特徴量抽出と変化点検出に基づくWebサーバの高集積マルチテナント方式におけるリソースの自律制御アーキテクチャ

松本 亮介[1,a)]　田平 康朗[2,b)]　山下 和彦[2,c)]　栗林 健太郎[1,d)]

概要：Web ホスティングサービスにて管理者がテナントごとのコンテンツを制御できないような高集積マルチテナント Web サーバ環境では，ホスト間のリソース競合を減らすことが安定運用にとって不可欠である．しかしホスト数が増えるにつれ，サーバ内の原因となるホストの監視や制御のコストも増加するため運用は難しくなる．本論文ではリソースの各指標の時系列データの変化点検出，ならびにサーバ内ホストやプログラムの各指標の重みづけによって，システムリソース逼迫状況下で多量のリソースを消費するリクエストを同定し隔離する自律的アーキテクチャを提案する．

Autonomous Resource Control Architecture for a Multi-Tenant Web Server by Detecting Change Points in Time Series Data of the Feature Quantities

Ryosuke Matsumoto[1,a)]　Yasuaki Tahira[2,b)]　Kazuhiko Yamashita[2,c)]　Kentaro Kuribayashi[1,d)]

Abstract: In a highly integrated multi-tenant Web server enviroment such as a Web hosting service where the server administrator has no control of the contents in each tenant, reducing resource competition between the server hosts is essential for the stable operation. As the number of the hosts increases, however, the operation becomes difficult due to the increasing cost of monitoring and control to manage the responsible hosts in the server. We propose an autonomous architecture of identifying and isolating the requests causing high system resource consumption under the resource exhaustion condition, by detecting the change points of the resource metrics as time series data, and by weighting the metrics of the hosts and programs in the server.

1. はじめに

Web サービスの低価格化とスマートフォンの普及に伴い，Web サービス利用者の数が増大している．そのような状況で，Web サービスを安定稼働させ，同時に Web サービス基盤の運用を効率化するために，Web サービスの基盤技術とシステム運用技術 [7] が注目されている．Web サーバの高集積マルチテナント方式は，単一の Web サーバに複数の利用者環境であるホストを高集積に収容することで，ハードウェアや運用のコストを低減するために利用される．しかし，高集積マルチテナント方式の一つである Web ホスティングサービスは，ホストに配置される Web コンテンツの動作を管理者が詳細に把握できないため，ホスト間でのリソース競合をあらかじめ予測することは困難である．また，原因となるホストの調査についても，高集積にホストが収容されている場合，数多くのホストが原因対象となる事が多く，かつ，その対象が時間の経過と共に変化

[1] GMO ペパボ株式会社 ペパボ研究所
Pepabo Research and Development Institute, GMO Pepabo, Inc., Tenjin, Chuo ku, Fukuoka, 810-0001, Japan
[2] GMO ペパボ株式会社 ホスティング事業部
Hosting Department, GMO Pepabo, Inc., Tenjin, Chuo ku, Fukuoka, 810-0001, Japan
a) matsumotory@pepabo.com
b) tahira@pepabo.com
c) pyama@pepabo.com
d) antipop@pepabo.com

していくため，適切な調査と対策に要するコストが著しく高くなる [6].

従来の Web サーバの高集積マルチテナント方式におけるリソース制御は，ホスト名やファイル名，接続元 IP アドレス等の同時接続数を計測し，設定された閾値を超えたらリクエストを拒否あるいは中断するような方式 [2] であった．その場合，同時接続数を超えると全く接続できなくなるため，利用者にとってはサービス停止と変わらず，サービス品質を低下させることになる．その問題を解決するために，我々は，リクエスト毎に CPU 等のコンピュータリソースの使用量を限定可能な隔離環境で，リソースは制限しサーバの負荷を低減させながらも継続的にレスポンス生成処理を行うリソース制御手法を提案した [11]．一方で，どのような状況においてどれぐらいのリソース使用量を割り当てるのが適切なのかということや，負荷原因の状況に応じて同時接続数制限との組み合わせをどう判定するかについて，刻々と変化しログの量も肥大化していく状況下で人力による調査に頼って判断することは依然高コストである．適切な制限項目や一定のルールに従った制限値を，いかにシステム管理者の運用コストをかけずに調査し制限するかという，ホスト単位で精細なリソース制御を行う際の課題が残っている．

従来の手法による問題を解決するためには，時間とともに変化する各ホストのリソースの変化を都度把握し，原因となるホストを自動的に検出した上で，その変化の特性によって制限の組み合わせを決定し，自動で制限することが効果的である．本研究では，Web サーバのコンピュータリソースの特徴量を時系列データとして抽出してリクエスト毎に変化点検出を行い，原因となるホストやプログラムの変化らしさの重み付けを行った上で，サーバ全体のリソース逼迫時には，重み付けリストの結果に基いて自律的に原因となるリクエストを同定し分離するアーキテクチャを提案する．時系列データには，ホストおよびプログラム単位でのレスポンスタイムのデータとその時点の同時接続数を使用する．この時系列データに対して，変化点検出アルゴリズム ChangeFinder[4] により変化点スコアを計算し，リクエスト時のホスト名とプログラム名にもとづいて，計測したスコアからリソースの傾向変化に寄与したホストおよびプログラムの重み付けリストを更新していく．そして，サーバ全体が高負荷状態になった場合に，重み付けリストに従って原因の可能性が高いリクエストのみを，リソース使用量が限定された隔離環境内で処理するようにする．これらを，Web サーバのレスポンス生成処理の過程に組み込むことにより，Web サーバはサーバ管理者の代わりに自律的に原因を解析し，必要な時にその原因に対処できる．

本論文の構成を述べる．2 章では Web サーバの高集積マルチテナント環境におけるリソース制御の既存手法と課題について述べる．3 章では，既存手法の課題を解決するための提案手法について，原因の自動的な検出と自動制御によって実現するリソースの自律制御アーキテクチャとその実装ついて述べる．4 章では提案手法の有効性を検証するための予備実験を行い，5 章でまとめとする．

2. リソース制御の既存手法と課題

高集積マルチテナント方式の代表的なサービスである Web ホスティングとは，複数のホストでサーバのリソースを共有し，それぞれの管理者のドメインに対して HTTP サーバ機能を提供するサービス [7] である．Web ホスティングサービスにおいて，ドメイン名 (FQDN) によって識別され，対応するコンテンツを配信する機能をホストと呼ぶ．

高集積マルチテナント方式によって構築された Web ホスティングサービスでは，ホストの Web コンテンツはサービス利用者によって自由に配置されるため，サーバ管理者はコンテンツを管理することはできない．そのため，サーバ高負荷時は，Web サーバソフトウェアや OS のプロセス管理の技術を使って制限する必要がある [1], [5].

高集積マルチテナント方式の Web ホスティングサービスに求められる要件は，低価格化を実現するためにいかに単一のサーバにホストを数多く収容 [12] し，品質を向上するためにいかに安定稼働させるかである．安定稼働のための重要な対策のひとつとして，一部のホスト間でのサーバのコンピュータリソース使用量の競合による大多数のホストの性能劣化を未然に防ぐ必要がある．

2.1 同時接続数制限の課題

Web サーバ高負荷時のリソース制御手法は，これまで，ホスト単位でのホスト名やファイル名，あるいは，接続元の IP アドレスを利用してサーバに対する同時接続数を計測し，運用管理者が設定した同時接続数の閾値を超過していた場合リクエストを拒否するような方式 [2] であった．これらの方式は，閾値超過時に接続を拒否するため，サーバ自体の高負荷状態を防止するには効果的であるが，サーバへリクエストを送信しているクライアントからすると，サービス停止と変わらず，サーバを高負荷状態から防ぐ事のみに特化した手法であった．すなわち，処理を継続しながらも，リクエスト単位でリソースを制御することのできない粗い制限手法であり，制限によってサーバを安定させたとしても，ユーザ体験の質は大きく低下する．また，CPU を 100%消費するようなプログラムへのリクエストは，同時接続数を制限したとしても，たったひとつのリクエストで CPU を占有するため，同時接続数制限では対処が難しい．

2.2 リクエスト単位のリソース制御の課題

リクエスト単位のリソース制御に関する既存手法は，CPU 等のコンピュータリソースが閾値を超えた場合に，リ

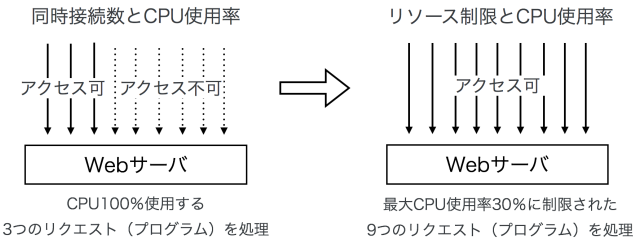

図 1 同時接続数と CPU 使用率の関係
Fig. 1 Trade-off between the number of simultaneous connections and CPU usage.

クエストを拒否あるいは中断する方式 [8] であった．この方式も 2.1 節と同様の理由によりユーザ体験の質を低下させる手法であった．これを改善するため，筆者らは，リクエスト毎に最大 CPU 使用率といったコンピュータリソース使用量を限定可能な隔離環境で，リソースは制限しつつも継続的にレスポンス生成処理を行うリソース制御手法を提案した [11]．この手法によって，サーバの負荷状態を防止するだけでなく，リソースの限定環境下では通常のレスポンス処理よりも低速ではあるが，クライアントにコンテンツを取得可能にすることができた．

図 1 に，同時接続数制限と CPU 使用率の関係，リソース制限と CPU 使用率の関係を示す．CPU コアを 3 つを搭載していた Web サーバ上で，CPU を 100%使用するプログラムに対してリクエストが送られてきた場合，同時接続数を 3 以上許可してしまうと容易に高負荷な状態になる．しかし，最大同時接続数を 3 に設定すると，他のリクエストは拒否する動作になるため，ユーザ体験の質は低い．一方で，リクエスト単位でリソースを制御する手法では，CPU を 100%消費しようとするプログラムに対して，CPU 利用率を最大 30%に制限することにより，同一の環境でも最大 9 並列でリクエストを処理できるようになる．このような制限を課すことで，モバイル環境のような低速なネットワークからのリクエストが増加していることを考慮した場合，ユーザ体験の質が向上することが見込める．

リクエスト単位でのリソース制御によって解決できない課題としては，リクエストに割り当てる CPU 使用率の最大値を設定することにより処理が低速化されたリクエストが停滞するため，Web サーバ全体の同時接続数の最大値を超えた場合はサービス停止につながるという問題がある．

2.3 原因調査と制限設定種別の取捨選択の課題

高集積マルチテナント方式の Web ホスティングサービスにおいては，サーバのリソースやサーバに対する HTTP アクセス，実行されるコンテンツが刻々と変化する状況下で，人力による負荷原因の調査に頼って判断することは高コストであるという課題が挙げられる．また，既存のサーバ監視手法では通常，サーバのリソース値があらかじめ設定した閾値を超過したらサーバ管理者にアラートを通知するような仕組みになっており，その閾値を低くすると，サーバのより詳細なリソース負荷をサーバ管理者が認識できる代わりにアラート通知の届く頻度が増加し運用コストが増大する．閾値以下であっても，サーバの負荷原因となる要因は既に存在していることが多く，そのような情報を大量のログなどから収集し，アラート時に迅速に対応することは難しい．

別のやり方として大量のログを解析することによって問題を分析し予測するような方法も考えられるが，低価格でサービスを提供することや，サーバ管理者やハードウェアの限られたコストを考慮すると新規導入は容易ではない．検知のリアルタイム性とコンピュータリソースのバランスをどう取るかといった問題もある．本手法では計算量をいかに少なくし，提供サービスの構成を大きく変更することなく Web サーバの拡張機能ソフトウェアとして導入できる方向性を検討する．

ホスティングサービスとしてコンテンツを快適に配信し続けることと，サーバを安定させるために問題となるホストを制限することを両立させるためには，単一の制限手法だけではなく，状況に合わせてホストやプログラムに対する同時接続数や各リクエストのリソース使用量の上限設定などを組み合わせて制限する必要がある．例えば，制限対象をホスト単位よりも，より粒度の細かいファイル単位にし，リクエスト単位の最大 CPU 使用量制限値と同時接続数制限値を適切に組み合わせることが必要である．しかし，高負荷状態が発生している中，迅速に原因対象の調査を行いながら，負荷をかけている原因の特徴を捉え，最適な組み合わせで制限設定を行うことは困難である．

以上を整理すると，高集積マルチテナント環境の Web ホスティング環境において，CPU の使用量といったリソースが逼迫している状況で，サーバ管理者が負荷原因を特定する基本的な作業フローは以下のようになる．

(1) サーバの高負荷状態やレスポンス遅延をアラートで検知
(2) 現時点のプロセスの CPU 使用量の状態から原因となるホストやプログラムの候補を検出
(3) レスポンスタイムやアクセス数のログの傾向から時間のかかっているホストやプログラムの候補を検出
(4) 候補の中からレスポンスタイムの傾向が大きく変化している対象を原因と判定
(5) 原因事象に対して，その特徴にあった制限設定を選択し実施

上記のようなフローで原因を特定する場合，サーバ管理者がアラートを受けて SSH でサーバにログインしてから作業を行う．高集積マルチテナント環境では，ホストの収容数が著しく多いため，各フロー項目の作業量のコストが高い．また，調査するログの量も多いため検出にも時間がかかる場合が多い．また，その後の原因の特徴を考慮した

制限の設定も困難である．

ここまで述べたことから，既存手法の課題として，以下の 2 点を挙げることができる．

(1) 高集積マルチテナント環境下での過去のログを活用した負荷原因の調査コストが高く即時性が低い
(2) 負荷原因に対する制限設定や設定の組み合わせが人の判断に依存しており対応コストが高い

高集積マルチテナント方式による Web ホスティングサービスにおいて，これらの課題を解決できる手法は未だ確立していない．

3. 提案手法

2 章で述べた課題を解決するためには，以下の 2 つを達成する必要がある．

(1) 過去のリソース使用量の傾向と特徴を逐次学習しながら高速に負荷原因を自動検出する
(2) 検出結果に基づいて負荷の特徴にもとづいた即時性の高い自動制限を行う

上記の 2 つの要件を満たすためには，サーバのリソースが逼迫していない状況でも，サーバのリソース変化に大きく寄与した原因となるホストやファイルの特徴を解析する必要がある．また，定常的にリソースを使用しているようなホストは，サーバのリソース使用量が他のホストと比較して多かったとしても，高負荷時の調査コストの観点から事前にリソース使用量が予測しやすく，経験的に突発的な高負荷の原因となることはまれであるため，それほど問題にならない．一方で，高集積マルチテナント方式における突発的な高負荷の原因とその調査コストの観点では，サーバのリソース使用量の変化が大きいホストやプログラムほど，問題になることが多いと考えられる．

本論文では，このサーバのリソース使用量の変化に注目した処理を行うアーキテクチャを提案する．提案手法では，まず Web サーバで管理している各ホストおよびプログラムのレスポンスタイムや同時接続数のようなリソースの使用量を表す特徴量を時系列データとして抽出し，時系列データの変化らしさを数値化するために変化点検出による変化点スコアを算出する．次にリソースの特徴量の変化点スコアからホストとプログラムの重み付けリストを作成する．サーバ全体のリソース逼迫時やレスポンスが遅くなった時は，重み付けリストの重み付けスコアにもとづいて自律的に原因となるリクエストをリソースの限定環境下で処理する．

3.1 負荷原因の自動検出の実装

処理のフローを**図 2** に示す．変化点検出を行う時系列データには，リクエストされたプログラムのレスポンスタイムとその時点での同一プログラムに対する同時接続数を使う．レスポンスタイムは実行されたプログラムのリソース使用量，特に CPU の使用時間と強く相関していると考えられる．プログラムによっては sleep を実行している場合もあるため，厳密にはプロセスの CPU 使用時間などを採用すべきであるが，本研究ではレスポンスタイムで十分にリソース使用量の用途に足りうると判断した．

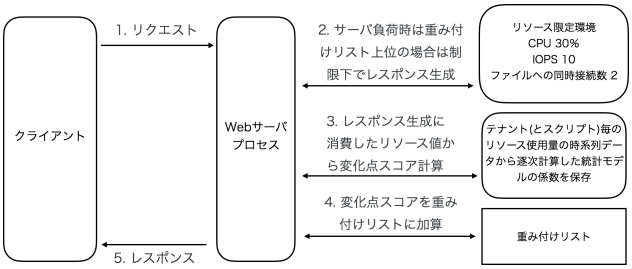

図 2 リクエスト単位での自律制御フロー
Fig. 2 Flow of autonomous control per request.

```
{
    host1: {                        # ホスト名
        st_score: 83,               # 同時接続数スコア
        rc_score: 32,               # レスポンスタイムスコア
        files: {
            path_to_progmra1: {     # プログラムファイルパス
                st_score: 30,
                rc_score: 20,
            },
            path_to_progmra2: {
                st_score: 53,
                rc_score: 12,
            },
        },
    },
}
```

図 3 重み付けリストのデータ構造
Fig. 3 Data structure of the weighted lists.

変化点スコアの計算には SDAR モデルに基づく変化点検出エンジン ChangeFinder[4] を使う．SDAR モデルは，オンラインでの解析のために計算量を少なくし，過去の統計情報の影響を次第に少なくしていく統計モデルである．ChangeFinder は外れ値と変化点を統一的に扱うために，SDAR アルゴリズムによる二段階の学習と平滑化を組み合わせて設計された非定常の時系列データにも対応しやすいアーキテクチャである．本手法では，Web サーバがリクエストの処理と並行して非定常な時系列データの変化点検出を行うため，計算量の少ないアルゴリズムを採用することで，リクエスト処理に極力影響を与えないようにしている．

ホストとプログラムの重み付けスコアの作成には，リクエストを処理する際に得られたホスト名とプログラム名を**図 3** に示すようなデータ構造で表現し，該当ホストとファイル名（図 3 中の host1 と files）が示す重み付けスコアに変化点スコアを加算していく．重み付けスコアは，ホスト

```
> cf = ChangeFinder.new 5, 0.01, 10, 0.01, 7
 => #<ChangeFinder:0x7fad5c80be50 @ts_data_buffer=[],
@change_point_analyze=#<ChangeFinder::SDAR:0x7fad5c80b
b80>, @smooth_term=5, @outlier_analyze=#<ChangeFinder:
:SDAR:0x7fad5c80be20>>

> cf.learn [1,2,1,2,3,2,1,2,1]
 => [6.2017912433901, 1.3973555597559, 2.4211198000217,
 2.3979400886673, 1.7835503570548, 1.4166612339939, 1.4
837836144657, 1.2835583707215, 1.1556254255408]

> cf.score 1
 => 1.1044914205061
```

図 4 mruby-changefinder の実行例
Fig. 4 Execution examples of mruby-changefinder.

およびプログラムに対して 2 種類用意しておき，st_score には同時接続数の変化点スコアを，rc_score にはレスポンスタイムの変化点スコアを加算する．ホストのそれぞれのスコアは，プログラムの各スコアの合計スコアを示す．この処理をリクエスト単位で繰り返し行っていくことにより，Web サーバのリソースの傾向変化に寄与したであろうホストとプログラムの重み付けリストを作成できる．

提案手法の実装には Apache[3] と筆者らが開発した mod_mruby[10] [*1]を採用した．mod_mruby は Apache の機能拡張を mruby[9] で実装でき，高速かつ少ないメモリ消費量で動作するため，本手法のようにリクエスト処理の中で複数の処理を実現する必要がある場合に有用である．また，ChangeFinder は mruby のモジュール mruby-changefinder[*2] として実装し，mod_mruby から直接実行できるようにした．mruby-changefinder の実行例を**図 4** に示す．この例では ChangeFinder.new に，変化点検出のためのパラメータであるモデルの係数の数と過去の学習結果の影響を少なくする忘却係数をそれぞれ 2 種類ずつ渡している．これは，SDAR アルゴリズムの中で二段階の学習を行っているためである．図 4 では，第 1 引数と第 2 引数に第 1 段階の係数の数と忘却係数，第 3 引数と第 4 引数に第 2 段階の係数の数と忘却係数を渡している．第 5 引数には各学習段階で得られたスコアの時系列データに対して，データの平滑化を行う時間の枠を示す値を設定する[*3]．

ChangeFinder ではモデルの係数を増やせばより細かく時系列データの特徴を抽出できるがその分計算量が増加し，忘却係数を大きくすれば過去の学習データの影響を少なくできる．また，平滑化の時間の枠を大きくすると，突発的な時系列データの変化の影響を少なくできるが，傾向変化のスコアの算出が遅延する．図 4 では learn メソッドの引数に時系列データの配列を渡すことによって，時系列データの初期状態の学習を行い，その後に score メソッドの引数に次の時点でのデータを渡すことにより変化点スコアを算出している．

変化点スコアの計算後は，図 3 のデータ構造に基づき，スコアに紐づくホスト名とプログラム名に該当する score にデータを加算する．**図 5** に，Apache が mod_mruby によってレスポンスタイムから変化点スコアを計算し，スコアをエラーログに出力する場合の mod_mruby の設定例を示す．mod_mruby の設定からは 2 つのスクリプトがフッ

*1 https://github.com/matsumotory/mod_mruby
*2 https://github.com/matsumotory/mruby-changefinder
*3 例えば，1 分毎に蓄積された時系列データに対して，10 を設定すると，10 分の枠の中でデータの平均値を計算し，時系列データを更新する．

```
# /etc/httpd/conf.d/mod_mruby.conf
LoadModule mruby_module modules/mod_mruby.so

# ChangeFinder の初期化処理をフック
mrubyPostConfigMiddle cf_init.rb cache

# 変化点スコアの計算処理をフック
mrubyLogTransactionMiddle cf_score.rb cache
```

図 5 mod_mruby の設定例
Fig. 5 Configuration examples of mod_mruby.

```
# ChangeFinder インスタンス生成
cf = ChangeFinder.new(5, 0.1, 10, 0.1, 3)

# 仮学習データによる事前学習
cf.learn [1,1,1,1,1,1,1,1,1,1]

# 各フェーズでデータを取り出せるようにユーザデータに保存
Userdata.new.cf_list = {}
Userdata.new.cf = cf
```

図 6 ChangeFinder の初期化処理スクリプト
Fig. 6 ChangeFinder initializing script.

```
r = Apache::Request.new
cf = Userdata.new.cf
cf_list = Userdata.new.cf_list hostname = r.hostname
res_time = r.response_time

# vhost の ChangeFinder インスタンスが存在しなければ複製
unless cf_list.has_key?(hostname)
   usercf[hostname] = cf.clone
end

# リクエストタイムから変化点スコアを計算しログに出力
Apache.log Apache::APLOG_ERR, “
 requesttime: #{r.response_time.to_s}
 score: #{cf_list[hostname].score(res_time)}
 hostname: #{hostname}”
```

図 7 変化点スコア計算スクリプト
Fig. 7 Change point score analysis script.

クされており，Apache が起動時に実行するフックには，**図 6** に示す Ruby コードを登録しておく．また，Apache がレスポンスを返した後に，ログ出力時に実行するフックには**図 7** に示す Ruby コードを登録する．図 6 の Ruby コードにより，ChangeFinder の初期化，事前学習，ホスト単位のインスタンスを保存するハッシュを用意する．レスポンス処理後は，図 7 の Ruby コードによって，ホスト毎に保存している ChangeFinder インスタンスから変化点スコアを計算し，エラーログに出力する．また，同様に mod_mruby で実装したファイル単位の同時接続数解析と制限ソフトウェアである http-access-limiter[*4]によって，リクエスト処理の時点での同時接続数を取得できるため，その値から変化点スコアが計算できる．

以上の手順により，Web サーバがリソース使用量の傾向変化に大きく寄与したホストとプログラムの重み付けリストを自動的に作成する．この手順に従うことで，人間にとっては簡単な作業である変化点の認識を，より細かく長期間に渡ってスコアとして自動的に保存し指標化することができる．

3.2 検出結果に基づく自動制限の実装

提案手法では，Web サーバが自律的に作成した CPU 使用量や同時接続数の変化らしさの重み付けリストを使い，Web サーバが高負荷の場合あるいはレスポンスタイムが増加した時に，重み付けリストの中で特にスコアが高いプログラムに対して自動的に制限を設定する．具体的には 3.1 節で作成した重み付けリストに基づいて，スコアが高いホストおよびファイルへのリクエストのみに対し，CPU 使用量が限定された隔離環境内でレスポンスを処理する．実装には，筆者らが開発した mod_mruby と mruby-cgroup[*5]を使用し CPU の最大使用率を，また http-access-limitter を使ってプログラム単位での同時接続数およびホスト単位でのプログラムに対する同時接続数を制限する．

判断の基準は以下の通りである．サーバの高負荷時あるいはレスポンスタイムが一定の閾値を超えた場合，リクエストのあったプログラムの st_score が特定の閾値を超えていた場合には，http-access-limiter によってそのプログラムに対する同時接続数を制限し，rc_score も閾値を超えていた場合は，mruby-cgroup により，リクエストを処理するプロセスの CPU の最大使用率が限定した状態でプログラムを実行する．制限設定後にサーバの負荷が安定し，レスポンスタイムの閾値を下回った場合は，その後一定の時間を設けた後に制限下で処理していたプログラムの制限を解除する．これらの基準を Web サーバのレスポンス生成処理の過程で Web サーバの拡張処理として組み込むことにより，Web サーバは自律的に原因を解析し，サーバの高負荷時には自動で適切な制限を行える．

[*4] https://github.com/matsumotory/http-access-limiter
[*5] https://github.com/matsumotory/mruby-cgroup

表 1 alba の実験環境
Table 1 Experimental Environment for alba.

	仕様
CPU	Intel Xeon E5-2620 v3 2.40GHz
Memory	32GBytes
Server	NEC Express5800/R120f-2E
OS	CentOS6 Linux Kernel 4.8.14
アカウント数	約 1300
ホスト数	約 10000
1 日の平均アクセス数	約 950 万

4. 実験と考察

提案手法の有効性を検証するにあたり，まずは，2.3 節で明らかにした運用フローを自動化するために，提案手法のプロトタイプ実装として筆者らが開発した自動監視・制限ツール alba をプロダクション環境に導入して評価を行った．続いて，重み付けリストの生成とリストの特徴にもとづいて適切な制限を行う処理については，プロダクションで動作させるまでには至っていないため，2.3 節で述べた運用フローによる調査結果と，提案手法にもとづいてサーバのログを解析して作った重み付けリストの結果を比較して考察を行った．

4.1 alba の評価と考察

表 1 に alba の動作検証環境を示す．本論文ではプロダクション環境で動作している複数のサーバから，alba による影響を考察しやすくするために，ホスト数やアカウント数，一日の平均アクセス数が半年間で大きく変化していないサーバを対象とした．

alba は，Web サーバとは別のプロセスで 3 分毎に実行される．alba の監視・制限のフローは以下の通りである．ただし，ここでのアカウントとは複数のホストを契約しているアカウントのことを意味し，アカウントに紐づく複数ホストの集合である．

(1) レスポンス監視用のコンテンツへの HTTP リクエストのレスポンス時間が 10 秒かロードアベレージが 60 を超える
(2) 過去 3 分間のログから以下のいずれかの計測条件から得られたアカウントに一致しているか，またはホストがサーバ全体の 10%以上の計測値を示せば (3) へ
  (a) アカウント単位で集計した処理時間の合計が最も多いアカウント
  (b) アカウント単位で集計したアクセス数の合計が最も多いアカウント
  (c) ホスト単位で集計したレスポンスタイムの合計が最も多いホスト
  (d) ホスト単位で集計したアクセス数の合計が最も多

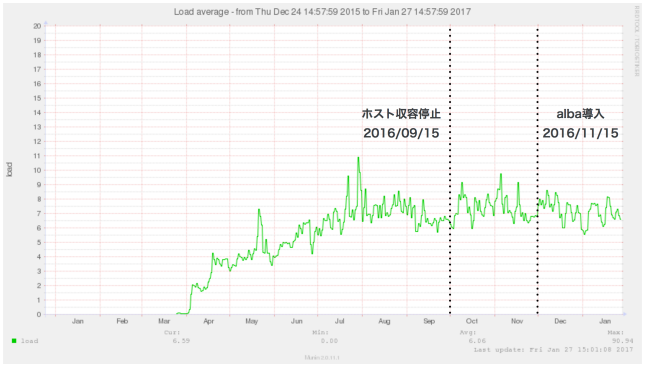

図 8 alba 導入前後のロードアベレージの遷移
Fig. 8 Transition of load average with alba.

いホスト

(3) 全体の 10%以上の処理時間またはアクセス数を占有している場合は最大同時接続数 10，全体の 25%以上のリソースを利用している場合最大同時接続数 2 に制限する（アカウントが一致した場合はアカウントが管理する全てのホストに対して制限を設定する）

(4) 制限 30 分経過後にレスポンス時間が 10 秒以下かつロードアベレージが 60 以下の場合，自動で制限解除

各パラメータは，導入した環境で，制限によって全体のアクセス数が大きく低下しないこと，Web サーバが高負荷状態にならないことを前提に予備実験を行い決定した．レスポンスタイム 10 秒としたのは，通常監視用コンテンツのレスポンスタイムは 1 秒程度であり，本来 10 秒となるとユーザ体験が大きく低下することになりかねないが，制限設定時に Apache のリロードに必要となる時間を考えるとその状況下では 3 秒の閾値では繰り返し検出してしまうため 10 秒と決定した．また，ロードアベレージについては，監視のアラートに繋がる値は導入環境において 100 程度であり，60 を超えた場合には 100 に到達してアラートとなることが経験的に確実であることから 60 と決定した．最大同時接続数の制限値については，予備実験を繰り返した結果，上記で述べた前提のもとに値を決定した．

比較のために alba の導入例を示す．2016 年 9 月 15 日にホストの新規収容を停止し，その後ホスト数が大きく変わらない状態で alba を 2016 年 11 月 15 日に導入し，約 2 ヶ月間ロードアベレージとのモニタリングを行った結果について，ロードアベレージのグラフを**図 8**，CPU のグラフを**図 9**，Apache に対する 1 秒間のアクセス数を**図 10** に示す．各図の横軸は時間を示しており，縦軸は各時点での値を示している．CPU グラフの図 9 はシステム CPU 使用率 (緑)，ユーザ CPU 使用率 (青)，idle 率 (黄) の重ね合わせを示している．この例では図 10 が示すように，全体として 1 秒間のアクセス数は概ね一定であり，図 8 や図 9 を見ると，アクセス数と負荷の変化は相関がある．alba 導入前後で傾向に大きな変化はなく，alba 導入によるサーバの

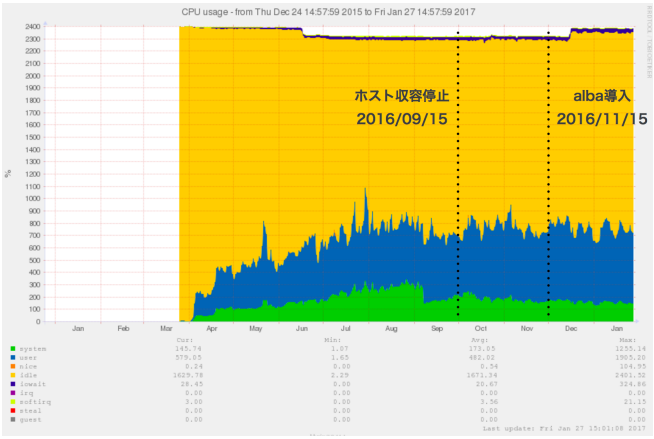

図 9 alba 導入前後の CPU 使用率の遷移
Fig. 9 Transition of CPU usage with alba.

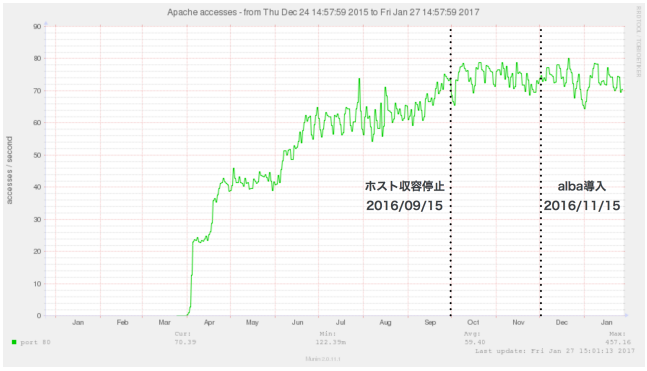

図 10 alba 導入前後の 1 秒間のアクセス数の遷移
Fig. 10 Transition of the number of access/second with alba.

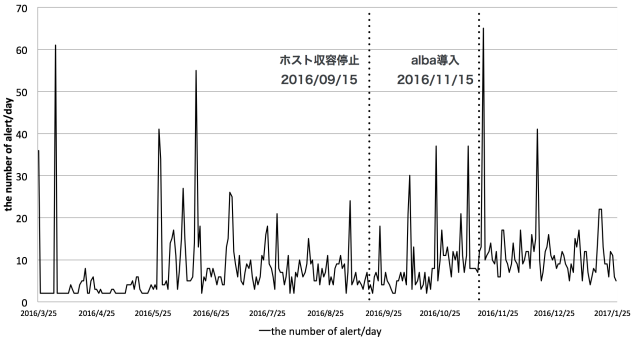

図 11 alba 導入前後の一日のアラート数の遷移
Fig. 11 Transition of the number of alert/second with alba.

負荷やアクセス数の直接的な改善は見られなかった．

alba 導入前後のサーバ管理者への 1 日毎のアラート数およびアラートの総数の遷移を図示したものを**図 11** と**図 12** に示す．alba 導入後にアラート数の削減に対する有効性を示すデータは得られなかった．この理由としては，実運用上はアラート通知を受けてサーバを調査した際は既に alba によって対応済みであり，設定反映のための Apache のリロードによってアラート通知が行われる場合が多かったことが挙げられる．現在の alba は，制限を行うために Apache のリロードが必要であることから、リロードによっ

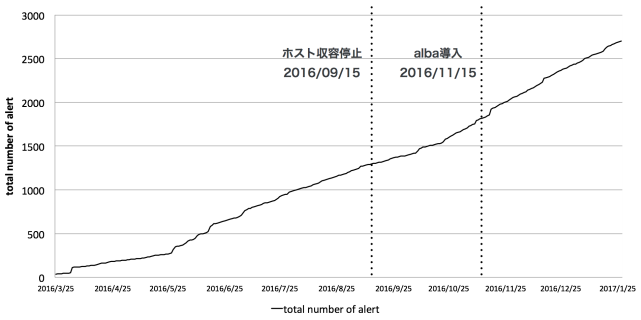

図 12 alba 導入前後のアラート合計数の遷移
Fig. 12 Transition of total number of alert/second with alba.

て発生したアラートが検知されたものと推測できる．

4.2 重み付けリストの考察

次に，alba と同一の作業フローによって，高負荷状態を検知した時刻の調査結果とその日のレスポンスタイムが記録されているログを調査した結果を，3.1 節で示したような重み付けリストを生成するためのサンプル実装によって解析し，ホスト単位の変化点スコアから重み付けリストを作成した結果との比較を行った．

その結果，日々の高負荷対応のフローでは，直近数十分のアクセス数や転送量，レスポンスタイムから原因ホストを見つけ出すことができなかったが，1 リクエスト単位のレスポンスタイムの平均値を比較したところ，問題となるホストの候補を検出でき，そのホストのみに着目すると高負荷時に大きく傾向が変化していることが分かり，負荷原因であることがわかった．一方，重み付けリストからは，該当ホストのアクセス数は少ないにもかかわらず，重み付けリスト上位に 10 位以内に位置し，問題となるホストとして検出できていた．この結果から，重み付けスコアにアクセス数の数を考慮した計算方法を考慮することでより精度が高くなると考えることができる．

5. まとめ

本論文では，高集積 Web ホスティングサービスのようにサービス事業者がホストに配置する Web コンテンツの動作をあらかじめ予測できない状況において，サーバ高負荷時の調査や対応を Web サーバが自律的に行う手法を提案した．そして，提案手法の有効性を示すために，まずは，2.3 節で述べた運用フローを実装した alba の評価を行い，その有効性を考察した．

今後の課題としては，提案手法の重み付けリスト機能や特徴を考慮した自動制限機能を定期実行の alba をベースに mod_mruby でリクエスト契機の処理に実装しなおした上で，プロダクション環境で動いている alba と置き換えを行い，有効性がどの程度あるかを評価する必要がある．また，アラートが通知されたとしても，その後の対応がツールによって自動化されており対応が不必要であった場合や，アラートの受信から対応完了までの時間を評価の指標に加える必要がある．これらの評価を行うことで，サーバのリソースが逼迫していない状況でも，サーバのリソース変化に大きく寄与した原因となるホストやファイルの特徴を解析する必要性を示すことができる．また，これらの評価によって、3 章で述べた本論文のアイディアの核である「負荷原因の調査コストが大きくなる原因としてはサーバのリソース使用量の大小よりも，ソース使用量の変化の大小が高集積マルチテナント方式における突発的な高負荷の原因とその調査コストの観点から問題になることが多い」という主張の有効性を示すことができる．

参考文献

[1] Che J, Shi C, Yu Y, Lin W, A Synthetical Performance Evaluation of Openvz, Xen and KVM, IEEE Asia Pacific Services Computing Conference (APSCC), pp. 587-594, December 2010.
[2] David J, mod_limitipconn, `http://dominia.org/djao/limitipconn.html`.
[3] Fielding R T, Kaiser G, The Apache HTTP Server Project, IEEE Internet Computing, Vol. 1, No. 4, pp. 88-90, 1997.
[4] J Takeuchi, K Yamanishi, “A Unifying Framework for Detecting Outliers and Change Points from Time Series,” IEEE transactions on Knowledge and Data Engineering, pp.482-492, 2006.
[5] Kovri A, Dukan P, KVM & OpenVZ virtualization Based IaaS Open Source Cloud Virtualization Platforms: OpenNode, Proxmox VE, IEEE 10th Jubilee International Symposium on Intelligent Systems and Informatics, pp. 335-339, September 2012.
[6] Mietzner R, Metzger A, Leymann F, Pohl K, Variability Modeling to Support Customization and Deployment of Multi-tenant-aware Software as a Service Applications, the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems, pp. 18-25, May 2009.
[7] Prodan R, Ostermann S, A Survey and Taxonomy of Infrastructure as a Service and Web Hosting Cloud Providers,10th IEEE/ACM International Conference on Grid Computing, pp. 17-25, October 2009.
[8] The Apache Software Foundation, Server Wide Configuration Limiting Resource Usage, `http://httpd.apache.org/docs/2.2/en/server-wide.html#resource`.
[9] NPO 法人軽量 Ruby フォーラム, `http://forum.mruby.org/`.
[10] 松本亮介, 岡部寿男, mod_mruby：スクリプト言語で高速かつ省メモリに拡張可能な Web サーバの機能拡張支援機構 , 情報処理学会論文誌, Vol.55, No.11, pp.2451-2460, 2014 年 11 月
[11] 松本亮介, 岡部寿男, リクエスト単位で仮想的にコンピュータリソースを分離する Web サーバのリソース制御アーキテクチャ, 情報処理学会研究報告 Vol.2013-IOT-23, No.4, 2013 年 9 月.
[12] 松本亮介, 川原将司, 松岡輝夫, 大規模共有型 Web バーチャルホスティング基盤のセキュリティと運用技術の改善, 情報処理学会論文誌, Vol.54, No.3, pp.1077-1086, 2013 年 3 月.