情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2011-ARC-195 No.18
Vol.2011-OS-117 No.18
2011/4/14

リクエスト処理時間の変化に着目した性能異常分類手法

岩田　聡[†1]　河野健二[†1,†2]

近年，ウェブアプリケーションの信頼性向上が求められている．その一方でウェブアプリケーションは複雑になってきており，異常発生時に原因究明を支援する機構が必要になってきている．本論文では発生した性能異常を原因ごとに分類する手法を提案する．性能異常は同時に複数発生することがあり，それぞれの性能異常の原因が同じか異なるかを判定することは原因究明にとって有益である．提案手法では，リクエスト処理時間が原因ごとに固有な変化をすることが多いという知見に基づき，異常の類似度を計算する．

Clustering Performance Anomalies Based on Root Causes

Satoshi Iwata[†1] and Kenji Kono[†1,†2]

Performance anomalies in web applications are becoming a big problem. The increasing complexity of modern web applications makes it much more difficult to identify root causes of performance anomaly. Making matters worse, a performance anomaly can occur from several root causes in a web application; two or more faults combine to incur the performance anomaly. In this paper, we propose a novel method that helps us to narrow down suspicious components. This method clusters anomalous requests based on their root causes. The key insight behind our method is that the measurements of anomalous requests that suffer from the same root cause deviates similarly from the standard measurements.

†1 慶應義塾大学
Keio University
†2 科学技術振興機構 CREST
CREST, Japan Science and Technology Agency

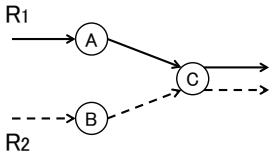


図 1　性能異常分類手法が有用な例　リクエスト R_1 はコンポーネント A と C を利用し，R_2 は B と C を利用する．ここでいうコンポーネントとはマシンやサーバソフトウェア，EJB（Enterprise Java Beans）などであり，特に明確な制限はない．

1. はじめに

近年，ウェブアプリケーションの性能異常が重要な問題になりつつある．本論文でいう性能異常とは，応答時間の増加やスループットの低下を指す．性能異常は SLA（service level agreement）違反による損害賠償や応答性悪化によるユーザ数減少を引き起こす可能性があるため，対処する必要がある．

ウェブアプリケーションの複雑化に伴い，性能異常の原因究明支援手法が求められている．近年の大規模なウェブアプリケーションはさまざまなコンポーネントによって構成されており，その発生箇所を特定することですら簡単ではない．原因究明支援手法を利用して性能異常から迅速に復旧することができれば，被害を最小限に抑えることができる．

本論文では原因究明支援手法として**性能異常分類手法**を提案する．一度に複数の性能異常が発生した場合，まずその性能異常の原因が同じか異なるかを究明する必要があり，提案手法はそれを行う．この性能異常分類手法は，例えば**図 1** の場合に有用である．リクエスト 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．1 つめはコンポーネント C のみが原因の場合，2 つめはコンポーネント A と B それぞれが原因の場合である．提案手法を用いてリクエスト R_1 と R_2 に発生している性能異常が同じだと分かれば C が原因だと考えられるし，異なると分かれば A と B が原因だと考えられる．

監視を細粒度に行うことで性能異常分類を行わずに発生箇所を特定することができるが，そのアプローチには限界があるといえる．図 1 において監視をリクエスト単位ではなくコンポーネント単位で行えば，性能異常を分類することなく原因のコンポーネントを特定できる．しかし，オーバーヘッドの観点からこれを行うことは難しい．監視粒度をリクエスト，EJB，メソッドと細かくしていき，全ての監視対象が互いに処理を共有しない粒度まで監視

Vol.2011-ARC-195 No.18
Vol.2011-OS-117 No.18
2011/4/14

対象を細かくする必要があるが，一方で監視に必要なオーバーヘッドは増加する．

提案手法は性能異常の分類を処理時間の変化傾向に着目して行う．各監視対象ごとに，処理時間の変化傾向をシグネチャとし，そのシグネチャが似ていれば発生している性能異常は同じだと考える．逆に似ていなければ性能異常は異なると考える．我々は，同じ原因による性能異常は処理時間を同様に変化させる可能性が高いと考えている．性能異常の検出は既存研究[1)–4)] に任せ，提案手法は性能異常の分類のみ行う．

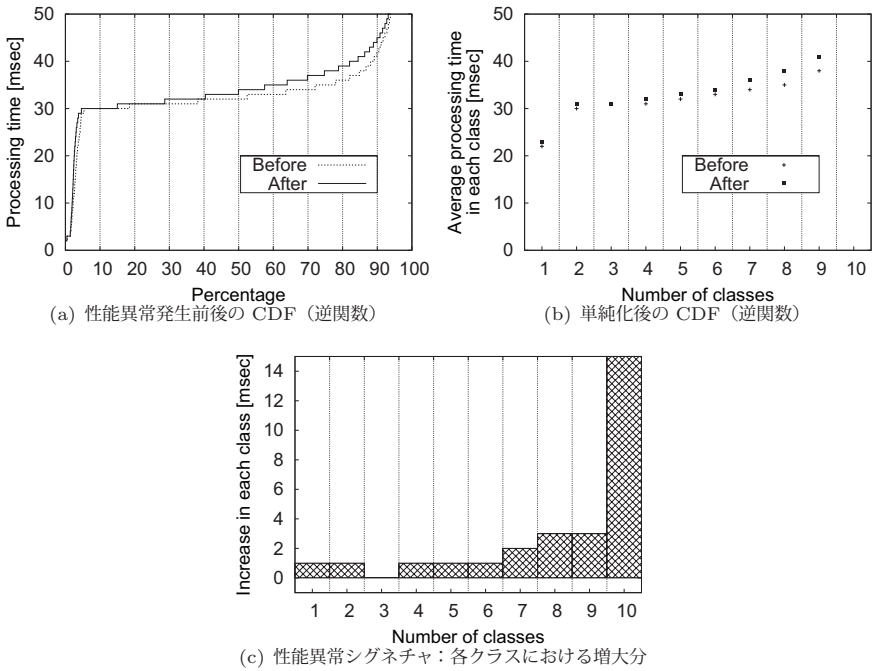
提案手法の有用性を示すために，提案手法を用いて性能異常を分類したケーススタディを示す．性能異常分類結果を用いて原因を絞り込み，性能異常から復旧することができた．発生箇所はサーバソフトウェアやメソッド単位で絞り込むことができた．それぞれで解決できた性能異常はコネクション数の設定誤りとデータベース初期化の不備であった．実験ではウェブアプリケーションに，eBay.com[5)] を模したオークションサイトである RUBiS[6)] を用いた．

本論文の構成を以下に示す．2 節で提案手法を説明する．3 節で，提案手法を用いて性能異常の分類を行えたケーススタディを示す．4 節で関連研究を紹介し，最後に 5 節で本論文をまとめる．

2. 提案手法

提案手法では，処理時間の変化傾向に着目して性能異常の分類を行う．性能異常は原因により変化傾向が異なることが多い．例えば，コネクション不足の発生初期では，一部のリクエスト処理時間のみ異常に大きくなり，その他のリクエスト処理時間は平常時と同じ値となる．これは，一部のリクエストのみ待ち行列に入れられることによる．一方，データベースの肥大化による性能異常発生時は，ほとんど全てのリクエスト処理時間がわずかずつ大きくなることが多い．これは，テーブルの肥大化がレコードの検索時間の増大を引き起こすためである．

このような変化傾向の違いの把握を，提案手法は 3 つのステップを設けて行っている．1 つめは，各監視対象における処理時間の変化傾向をシグネチャとして表現するステップである．2 つめは，1 つめのステップで得たシグネチャを比較し，処理時間の変化傾向の類似度を計算するステップである．最後に 3 つめのステップで，2 つめのステップで得た各性能異常の類似度を基に性能異常を原因ごとに分類する．本節では，以下 3 つの小節に分けてそれぞれのステップを詳しく説明する．



(a) 性能異常発生前後の CDF（逆関数）
(b) 単純化後の CDF（逆関数）
(c) 性能異常シグネチャ：各クラスにおける増大分

図 2 性能異常シグネチャの作成

2.1 性能異常シグネチャ

我々の提案手法で利用する性能異常シグネチャは処理時間の “分布” の変化傾向を表現する．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，かつスケジューリングの影響などの無駄な情報を排除することを目標にしている．平均値や最大値など統計値の変化量を単純に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可能性もある．

提案手法では性能異常シグネチャを**図 2(c)** のように棒グラフの形式で表す．シグネチャとなる棒グラフ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）から計算した差を基に作成する．CDF は統計値と異なり分布を表すため，処理時間の分布の変化傾向

Vol.2011-ARC-195 No.18
Vol.2011-OS-117 No.18
2011/4/14

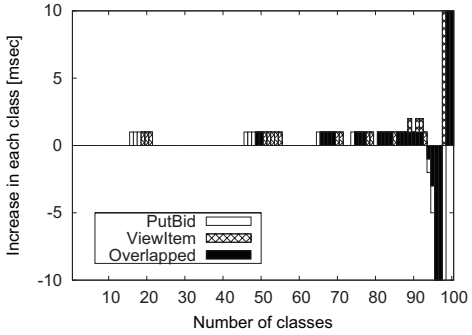


図 3 2 つのシグネチャを重ね合わせた例　RUBiS におけるリクエストの種類である `PutBid` と `ViewItem` のシグネチャを重ね合わせた．黒い領域が重なった部分である．

を特徴付けるのに適している．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．ここでは，**図 2** を用いて，図 2(c) の性能異常シグネチャを導出する過程を説明する．

（a）**図 2(a)** 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．ただし，提案手法では CDF の逆関数をプロットしている．これは横軸を割合にすることでシグネチャとなる棒グラフを見やすくする工夫である．

（b）**図 2(b)** のように，図 2(a) で得た CDF を単純化する．横軸を 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．こうすることで，次に CDF の差を計算することを可能にしている．通常，前後それぞれの CDF に含まれるリクエスト数は異なるためこの操作が必要となる．図 2(b) 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．

（c）単純化された CDF（図 2(b)）の各クラスの増大量を図 2(c) に示すシグネチャ棒グラフの各クラスの高さとする．

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り，つまりそれらの性能異常の原因が同じである可能性が高いといえる．

2.2 シグネチャの類似度計算

提案手法では 2 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 2 つのシグネチャが類似度していると判断する．**図 3** が 2 つのシグネチャを重ね合わせた例である．黒い領域の面積が大きいほど 2 つのシグネチャが類似していると判断する．

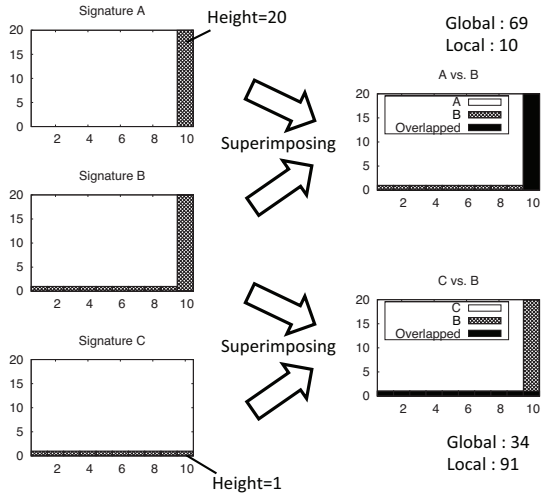


図 4 2 種類の正規化の特性　3 つのシグネチャ A，B，C を左側に，それらを重ね合わせたものを右側に記載した．

提案手法では重なる面積の計算方法を 2 種類用意した．提案手法ではシグネチャの類似度を正規化し 0 以上 100 以下としているが，正規化の方法を 2 種類用意することで，シグネチャに現れるスパイクの扱いを変えることができる．これらをそれぞれ**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同じにし，スパイクのみに左右されるのではなく，小さな変化にも注目することができる．これら 2 つを効果的に組み合わせることで性能異常の分類能力をあげることができる．

2.2.1 大域的な正規化

図 4 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．シグネチャ A と B 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．シグネチャ X と Y の類似度 $G(X,Y)$ を以下のように計算する．

Vol.2011-ARC-195 No.18
Vol.2011-OS-117 No.18
2011/4/14

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

ここで n はクラス数である（図 4 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 2 つの棒の共通部分（重なる面積）を表し，$X_i \cup Y_i$ は 2 つの棒を合併させてできる棒の面積である．図 4 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．

2.2.2 局所的な正規化

局所的な正規化を用いて計算することで，わずかずつたくさんのクラスにまたがる変化傾向の影響を大きく反映することができる．図 4 でシグネチャ B と C は 1 から 9 番目のクラスの棒の高さが 1 であり，わずかな変化がたくさんのクラスにまたがっている．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．

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

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 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．全てのクラスから受ける影響を同じにすることで，$L(B,C)$ が $L(A,B)$ に比べて大きくなり，スパイクによる影響を抑えられていることがわかる．

X_i と Y_i 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 であり同じなので，直感的には $X_i \cap Y_i / X_i \cup Y_i$ を 1 と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．2 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監視対象のシグネチャ S と性能異常が発生している監視対象のシグネチャ 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，さらに類似度が 0 以上 100 以下であるのを保つために $100/(n - \#z)$ を掛けることにした．

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．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比べて大きくゆらぎ，右側の数クラスの値を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つながる．例えば，図 4 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場合でも，$L(A,B)$ は 10 にとどまっている．

2.2.3 横方向のずれの調整

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 3 の 20 番目のクラス付近に注目してほしい．2 つのシグネチャは似ているが，重なる領域は存在しない．よって，この付近の類似度は 0 と計算されてしまう．同じ原因による性能異常だとしても，処理時間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，シグネチャを横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 2 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．本論文では，ページ数の都合上詳細な説明は省く．

2.3 類似度を基にした分類

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．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．

階層的クラスタリングでは**図 5** の左側に示した入力テーブルから右側に示した出力木が得られる．テーブル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．葉は各監視対象であり，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 2 つのクラスタを結合する．図 5 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 4 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

Vol.2011-ARC-195 No.18
Vol.2011-OS-117 No.18
2011/4/14

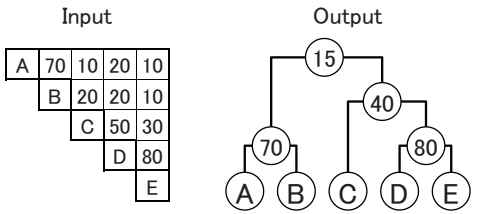


図 5 クラスタリングの入力と出力

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 1 つになるまでこの処理を繰り返す．図 5 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] となり，30 に定めれば [A，B]，[C，D，E] となる．3 節では閾値を 60 に定めて実験を行った．1 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監視対象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監視対象同士の共通部分や，クラスタに含まれない監視対象との排反部分を探すのがいくぶん容易である可能性が高い．

3. ケーススタディ

提案手法の有用性を，ケーススタディを用いて示す．それぞれのケーススタディでは既存の性能異常検出手法 1) を用いて RUBiS に発生した性能異常を検出後，提案手法を適用しリクエストの種類を分類した．その後，分類結果を考慮して異常の原因究明を行った．提案手法にある程度の変更を要するかもしれないが，1) 以外の性能異常検出手法の出力に対しても，提案手法を適用することは可能だと考えている．

1) はリクエストの種類ごとに，また処理時間の統計値ごとに性能異常を検出する．ウェブアプリケーションは処理内容によってリクエストする HTML ファイルやサーブレット，それらに送信するパラメータが異なる．これらの情報を URL を用いて確認し個別に監視する．また本ケーススタディでは平均値，最大値，中央値，最小値の 4 種類の統計値を個別に監視した．

性能異常はワークロードやサーバの設定など環境に応じて発生する．そのため，クライアント数やサーバの設定をさまざまに変えて実験を行い，その中で性能異常が発生したケースについて本節では説明する．

3.1 実験環境

RUBiS[6] はオークションサイト eBay.com[5] を模したウェブアプリケーションであり，Java EE プラットフォーム上で動作する．RUBiS はウェブ層，アプリケーション層，データベース層からなる典型的な 3 層構造によるウェブアプリケーションである．実験では各層にはそれぞれ Apache 2.2.11，JBoss 5.0.1，MySQL 5.1.34 を用いた．また，RUBiS にはクライアントエミュレータも付属しており，人間のユーザを模して HTTP リクエストを送信することができる．実験では合計 4 台のマシンを用意し，クライアントエミュレータと各層のサーバソフトウェアに 1 台ずつ割り当てた．全てのマシンは 3.00-GHz の CPU を 4 つと 2GB のメモリを搭載しており，Red Hat により提供されている Linux 2.6.27 カーネルのオペレーティングシステムが動作している．

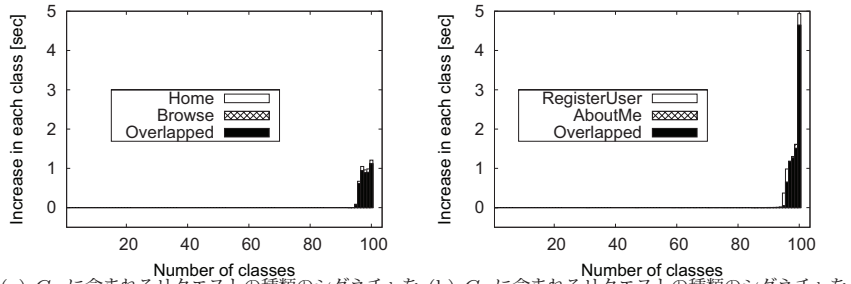
性能異常検出手法 1) は 4 種類の統計値を個別に監視することで性能異常を検出する．本実験では，性能異常を検出した統計値の種類に応じて類似度計算時の正規化を使い分ける．最大値に異常が発生した際は大域的な正規化，それ以外の統計値に異常が発生した際は局所的な正規化を用いる．2.2.2 節で述べたとおり，局所的な正規化を用いることで自然のゆらぎの影響を無視できるため，通常はこちらを用いる．それに対し，最大値に性能異常が発生した場合は自然のゆらぎを超える大きなスパイクを生み出すため，大域的な正規化を用いる．

3.2 ケース 1：異常発生サーバの特定

本小節では，提案手法が異常発生しているサーバの特定に役立ったケースを示す．本ケーススタディではサーバをデフォルト設定のままクライアント数を 200 から 300 へ増加させた．クライアント数が 300 に増加したときに性能異常を検出したため，提案手法を適用し性能異常の分類を行った．性能異常が発生したのは 27 種類中 26 種類のリクエストであった．このとき最大値に性能異常が発生したため，前述の通り大域的な正規化を用いて計算したシグネチャの類似度を用いる．

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`，

Vol.2011-ARC-195 No.18
Vol.2011-OS-117 No.18
2011/4/14



(a) 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　(b) 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 6　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

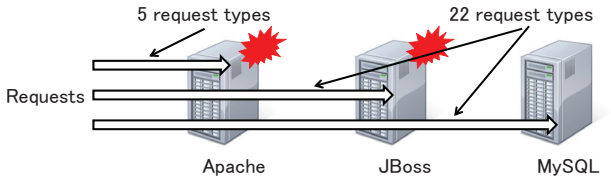


図 7　ケース 1 における原因の予想

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 6** に示した．C_b に含まれるリクエストの種類のシグネチャは 100 番目のクラスの値が C_a に含まれるリクエストの種類のシグネチャに比べて大きくなっているのが分かる．

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 7** 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 1 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．

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 200（デフォルト）から 250 に増加し，もう一度実験を行った．その実験では先ほどに比べて異常が小さくなったものの，まだ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 1 つのクラスタに分類された．図 7 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．

分類結果を受けてウェブサーバを調査した．さまざまなパラメータを変更しながら実験を繰り返したところ，`KeepAliveTimeout` を 5（デフォルト）から 2 に減少させることで性能異常から回復できた．`KeepAliveTimeout` はレスポンス終了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り，RUBiS においてはクライアント数が増加した場合は短い方が性能が向上することが分かった．以上のように，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．

3.3　ケース 2：異常発生メソッドの特定

本小節では提案手法が異常発生しているメソッドの特定に役立ったケースを示す．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 3.2 節で述べたように，RUBiS の性能を向上するために行った．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．以上の設定を行い，クライアント数を 200 から 300 に増大させ実験したところ，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が異常を発した．

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ところ，異常を発した 5 種類のリクエストの種類はそれぞれ別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 2 つのリクエストの種類のシグネチャを重ね合わせた

Vol.2011-ARC-195 No.18
Vol.2011-OS-117 No.18
2011/4/14

図はすでに図 3 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．本小節ではこのクラスタに焦点を当てて説明する．

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，ケース 1 よりも細かい粒度で調査した．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表示するメソッドである．

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するが，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で入札が可能になる．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，在庫数が n 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，在庫数が n の商品がリクエストされるべき割合は 20%であった．しかし，2 つのバグにより本実験ではこの割合が 10%から始まり徐々に 20%へと近づいていた．クライアント数の増加により，割合の増加が加速され，性能異常にいたったと考えられる．バグを修正することで性能異常から回復できた．

4. 関 連 研 究

著者達の知る限りでは，性能異常の分類に主眼を置いた研究は存在しない．本論文の目的と最も近い目的をもった研究は，現時点で発生している性能異常が過去に観測されたものであるか判定する研究[7)–9)] である．これらの手法により，現時点で発生している性能異常の種類を知ることができれば，過去に得た情報を再利用することができ，復旧に役立つ．性能異常同士が同じであるかどうかを判定するという意味で，これらは本論文の目的と類似している．しかし，本論文の目的が同時に 1 つのシステム内に発生する異常を分類するのに対し，これらの手法は発生している異常を過去に発生した異常と比較するところが異なる．

Bodík らの手法[7)] と Cohen らの手法[8)] はどちらも CPU 使用率やディスク I/O 発行量などシステムから得られる情報を基に異常の類似度を判定する．これらの指標は通常，システム全体として測定されるため，本論文のようにシステム内に同時に発生した異常を分類するのには適さない．Yuan らの手法[9)] はシステムコール列を記録し，シグネチャとして用いる．シグネチャは発行されたシステムコールの名前，引数，返値などからなる．この研究の対象は性能異常ではなく “ウェブページが表示できない” などの異常である．性能異常がシステムコール列そのものに現れるとは考えづらく，本論文の目的には適さないといえる．

処理時間に注目している研究は過去にも存在し，処理時間が性能異常分類に役立つ情報を含んでいることがわかる．Joukov ら[10)] は処理時間の分布に着目してオペレーティングシステムのプロファイリングを行っている．カーネルによって提供されている関数ごとに処理時間のヒストグラムを作成し，類似した組をユーザに出力することでプロファイリングを支援する．例えば，2 スレッドで発行した read のヒストグラムが 1 スレッドで発行した read のものと異なった分布を表した際には資源の競合が起きていることが予想される．Chen ら[3)] は性能異常の検出に CDF が有用であることを示している．Chen らの研究では 2 つの CDF が同じ分布から得られたかどうかを判定しているのに対し，本論文の提案手法では CDF の差をシグネチャとして利用している．

これまでに多くの性能異常検出手法が提案されている．著者達は，それら多く手法に対して，異常検出後に提案手法を原因究明支援のために適用することが可能だと考えている．Aguilera らの手法[11)] と Tak らの手法[12)] はどちらもマシン間のリクエストを監視し，リクエストがクラスタ内をどのように行き交っているのかを探る．この情報を利用して管理者は，処理に時間が異常にかかっているマシンやリクエストが集中しているマシンを発見することができる．Bodík ら[2)] は各ウェブページへのアクセス数を統計手法を用いて監視し，また視覚化も行うことで性能異常の検出を行う．提案手法にある程度の変更を要するかもしれないが，処理時間の代わりにアクセス数を用いて分類を行うことも原理的には可能だと考えている．さらに，Cohen ら[13)] は CPU 使用率や I/O 発行量などのシステム情報を，Xu ら[14)] はサーバによって出力されるログを用いて異常の検出を行っている．

より細かい粒度で性能異常を検出するために Chen ら[3)] はサーブレットや EJB のコンテナに，Chanda ら[4)] はサーバライブラリにそれぞれ修正を加えている．1 節でも述べたように，このように細粒度で監視することは後の原因究明に有益な情報をもたらすが，全ての監視範囲が互いに排反になるようにするには性能，実装両面のコストを考えると非現実的である．したがって，これらの手法で性能異常を検出したあとも，本論文の提案手法が必要だと考えている．

また，性能異常回避手法[15),16)] も提案されている．これらはそれぞれ各性能異常に特化しており，本論文の提案手法を用いて原因究明を行った後に，個々の原因に応じて適用可能だ

Vol.2011-ARC-195 No.18
Vol.2011-OS-117 No.18
2011/4/14

と考えている．

5. ま と め

近年，ウェブアプリケーションの性能異常が重要な問題になりつつある．さらに，システムの複雑化に伴い，性能異常の原因究明が困難になり，それを支援する手法が求められている．本論文では，同時に発生した複数の性能異常を原因ごとに分類する手法を提案した．分類結果を利用し各クラスタに共通なコンポーネントを探すことで，性能異常の発生箇所を特定することができる．

提案手法は処理時間の分布の変化をシグネチャとする．性能異常発生前後で処理時間のCDF を作成し，その差を棒グラフとして表現しシグネチャとする．その後シグネチャ同士の類似度を計算し，結果に階層的クラスタリングの群平均法を適用することで性能異常を分類する．類似度計算の際に 2 種類のアルゴリズムを導入することで，特性の異なる 2 種類の性能異常それぞれに注目することができる．

提案手法の有用性を示すために，性能異常の分類結果を用いて発生箇所を特定することができたケーススタディを示した．2 つのケーススタディではそれぞれ，分類結果が，性能異常が発生しているサーバソフトウェアとメソッドを特定する助けとなった．ケース 1 ではリクエストの種類が処理の及ぶサーバに応じて 2 つのクラスタに分類された．ケース 2 では同じクラスタに分類された 2 つ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を探し，あるメソッドが原因であることを突き止めた．

参 考 文 献

1) Iwata, S. and Kono, K.: Narrowing Down Possible Causes of Performance Anomaly in Web Applications, *Proc. of European Dependable Computing Conf.* (2010).
2) Bodík, P., Friedman, G., Biewald, L., Levine, H., Candea, G., Patel, K., Tolle, G., Hui, J., Fox, A., Jordan, M.I. and Patterson, D.: Combining Visualization and Statistical Analysis to Improve Operator Confidence and Efficiency for Failure Detection and Localization, *Proc. of IEEE Int'l Conf. on Autonomic Computing* (2005).
3) Chen, M.Y., Accardi, A., Kiciman, E., Lloyd, J., Patterson, D., Fox, A. and Brewer, E.: Path-Based Failure and Evolution Management, *Proc. of USENIX Symp. on Networked Systems Design and Implementation* (2004).
4) Chanda, A., Cox, A.L. and Zwaenepoel, W.: Whodunit: Transactional Profiling for Multi-Tier Applications, *Proc. of ACM European Conf. on Computer Systems* (2007).
5) eBay: eBay.com, (online), available from ⟨http://www.ebay.com/⟩ (accessed 2011-02-26).
6) RUBiS: RUBiS: Rice University Bidding System, (online), available from ⟨http://rubis.objectweb.org/⟩ (accessed 2011-02-26).
7) Bodík, P., Goldszmidt, M., Fox, A., Woodard, D.B. and Andersen, H.: Fingerprinting the Datacenter: Automated Classification of Performance Crises, *Proc. of ACM European Conf. on Computer Systems* (2010).
8) Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T. and Fox, A.: Capturing, Indexing, Clustering, and Retrieving System History, *Proc. of ACM Symp. on Operating Systems Principles* (2005).
9) Yuan, C., Lao, N., Wen, J.-R., Li, J., Zhang, Z., Wang, Y.-M. and Ma, W.-Y.: Automated Known Problem Diagnosis with Event Traces, *Proc. of ACM European Conf. on Computer Systems* (2006).
10) Joukov, N., Traeger, A., Iyer, R., Wright, C.P. and Zadok, E.: Operating System Profiling via Latency Analysis, *Proc. of USENIX Symp. on Operating Systems Design and Implementation* (2006).
11) Aguilera, M.K., Mogul, J.C., Wiener, J.L., Reynolds, P. and Muthitacharoen, A.: Performance Debugging for Distributed Systems of Black Boxes, *Proc. of ACM Symp. on Operating Systems Principles* (2003).
12) Tak, B.C., Tang, C., Zhang, C., Govindan, S., Urgaonkar, B. and Chang, R.N.: vPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities, *Proc. of USENIX Annual Technical Conf.* (2009).
13) Cohen, I., Goldszmidt, M., Kelly, T., Symons, J. and Chase, J.S.: Correlating instrumentation data to system states: A building block for automated diagnosis and control, *Proc. of USENIX Symp. on Operating Systems Design and Implementation* (2004).
14) Xu, W., Huang, L., Fox, A., Patterson, D. and Jordan, M.I.: Detecting Large-Scale System Problems by Mining Console Logs, *Proc. of USENIX Annual Technical Conf.* (2009).
15) Xi, B., Liu, Z., Raghavachari, M., Xia, C.H. and Zhang, L.: A Smart Hill-Climbing Algorithm for Application Server Configuration, *Proc. of Int'l World Wide Web Conf.* (2004).
16) Sugiki, A., Kono, K. and Iwasaki, H.: Tuning mechanisms for two major parameters of Apache web servers, *Software: Practice and Experience*, Vol.38, No.12, pp. 1215–1240 (2008).