://www-jp.mysql.com/downloads/mysql/.

[81] Apache Commons HTTP Client. http://jakarta.jp/commons/httpclient/index.html.