Webサーバの高集積マルチテナントアーキテクチャに関する研究

Studies on Highly Integrated Multi-Tenant Architecture for Web Servers

松本　亮介

2017年3月

概要

Webサービスの普及とスマートフォンのように手軽にWebサービスを利用できる個人端末の爆発的普及により，運用コストを低減し安価にサービスを提供するために，単一のWebサーバ上に複数のホストを高集積に同居させる高集積マルチテナントアーキテクチャが利用される．高集積マルチテナントアーキテクチャは，複数のユーザを同居させる特性上，ユーザ単位でセキュリティを担保するために，多くの場合ソフトウェアによって適切な権限分離を実現する必要がある．

従来のマルチテナントアーキテクチャは，ホストに配置されるWebコンテンツを事業者が管理できない場合，セキュリティを担保するために性能を大幅に犠牲にする必要がある．また，運用コストを低減するためにマルチテナントアーキテクチャを採用しても，収容するホスト数が増えるにつれ，適切にリソース分離ができないという問題がある．サーバのリソースを沢山使用しているユーザが大量に存在した場合も，高負荷の原因を調査するのが難しく，かえって運用コストの増大につながっている．

本論文では，Webサーバの高集積マルチテナントアーキテクチャにおいて，Webコンテンツをサービス事業者が管理できない状況における，高品質でありながらハードウェアや運用管理コストを低減させるための最適なアーキテクチャについて述べる．本アーキテクチャは，セキュリティと性能の両立や，リソースを超過して使用しているホストの迅速な検知と制限および適切なホスト単位でのリソース制御を行うことよって運用コストを低減し，さらに，Webサーバの処理を容易に各Webサービス事業者がカスタマイズできるように，性能，メモリ使用，セキュリティの課題を解決したWebサーバの機能拡張を実現することを主目的とする．本目的を達成するために，Webサーバのマルチテナントアーキテクチャの課題を関連研究にもとづいて，運用面，セキュリティ，リソース制御の観点からまとめた上で，高集積Webホスティング基盤のセキュリティと運用技術の改善のためのアーキテクチャ，スレッド単位で高速に権限分離を行うWebサーバのアクセス制御アーキテクチャ，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アーキテクチャ，リクエスト単位で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，の4つのアーキテクチャを提案する．その結果，Webホスティングサービスをはじめ，多くのマルチテナントアーキテクチャを採用するWebサービスの品質が向上し，低価格化も実現されるようになり，Webサービスの普及と利便性向上に貢献できる．

目次

図一覧

表一覧

1 緒論

1.1 研究の目的

Webサービスの普及とスマートフォンのように手軽にWebサービスを利用できる個人端末の爆発的普及により，安定したWebサービスを構築するための基盤技術とシステム運用技術が注目されている[53]．単一のWebサーバ上に複数のホストを高集積に同居させることで，大幅にハードウェアコストや運用コストを低減し，Webサービスを低価格化することを目的とした高集積マルチテナントアーキテクチャが重要性を増している[42]．

高集積マルチテナントアーキテクチャでは，複数のユーザを同居させる特性上，ユーザ単位でセキュリティを担保するために，多くの場合ソフトウェアによって適切な権限分離を実現する必要がある．近年のWebサービスに関するセキュリティインシデントの増大に伴い，高集積マルチテナントアーキテクチャのためのソフトウェア技術への注目度が増している．しかし，従来のWebサーバのマルチテナントアーキテクチャは，ホストに配置されるWebコンテンツを事業者が管理できない場合，セキュリティを担保するために性能を大幅に犠牲にする必要があった[24][25][44]．また，運用コストを低減するためにマルチテナントアーキテクチャを採用しても，収容するホスト数が増えるにつれ，適切にリソース分離ができないという問題がある．サーバのリソースを沢山使用しているユーザが大量に存在した場合も，高負荷の原因を調査するのが難しく，かえって運用コストの増大につながっている[42]．このような問題を解決するために各事業者は様々な取り組みを行っているが，汎用的かつ実践的なソフトウェアとして実現できておらず，依然として従来の手法に頼っている．

本研究では，Webサービスをより低コストで実現するための高集積マルチテナントアーキテクチャにおいて，セキュリティと性能の両立や，リソースを超過して使用しているホストの迅速な検知と制限および適切なホスト単位でのリソース制御を行うことよって運用コストを低減し，さらに，Webサーバの処理を容易に各Webサービス事業者がカスタマイズできるように，性能，メモリ使用，セキュリティの課題を解決したWebサーバの機能拡張支援アーキテクチャを実現することを主目的とする．その結果，Webホスティングサービスをはじめ，多くのマルチテナントアーキテクチャを採用するWebサービスの品質が向上し，低価格化も実現されるようになり，Webサービスの普及と利便性向上に貢献できる．

1.2 研究の位置づけ

代表的なマルチテナントアーキテクチャのサービスであるWebホスティングサービスは，低価格化と高集積化が進んでいる[53]．高集積マルチテナントアーキテクチャは単一のハードウェア上にできるだけ高集積にユーザ環境を構築することによって，環境構築のコストを低減し，低価格化を実現する．Webホスティングサービスは，各ユーザ領域に利用者が任意のWebコンテンツを置くことを許すため，管理者はWebコンテンツに依存しないOSやミドルウェアといった基盤技術のみで，マルチテナント環境におけるセキュリティや安定性を担保しなければならない[44]．Webサーバのマルチテナントアーキテクチャが持つこのような条件を考えると，事業者がWebコンテンツを管理できない場合の課題を解決することで，コンテンツに依存しない汎用性の高いマルチテナントアーキテクチャを実現できる．

これまで，ハードウェアリソースを必要最小限に抑えるためには，アクセスのあったホスト名でコンテンツを区別し，単一のプロセスで複数のホストを処理する方式である仮想ホストを用いる方法が一般的であった．しかし，高集積であればあるほど，運用と性能の安定化は困難で，運用技術の改善と性能劣化の少ないセキュリティ機構の設計，さらに，基盤を構築するための汎用的の高いWebサーバの機能拡張支援機構の実現が課題であった．

以下に，高集積マルチテナントアーキテクチャのWebホスティングにおけるセキュリティと性能の課題，Webサーバの機能拡張に関する開発コストや運用上の課題，リソース制御の課題をまとめる．

まず第1に，マルチテナントアーキテクチャにおけるセキュリティと性能の課題について述べる．高集積マルチテナント環境において，ホスト単位で権限を分離するためには，代表的なWebサーバソフトウェアであるApacheを用いる場合，suEXEC[75]と呼ばれるアクセス制御手法を利用している．suEXECのアーキテクチャは，リクエスト単位で権限分離用のプロセスを生成し，そのプロセスでレスポンスを処理した後に，プロセスを破棄する．しかし，このような従来のアクセス制御アーキテクチャは，権限分離のためにWebサーバへのリクエスト毎にプロセスの生成，破棄や比較的大きなバイナリの実行が複数回必要となり，性能が低い．suEXECの性能の問題を解決するために，リクエスト単位でプロセスの生成・破棄を行わないmod_suidやmod_ruid[27]といったアーキテクチャが提案されたが，これは性能が大きく向上する一方で，権限分離の特権を保持したままWebコンテンツを実行するため，セキュリティレベルが大きく低下する．そのため，セキュリティレベルをsuEXECと同等にまで向上させようとすると，結局suEXECと同様のアーキテクチャを使う必要があり，セキュリティと性能の両立は達成できていない．さらに，動的コンテンツを実現す

るプログラミング言語のインタプリタ自身に独自の権限分離機構が実装されていたり，プログラム実行方式それぞれに専用のアクセス制御手法を用意していたりと，インタプリタや実行方式に応じて手法を選択する必要があり，サーバ管理者にとって扱いにくいものとなる．

第2に，Webサーバの機能拡張に関する開発コストや運用上の課題を述べる．運用面や安定性を考慮した高集積マルチテナントアーキテクチャを構築する際に，Webサーバソフトウェアの機能拡張が必要になる場合が多い．Webサーバの機能拡張において，高速かつ軽量に動作することを重視した場合，C言語による実装が主流であった．一方で，生産性や保守性を考慮した場合は，スクリプト言語で機能拡張を行う手法も提供されている[37]．しかし，従来手法は，Webアプリケーションの実装ではなくWebサーバの内部処理を拡張することを主目的とした場合，性能，メモリ使用，セキュリティの面で課題が残る[77]．また，Ruby on RailsやPHPの普及に伴い，Webアプリケーションがスクリプト言語によって実装される機会が増え，Webサービスを構築する技術者がC言語よりもスクリプト言語を得意としていることが多くなってきた．このような状況を考えると，Webサーバの機能拡張を効率よくスクリプト言語で実現する汎用的なソフトウェアの実現が急務である．

第3に，リソース制御における課題について述べる．高集積マルチテナント環境を実現するWebサーバソフトウェアにおいて，CPUリソースやDisk I/Oを多く消費するリクエストがあった場合，その処理を適切にホスト単位で制御する必要がある．従来のWebサーバのリソース制御は，リソース制御そのもののオーバーヘッドを低減した方式[62]はあるものの，基本的にはプロセス単位の粒度での制限[3][65]であったり，リクエストの同時接続数や単位時間当たりのリクエスト数[12]あるいは一リクエストのCPU使用時間やメモリ使用量が，管理者があらかじめ設定した閾値を超えた場合に，リクエストを強制的に切断あるいは拒否するようなアーキテクチャ[73]になっている．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が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．現在のWebサービスの普及やHTTP/2プロトコル[39]がWebサーバのプロトコルとして採用される状況を考えると，今後はユーザ体験の質を極力低下させないことを目指して，リクエスト単位でOSのリソース制御と連携して行うアーキテクチャの必要性が増すと考える．

以上より，本研究の位置づけとして，Webコンテンツに依存せずOSやミドルウェアといった基盤技術での改善が必要となるWebホスティングサービスに着目し，高集積マルチテナントアーキテクチャにおける課題について，以下4つの提案により解決することを目標とする．

(1) 高集積マルチテナントアーキテクチャによるWebホスティング基盤のセキュリティと運用技術の改善
(2) スレッド単位で高速に権限分離を行うWebサーバのアクセス制御アーキテクチャ
(3) 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アーキテクチャ
(4) リクエスト単位で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

さらに，アーキテクチャの実装は，オープンソースソフトウェアとして公開し，多くのWebホスティングサービス企業が容易に適用可能な状態にする．

アーキテクチャの実装における本研究の目標について述べる．現在のWebサービスにおいて，OSにはLinuxを採用することが一般的であり，その多くの場合に，性能や信頼性および保守性や公開情報を考慮して，ミドルウェアにオープンソースソフトウェアのApache httpdやnginxが使われている．そのため，新しく研究開発されたアーキテクチャは，広く使われているオープンソースソフトウェアの機能拡張モジュールとして実現することにより，導入障壁を下げることが望まれる．仮に独自の実装で実現したとしても，運用・保守のコストが大幅に増大したり，これまでのノウハウや実績を活かせなくなるため，各企業で導入には至らない場合がほとんどである．そこで，本研究でも，実装においては，新しいアーキテクチャを普及させるためにも，既存の広く使われているソフトウェアの機能拡張モジュールとして実装を行うという制約条件を課した上で，アーキテクチャの実現を目指す．

本研究では，歴史あるWebサービスであるWebホスティングサービスの汎用性に着目し，周辺のシステムやミドルウェアの課題を体系的にまとめ，その課題を新しいアーキテクチャによって解決した上で，既存の運用技術を考慮した企業が採用しやすい実装を行う．それによって，Webサービスにおけるシステムやミドルウェアのセキュリティや性能および運用技術の向上に寄与することを学術的貢献とする．

1.3 論文の構成

本論文の構成を述べる．

第2章では，高集積マルチテナントアーキテクチャおよびそれを実現したWebサービスであるWebホスティングサービスのためのアーキテクチャの従来手法や基本概念，

用語を整理し，Webサーバのマルチテナントアーキテクチャの課題を関連研究に基いて，運用面，セキュリティ，リソース制御の観点からまとめる．

第3章では，代表的なマルチテナント環境サービスで，特に基盤技術による制御が必要不可欠な高集積マルチテナントアーキテクチャを採用したWebホスティングサービスに対して，仮想ホスト採用と大規模対応にともなって生じる運用面とセキュリティ上の課題を明確化し，リソース制限を実現するApacheモジュールの開発と従来のアクセス制御手法であるsuEXECの改善によって，それらを解決する手法を提案する．本手法によって，信頼性と運用性の高い大規模Webホスティング基盤を構築できる．

第4章では，Linuxのスレッドの拡張機能であるスレッド単位での権限分離機能を活用し，Webコンテンツ処理時にサーバプロセス上で新規スレッドを生成し，スレッドで権限分離を行った上で，スレッドがコンテンツの処理を行うアクセス制御アーキテクチャのmod_process_securityを提案する．このアーキテクチャは，動的コンテンツの実行方式やインタプリタによらない統一的なアクセス制御であり，サーバ管理者が扱いやすく，性能劣化も少ない．広く使われているLinux上で動作するApacheのモジュールとして実装した．

第5章では，スクリプト言語で安全に機能拡張でき，かつ，高速，省メモリに動作するWebサーバの機能拡張支援機構mod_mruby，ならびに，ngx_mrubyを提案する．本論文では提案手法として，Webサーバプロセスから内部処理としてスクリプトが呼び出された際，高速に処理するためにインタプリタの状態を保存する状態遷移保存領域をサーバプロセス起動時に生成しておいて，それを複数のスクリプトで共有して実行するアーキテクチャをとった．また，使用メモリ量の増加を低減し，かつ，状態遷移保存領域を共有することにより生じるスクリプト間の干渉を防止して安全に機能拡張するために，スクリプト実行後に状態遷移保存領域からメモリ増加の原因となるバイトコードおよび任意のグローバル変数・例外フラグを解放するようにした．このアーキテクチャの実装には，組み込みスクリプト言語mrubyと代表的なWebサーバソフトウェアであるApacheとnginxを採用し，Rubyスクリプトによって容易にApacheとnginx内部の機能拡張を行えるようにした．

第6章では，同一のサーバプロセス上で，リクエスト単位でコンピュータリソースの分離が可能なリソース制御アーキテクチャを提案する．このアーキテクチャによって，各リクエスト処理には一時的にCPUやDISK I/Oなどの仮想的なリソースが割り当てられ，そのリソースの範囲内で処理が行われる．そのため，Webサーバに対する各リクエストが，特定のコンテンツに対する突発的なアクセス集中や特定のリクエスト処理のリソース使用超過による影響を受けにくくなる．さらに，最大同時接続数

の範囲内で，大量にリソースを消費するようなリクエストであっても，リクエストを拒否・切断することなく，割り当てたリソースの範囲内で処理を継続させることが可能となる．

第7章では，本研究の貢献をまとめると共に，今後の課題や展望についても述べる．

2. 基本概念と関連研究

2.1 緒言

近年，インターネットの普及に伴い，自社でインフラ設備をもつことができない企業や個人は，Webやメールの機能をレンタルサーバと呼ばれるホスティングサービス[54]を介して利用する機会が増大している．その結果，ホスティングサービス企業の間では価格競争が起き，AmazonEC2[84][80][1]に代表されるようなクラウド[6]の台頭と共に，ホスティングサービスの低価格化が進んでいる．

これまで，筆者が所属していたファーストサーバ株式会社では，2009年にApache HTTP Server[16]のVirtualHost機能[70]を利用することで，メモリ4GB程度のサーバ1台で約2000ホスト，3サーバで約6000ホストの高集積型Webホスティング基盤の構築は達成できている．VirtualHostとは，Webサーバのマルチテナントアーキテクチャの実装の一つで，単一のWebサーバソフトウェア上で，複数のホスト領域を仮想的に処理することで，ホストの高集積化を実現する機能である．しかし，その収容数での運用コストは非常に高く，また，性能劣化の少ないセキュリティ機能を実装することは困難であった．

上記で実現したWebホスティングシステムのWebサーバの運用性[26]において，VirtualHostは単一のサーバプロセスで複数の仮想ホストを処理しているため，特定の仮想ホストがアクセス集中等によって高負荷になった場合，すべての仮想ホストが影響を受ける．ただし，ここでいう単一のサーバプロセスとは，ホスト毎にサーバプロセスを起動させるわけではなく，複数のホストでサーバプロセスを共有することを示す．実際にサーバプロセスの処理を行うプロセスは，ホスト数には依存しないものの数百存在する．さらに，仮想ホストの数が膨大になってくると，仮想ホストの設定数も増加し，設定を読み込むために必要なメモリ量も非常に多くなるため，設定変更によるプロセスの再読み込み処理が遅くなり，サービスの停止時間が増加する．

セキュリティにおいても，動的コンテンツを仮想ホスト単位で権限分離するためには，サーバプロセスをroot権限で起動し，コンテンツ実行時にコンテンツの権限にプロセスを変更してから実行し，実行後はサーバプロセスを破棄する必要があった．この構成は，例えばApacheのサーバプロセスそのものの脆弱性を突く攻撃を受けた場合において，それが任意のコマンドを実行できる脆弱性である場合，攻撃者がroot権限でコマンドを実行できるため非常に危険性が高い．また，サーバプロセ

スをコンテンツ処理毎に生成，破棄しなければならないという点で，アクセス集中に非常に弱い構成であった．

実際の運用においては，収容数よりも運用面やセキュリティを重視した場合，OS上でchroot()システムコール[1]や仮想マシン等を利用して，複数の隔離した領域を構築し，各種サービスを領域内部に閉じ込め，稼動させる構成が使われてきた．しかし，高集積化を想定した場合，収容ホスト数に比例してプロセス数が非常に多くなるためリソース効率が低い．また，ユーザ専用のサーバプロセスとハードウェアが紐づくため，複数のWebサーバで共有ストレージ上のコンテンツを共有し，どのWebサーバにアクセスがあっても同一のユーザへのホストへのレスポンスとするような負荷分散が難しく，システムをスケールさせることが困難である．このように，大規模かつ高集積に対応したWebホスティング基盤を構築するにあたって，いかに性能を劣化させずにセキュリティを担保し運用者の負担を低減するかは大きな課題となっていた．

今後，クラウドサービスや低価格ホスティング等が主流になっていくことを考えると，限られたリソースで最大限のセキュリティと運用性を担保し，汎用性のある大規模対応基盤を構築しなければならない．本章ではこの前提を満たすために，複数のホストを単一のサーバプロセスで扱い，高集積時においてもプロセス数がホスト数に依存しないように，リソースを必要最小限に抑えることのできるVirtualHostのような高集積マルチテナントアーキテクチャについて論ずることとする．

VirtualHostのような高集積マルチテナントアーキテクチャを採用した場合には，運用技術やセキュリティおよび性能上の課題が生じる．本章では，汎用性のある高集積マルチテナントアーキテクチャを採用した大規模Webホスティング基盤を実現するにあたり，Webサーバのアーキテクチャにもとづいて[15]，高集積マルチテナントアーキテクチャを採用することで生じる，セキュリティや運用面の課題およびWebサーバの機能拡張支援機構についての従来研究を体系的にまとめる．従来研究をまとめるにあたって，これまで広く研究され，高集積マルチテナントアーキテクチャの実サービスとしても多くの企業で運用されてきたApacheが採用しているアーキテクチャに着目しながら体系的に整理する．

以下，2.2節ではLinuxにおけるプロセスと権限について整理し，2.3節では組み込みスクリプト言語mrubyについて述べる．2.4節ではWebサーバのアーキテクチャやWebホスティングシステムの構成について整理し，2.5節ではマルチテナントアーキテクチャの運用面の課題，2.6節ではマルチテナントアーキテクチャのセキュリティの課

[1] https://linuxjm.osdn.jp/html/LDP_man-pages/man2/chroot.2.html

題，2.7節ではマルチテナントアーキテクチャにおけるリソース制御の課題についてそれぞれ述べる．2.8節で基本概念と関連研究の総括をする．

2.2 Linuxにおけるプロセスと権限

2.2.1 UNIXのfork()およびexecve()システムコール

UNIX系OSおよびSUSv3[32]を満たすLinuxでは，fork()システムコールによって呼び出し元プロセスを複製し，新しいプロセスを子プロセスとして生成できる．新規作成された子プロセスは，呼び出し元プロセスである親プロセスのスタックやデータ，ヒープ，テキストセグメントを複製して持つ．fork()システムコールで生成された子プロセスは，親プロセスのメモリに影響を与えることなく新規でメモリを確保し操作することができる．fork()システムコールは，指定されたプログラムを実行するexecve()システムコールとセットで使う場合が多く，execve()システムコールは，新しいプログラムを実行するプロセスのメモリ空間にロードする．その際には，実行前のプロセスのテキストセグメントは破棄され，スタックやデータ，ヒープセグメントは新しく確保される．fork()システムコールで生成された子プロセスは，親プロセスのメモリに影響を与えることなく新規でメモリを確保，操作することができる．

Webサーバが動的コンテンツを生成するためのプログラムを実行する仕組みであるCGIは，これらのシステムコールを使って実現されている．Webサーバへリクエストがあると，Webサーバプロセスは新たなCGI実行用のプロセスをfork()システムコールによって起動し，そのプロセス上で実行すべきプログラムをexecve()システムコールによって実行する．

プログラム実行時に，プロセスよりもメモリ領域を多く共有する処理方式にスレッドがある．プロセスは，親プロセスからfork()システムコールによって生成された際にメモリ領域を新たに確保するため，実行に時間がかかる．一方で，スレッドはpthread_create()ライブラリ関数などによって生成され，生成したプロセスと同一のメモリ空間上で実行されるため，メモリ領域をコピーする処理が省略され生成処理が速い．

2.2.2 Linuxの権限モデル

Webサーバ上でプログラムを実行する際に，CGI実行時に利用できるアクセス制御アーキテクチャについて解説する．このアーキテクチャは，クライアントからCGIプログラムにアクセスがあった場合，Webサーバによって一旦fork()システムコールに

よって子プロセスを生成した後に，一般ユーザであってもファイルに設定されたroot権限で実行できるようにアクセス権が設定されたラッパープログラムをexecve()システムコールによって起動させて，一旦root権限を付与する．その後，アクセスのあったCGIプログラムのオーナの権限に降格するようにsetuid()，および，setgid()システムコールを実行してから，execve()システムコールによりCGIプログラムを実行する．このように権限変更を行ってからプログラムを実行することで，CGIプログラムによる他ユーザ権限の領域へのアクセスやプログラムの閲覧および実行を防止できる．また，脆弱性を突かれたプログラム経由の被害も同一の権限内に収めることができる．例えばApacheは，suEXECというソフトウェアでこのアーキテクチャを実現している．

このWebサーバのアクセス制御アーキテクチャは，SUSv3を満たすLinuxのようなUNIX系OSの伝統的なroot権限と一般ユーザ権限という2階層の権限モデルに基いている．一般ユーザ権限は，ユーザIDが異なれば，異なるユーザIDのプロセスを制御することは基本的にできない．また，一般ユーザ権限は自身のユーザIDを別ユーザIDに変更することもできない．そのため，一般ユーザ権限においては，ユーザIDが異なれば権限分離できていることになる．一度プロセスを一般ユーザ権限で動作させてしまえば，プロセスを破棄するまでは同一の一般ユーザ権限で動作し続けることを保証できる．

Linuxには，Linuxカーネルのバージョン2.2以降（現在のすべての要件を満たす機能を持ったのは2.6.24以降）に独自のセキュリティ機構として，Linux Capabilitiesという権限モデル[2]がある．Linux Capabilitiesは，root権限が持つ特権を細分化して，スレッド単位で特権を付与することができるモデルである．例えば，前述の通り一般ユーザ権限では自身のユーザIDを他のユーザIDに変更することはできない．この変更を可能とするのはroot権限だけである．Linux Capabilitiesは，このユーザIDを変更する特権をCAP_SETUIDとして表現している．CAP_SETUIDを一般ユーザで動作しているプロセスに与えておけば，一般ユーザであっても別のユーザIDに変更することが可能になる．一方で，root権限が持つその他の様々な特権は扱うことはできない．この基本概念は，第3章，第4章の提案手法と深く関連する．

2.2.3 Linuxのプロセスリソース管理

Linuxには，cgroups[86]と呼ばれるプロセスのリソース管理技術がある．cgroupsは，2006年9月から開発が開始され，2008年1月にLinuxカーネルのバージョン2.6.24に取り込まれた．cgroupsは，プロセスの優先度を変更するniceのような機能から，コンテナであるLXCやOpenVZのようなOSレベルの仮想化までの，様々な仮想化の用途

に対応するための統一されたインターフェイスを持っている．cgroupsは，以下の5つの機能を提供している．

(1) リソース制限
(2) 優先順位
(3) 説明
(4) 隔離
(5) コントロール

(1)はプロセスグループのメモリ使用量やファイルシステムキャッシュを制限する機能を提供する．この機能により，プロセス単位でメモリ消費量の上限を制限することが可能となる．制限値を超えた場合は，プロセス停止処理が動作する．(2)はCPUやトラフィック，I/Oを制御する機能を提供する．この機能により，プロセスグループで利用可能なCPU使用率の割合を最大10%程度にしたり，トラフィックの流量を最大10Mbpsに制限したり，デバイスI/OのByte/secやIOPSの最大値を任意の値に制御できる[10]．(3)はプロセスグループのリソース消費の統計値を計測する機能を提供する．例えば，各プロセスグループのリソース消費量を可視化したり，従量課金の基準を定義することが容易になる．(4)は異なる名前空間にプロセスグループを分離し隔離する機能を提供する[56]．(5)はプロセスグループをサスペンドしたりリストアしたりする機能を提供する．この機能を利用することで，再起動無しでのカーネルの置き換え，コンテナやプロセスレベルでのサスペンド・レジューム機能，コンテナやプロセスのライブマイグレーションが可能となる[7]．

cgroupsの機能を利用することで，サーバ管理者はシステムリソースの割当，優先順位付，モニタリング等，粒度の細かいコントロールが容易に可能となる．第6章の提案手法では，cgroupsの機能を用いる．

2.3 組み込みスクリプト言語mruby

Webサーバソフトウェアの機能拡張において，生産性や保守性を意識しながら高品質かつ短納期に実装する，という課題は，組み込みソフトウェア開発現場における課題と類似している点がある．組み込みソフトウェア開発現場の課題を解決するために開発されている組み込みスクリプト言語mruby[47]がある．mrubyは組み込み用の処理系であり，通常のRuby処理系より少ないメモリで動作する．また，機能的にはRubyのサブセットであり，ISO/IEC30170規格に沿って実装されている．

2.3.1 mrubyの概要

mrubyは，まつもとゆきひろ氏を中心に開発され，2012年4月20日にソースコードが公開された組み込みスクリプト言語である．組み込みスクリプト言語mrubyをC/C++で実装されたホストアプリケーションに組み込むと，そのホストアプリケーションの一部をRubyで実装することが可能となる．以前であれば，C言語での実装が前提となっていたような組み込み機器上でも，mrubyをホストアプリケーションに組み込むことで，Rubyによる記述が可能となる．C/C++アプリケーションが主とすると，mrubyは従の関係にあり，C/C++アプリケーションの専用のパーツとしての使い方が想定されている．スクリプト言語と相性の良いテキスト処理等の実装はRubyで実装し，高速性や省メモリ等，緻密さが優先される箇所はC言語で実装する，といった使い方ができる．これによって，大規模，複雑化した組み込みソフトウェア開発をC言語のみで実装する場合，高品質，短納期，高い保守性を実現するのは困難であったが，mrubyは今後その課題を解決するための1つの手法になっていくと考えられる．

2.3.2 mrubyの設計方針

組み込みソフトウェアのためのRubyであるmrubyは，以下の特徴を持つ．

(1) 組み込みAPI
(2) 省メモリ
(3) モジュールやクラスが取り外し可能
(4) JIS/ISOを尊重
(5) 移植性
(6) リアルタイム性の向上

(1)の組み込みAPIに関して，プログラミング言語Rubyは，1つのプロセス上に1つの仮想マシンしか動作させることはできない．しかし，mrubyでは，組み込みソフトウェアで実装されるような，1つのプロセスに複数の仮想マシンを持ち，それぞれの仮想マシン上でRubyスクリプトを呼び出せる設計になっている．

mrubyは文字列の処理やメモリの自動確保のための組み込みAPI充実しており，C言語で実装されたホストプログラム上でもRubyスクリプト実行までの各種処理過程で，任意のタイミングで構文解析をしたり，バイトコードを生成したり，様々な最適化が容易になる．

(2), (3)に関して，省メモリを達成するために，mrubyでは，Rubyのソースコードを仮想マシンが解析してバイトコードにコンパイルする機能を予め取り外すことができる．開発中は，コンパイラ及びコード生成機能をリンクしておき，開発完了後は，Rubyのソースコードからあらかじめバイトコードを生成しておくことで，コンパイラやコード生成機能を不要にすることができる．通常，複雑なプログラムでなければ，mrubyのプログラムサイズは500Kバイトから1000Kバイト程度であり，コンパイル機能を取り外せば，そこからさらに100Kバイト程度のプログラムサイズを削減できる．さらに，mubyの標準ライブラリからも，実行に不必要なモジュールやクラスを取り外すことができる．例えば，標準出力が不要なプログラムの場合は，標準出力のクラスを取り外したり，配列や連想配列等一部の機能を使わない場合は関連するクラスを取り外したりすることで，さらにメモリを節約することができる．

(4), (5)に関して，組み込み領域では標準規格が重視されるため，mrubyはRubyのISO規格[29]を尊重して設計されている．また，mrubyはC言語で実装されており，移植性を高めるために1999年に制定されたC言語の国際基準であるC99に準拠した記述にしている．mrubyはファイルシステムやさらにはOSがないような環境でも動作するように実装されている．

(6)に関して，組み込みソフトウェアではリアルタイム性が重視される．リアルタイム性とは，処理時間があらかじめ指定された一定時間内に収まることが保証されていることを意味している．従来のRuby処理系は人間に検知できる程度の長時間ガベージコレクションで停止することがあり，リアルタイム用途への障害となっていたが，mrubyはインクリメンタルGCの採用でリアルタイム性を向上させている．

以上のように，mrubyが開発された理由として，組み込みソフトウェアの規模と役割の増大に伴い，開発手法に対して，高品質，短納期，高い保守性が求められていることを挙げた．ハードウェアの性能が向上してきいるため，Webサービスにおいては，高速性や省メモリを優先したC言語による開発よりも，生産性や保守性を優先した開発が重要になってきている．また，Webサービスの大規模・複雑化に伴うWebサービス開発の高品質・短納期・高い保守性の実現への要求は，組み込み機器のために開発されたmrubyの背景と類似している．

2.3.3 mrubyとLua

C言語で実装されたApacheの内部処理としてRubyやPerl等のリッチなスクリプト言語を使うには課題がある一方で，軽量組み込みスクリプト言語であるLua[55]が人気を高めてきた．Luaはリオデジャネイロ・カトリカ大学の情報工学科コンピュータグラフィックステクノロジーグループTeCGrafらによって設計開発された組み込みスク

リプト言語である．LuaはC/C++で実装されたホストプログラムに組み込み，ホストプログラムの一部の処理をLuaスクリプトで実装することを目的に設計されており，高速な動作と高い移植性，組み込みの容易さが特徴[30]である．移植性を高めるため，一旦バイトコードにコンパイルされ，Lua VM上で実行する方式をとっている．

組み込みスクリプト言語Luaは，ネットワークルータの機能拡張やゲーム上で動くキャラクターの振る舞いの実装に活用され，最近ではiPhoneアプリの開発にも利用されてきている．しかし，Rubyと比べてデフォルトで使用できるライブラリが非常に少ない．mrubyではそのような問題を解決するために，これまでに多くの技術者によって改善がなされてきたRubyの言語仕様やライブラリやRubyのオブジェクト指向の記述方法を考慮して，Luaよりもライブラリを充実させることに努めており，Luaほどではないにしても，低いマシンスペックで動作するように実装されている．mrubyが動作した機器として，32bitのARM core，クロック周波数84MHz，SRAM96KB，プログラム格納用フラッシュメモリ512KBのArduino Dueがある．

2.4 WebサーバとWebホスティングシステム

2.4.1 Apacheのアーキテクチャ

WebサーバソフトウェアであるApacheは，世界で最もシェアの高いソフトウェア（2013年7月時点）[49]である．Apacheは，リクエストを処理するための子プロセスをサーバプロセス起動時に複数起動させておき，1リクエストに対して1つの子プロセスを専有してレスポンス生成を行うモデルを基本とする．そのため，このモデルの場合，Apacheへの同時接続数の上限は子プロセスの起動数に依存する．

Apacheで高集積マルチテナントを実現する場合，収容ホスト数にサーバプロセス数が依存しないようにするための機能として，VirtualHostがある．VirtualHostは，単一のサーバプロセスで複数のホストに対するリクエストを処理する．そのため，ホストの収容数を増加させたとしても，子プロセスの数を増やす必要がないため，コンピュータリソースを効率よく利用することができる．

VirtualHostの設定は，ホスト単位でsuEXECによる権限分離を行う場合，ホスト単位で一意の設定が必要になる．そのため，VirtualHostで管理するホスト数が増えるにつれ設定数は増加し，Apache起動時のメモリ使用量も増加する．

Apacheのサーバ機能の拡張には，Apacheモジュールというプラグイン機構[71]を用いる．ApacheはC言語で実装されているため，通常インストールにはコンパイルが必要となる．Apacheモジュールのプラグイン機構を使うと，Apacheモジュールが提供しているAPIに基いて拡張機能をC言語で実装することにより，Apacheそのものを

再コンパイルすることなく，拡張機能のみをモジュールとしてコンパイルしてApache本体に組み込み機能を拡張できる．通常，Apacheモジュールはmod_“モジュール名”のような命名規則で名付ける．

Apacheは最低限のWebサーバ機能をコアとして持ち，その他の機能はコアを改修せず追加できる．図2.1に，Apacheのコアとモジュールの概要図を示す．

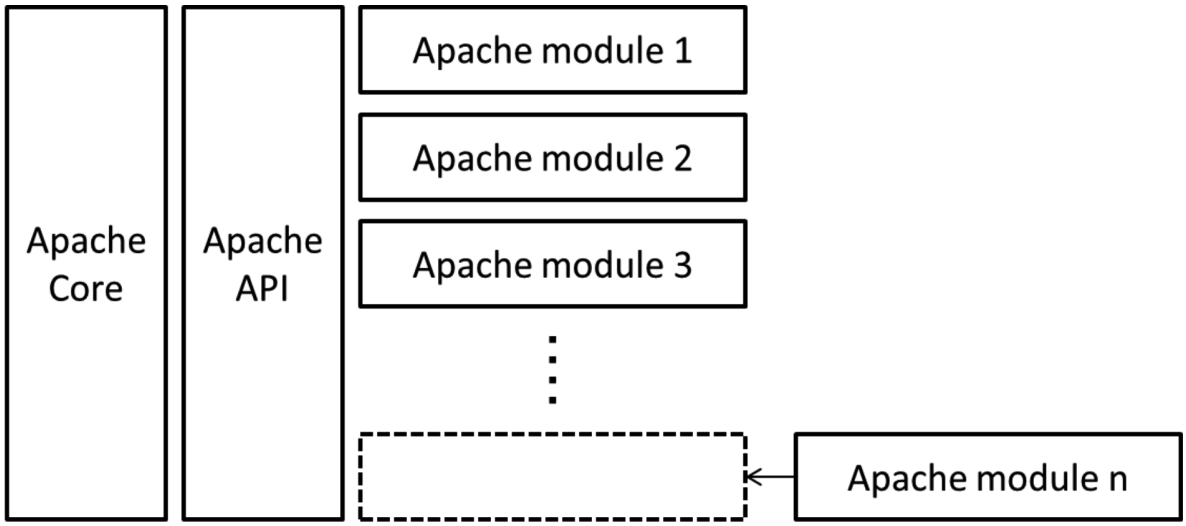

図 2.1　Apacheモジュールの仕組み

ApacheのコアとApacheモジュールの連携は，Apache独自のAPIを介して実現されている．モジュールは，コアに近い実装からWebアプリケーションに近い実装まで，様々な領域の実装が可能となっている．一般的なWebサーバソフトウェアと同様，Apacheモジュールによる機能拡張は高速性と省メモリを考慮してC言語で実装する仕様になっている．

2.4.2 Webホスティングシステム

高集積マルチテナントアーキテクチャ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である．Webホスティングサービス [53]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．

サービス提供側がWebホスティングシステムを構築する従来手法は，主に以下の4種類に分類される．

(1) XenやVMware等の仮想マシンでホストを分ける手法[79][40]
(2) chroot()システムコールでファイルシステムを隔離してOS上に複数の仮想的な隔離環境を用意しホストを分ける手法[11]
(3) IPアドレスやポート単位でWebコンテンツが配置された複数のホストを分離し各ホストに個別のプロセスを用意して起動させる手法
(4) 単一のサーバプロセス群で複数のホストを扱う手法

サーバの運用面やセキュリティを重視した場合は，手法(1)(2)(3)等の，管理者それぞれに対して，個別のサーバプロセスや仮想マシンを割り当てる構成がとられてきた．例えば，OSのシステム領域に近い環境を，特定のディレクトリ配下に作り，IPアドレスを1台のサーバに複数設定した上で，リクエストを受けたIPアドレスに応じてそのディレクトリ内へchroot()システムコールによりルートディレクトリを移動する．そのchroot環境を複数用意し，それぞれで個別のIPアドレスで通信できるWebサーバプロセスを起動する．図2.2で，運用面とセキュリティを重視した場合に，(2)の手法を採用した構成の概要図を示す．

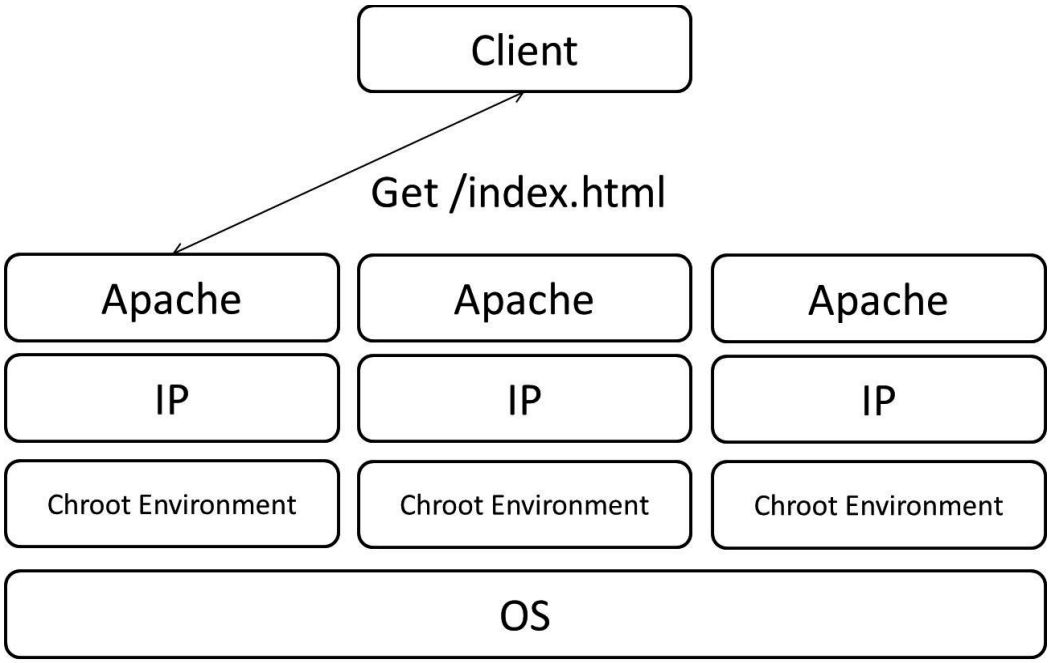

図 2.2　chroot()システムコールを活用したホスト分離

chroot環境内部からOSのシステム領域に到達することはできないため，セキュリティ面で堅牢である．

また，chroot環境はOSのシステム領域とほぼ同等のライブラリ群を任意に配置できる．サーバプロセスの設定を全てホスト専用に設定することができ，運用面でも可用性が高い．さらに，不必要なコマンドやライブラリを配置しないことで，セキュリティを高めることもできる．

JaveServlet[31]のように個別のJVMを複数用意して，各JVMのプロセス単位で複数のホストを収容する方式は(3)に該当する．同様に，Ruby on Railsも(3)の方式であり，Webサーバ機能と一体となってアプリケーションサーバとして動作するため，複数ホストを収容するためにはその数だけアプリケーションサーバプロセスを用意する必要がある．従来の研究として，(3)の手法を利用して複数のサーバプロセスをそれぞれ異なるユーザ権限で起動する手法がある[64]．各chroot環境などで，ホスティング利用者単位でサーバプロセスを起動させると，そのサーバプロセスを複数のサーバへのスケールアウト[14]することが困難になる．例えば，大規模なWebホスティング基盤においてロードバランサを使い，6台の物理サーバで12000のホスティング用ホストを処理することを想定する．その場合，コンテンツは共有ストレージを利用し，複数の物理サーバそれぞれを同じ構成にして，コンテンツの処理を負荷分散することが期待される．その構成によって，利用者が増えた場合や処理性能が劣化した場合は，さらに同一のサーバを増やすことで容易にスケールアウト型のリソース追加ができる．しかし，ホスティング利用者専用のサーバプロセスや仮想マシンを起動していた場合，共有ストレージによる負荷分散のためには，6つのサーバを同じ設定にする必要がある．つまり，同様にそれぞれ12000のユーザプロセスや仮想マシンを起動させる必要があり，リソース効率が大幅に低下する．また，12000のホストを6台のサーバに均等に分け，2000のユーザ専用のプロセスとしてサーバに収容すると，ユーザ専用のプロセスは特定のサーバに固定されるため負荷分散はできない．その結果，リソースの偏りが生じ，収容サーバそのもののハードウェア増強といった対応しかできず，大幅に運用性が低下する．(1)(2)(3)の手法ではこのような問題が生じる．

一方，(4)の手法は単一のサーバプロセスで複数のホストを仮想的に処理する構成の場合，　VirtualHostと呼ばれるマルチテナントアーキテクチャを用いることで，スケールアウト型の負荷分散の課題を解決できる．図2.3でその構成を示す．

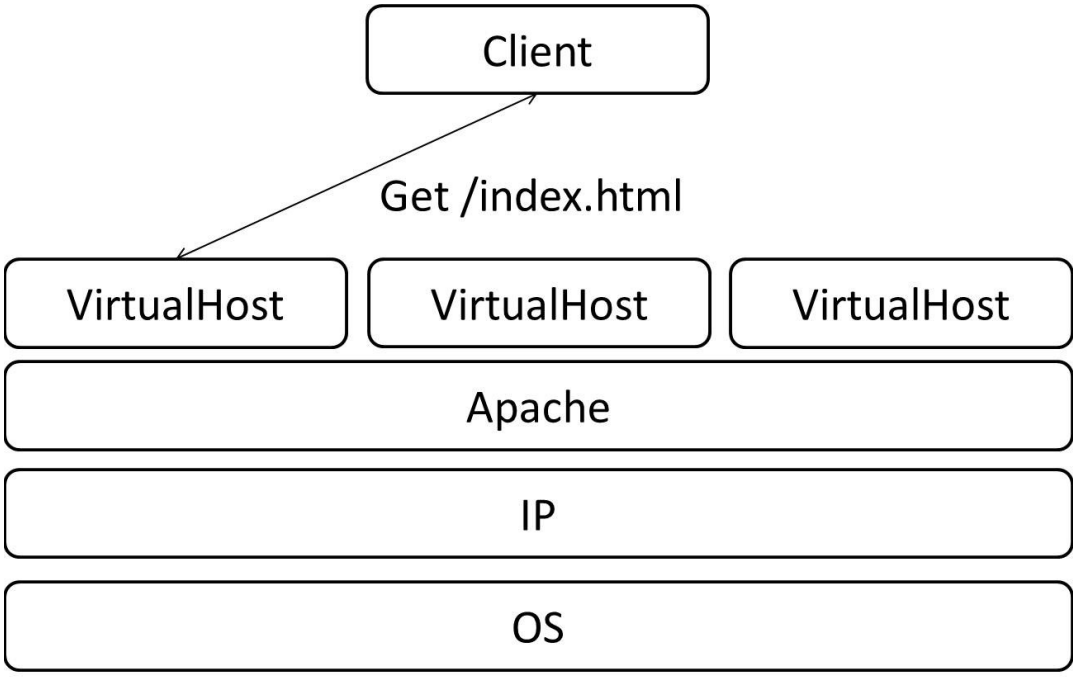

図 2.3　単一プロセスで複数ホストを扱う構成

VirtualHostでは，アクセスのあったホスト名に対応したドキュメントルートにアクセスするようにWebサーバ内部で制御する．そのため，複数のホストに対して単一のサーバプロセスが起動していればよい．以上の特徴から，複数台のVirtualHostの設定がされたサーバを用意しておけば，いずれかのサーバのWebサーバプロセスにアクセスがあれば，共有ストレージを介して，適切なレスポンスを返すことができる．同時に，各サーバのサーバプロセス数は収容するホストの数に依存しないため，効率良くリソースを扱える．その結果，複数台の物理サーバを容易に同一の環境にでき，リソースも最小限に抑えることができるため，共有ストレージを用いた効率の良い負荷分散構成も構築できる．

しかし，大規模・高集積な共有型Webホスティングを想定した場合では，マルチテナントアーキテクチャであるVirtualHostを採用することで，運用面とセキュリティの課題が生じる．

2.4.3 Webサーバ機能の拡張

本節では，Webサーバの機能拡張機構における従来研究についてまとめる．これまで，Webサーバの機能拡張は，高速かつ軽量に動作することを重視して，C言語のよる実装が主流であった[61][82][45]が，生産性や保守性を考慮してスクリプト言語で機能拡張を行う手法も提供されている[41][63][17][78]．しかし，従来手法はWeb

サーバの機能拡張だけではなく，CGIよりも高速にWebアプリケーションを実行する用途として開発されているため[8]，純粋にWebサーバの機能拡張を目的とした場合に，高速性・省メモリ・安全性の面で課題が残る．そこで，代表的なWebサーバソフトウェアであるApacheを例に，従来手法の特徴および問題点について言及する．

これまで，Webサーバソフトウェアの機能拡張を支援するために，Apacheにおいて，スクリプト言語のPerlやRubyで，Apacheモジュール相当の実装を可能とするmod_perl[36]やmod_ruby[60]およびmod_python[20]が開発されてきた．これらの従来手法は，高速に処理するために，インタプリタを複数のスクリプトで共有するが，アプリケーションとして全く別々のスクリプト間でもグローバル変数名が干渉し合い，安全に実装できないという問題があった．また，そもそもPerlやRubyのインタプリタやライブラリが巨大であるため，mod_php[51]のようにWebアプリケーションの実装を主目的として，ライブラリが充実していることが要求される場合には適しているが，高速かつ省メモリにWebサーバそのものの機能拡張を主目的とした場合には適していない．

機能拡張が高速に動作することが重視される場合は，一般的にC言語で実装される．例えば，Apacheが標準で提供している機能の中で，プロキシ機能を実現するmod_proxy[2]やリクエストのあったURLを書き換える処理を実現するmod_rewrite[3]などはC言語で実装されている．しかし，ApacheモジュールをC言語で実装すると，修正ごとにコンパイルが必要であり，また，Apacheに組み込むためにApacheサーバプロセスの再起動が必要となるので，生産性や保守性の面でリクエスト毎にコンパイルが実行されるスクリプト言語に劣る．Webサービスの大規模・複雑化に伴い，スクリプト言語による開発が主流となっている中，開発者が高品質，短納期，さらに高い保守性を求められている状況において，C言語による拡張機能の実装は次第に困難になってきている．

2.5 マルチテナントアーキテクチャの運用面の課題

ホスティングシステムを提供する側が，サーバを運用管理する工数は，インターネットの普及に伴い，日々増加してきている．2.4.2節で述べた通り，高集積マルチテナントアーキテクチャにおいては，ApacheのVirtualHost機能を使うことにより効率的にホストの収容数を増やすことができる．一方で，収容数が増えることにより運用面の課題が生じる．

---

[2] https://httpd.apache.org/docs/2.4/en/mod/mod_proxy.html

[3] https://httpd.apache.org/docs/2.4/en/mod/mod_rewrite.html

2.5.1 高負荷ホストの特定

Webサーバは，静的なファイルの配信だけでなく，負荷のかかるCommon Gateway Interface(CGI)[69]プログラムなどの動的コンテンツが実行される．マルチテナントアーキテクチャにおいて，特定のホストがリソースを占有することで，他のホストが影響を受ける状況はできるだけ回避したい．そのため，サーバ高負荷時の迅速な原因の特定と対処は非常に重要である．

ホスト単位で専用のサーバプロセスを起動するようなマルチテナントアーキテクチャの場合は，リソースを占有しているホストが特定できれば，最悪の場合，専用のサーバプロセスさえ停止させれば他のホストへの影響を緩和できる．一方，VirtualHost方式では，単一のサーバプロセスで複数のホストを処理するために，リソースを多く占有しているホストやファイルを厳密に特定して，問題となるリクエストのみを制限する必要がある．

Webサーバにおける動的コンテンツの実行方式は，Webサーバプロセスにインタプリタを組み込み，Webサーバプロセス内部で直接プログラムを実行するDynamic Shared Object(DSO)[72]実行方式と，新たに別のプロセスをfork()システムコールにより生成して，そのプロセスでexecve()システムコールによりプログラムを実行するCGI実行方式がある．

DSO実行方式で実行されたプログラムは，psコマンドによりプロセス情報を取得すると，プロセス名はプログラムファイル名ではなくサーバプロセス名であるhttpdとなる．そのため，サーバプロセスがリソースを大量に消費していた場合は，それがどのホストのどのスクリプトによるものであるかを迅速に特定することが困難である．

CGI実行方式で実行されたプログラムは，CGIバイナリの引数としてプログラムファイル名が渡され，プロセス名としてはCGIバイナリ名が表示されるため，高負荷時等，迅速に対応しなければならない状況において，該当の原因となるプログラムファイル名を正確に特定できない．以上より，高負荷状況になった場合に，どのスクリプトがどの程度のリソースを消費しているかを迅速かつ適切に調査できる仕組みが必要である．

2.5.2 VirtualHost単位でのリソースの制限

chroot環境や仮想マシンでWebサーバを起動している場合は，ホスト毎にサーバプロセスが起動しているため，リソースの制限に関してサーバが持つすべての設定を利用することができる．しかし，VirtualHostを利用した構成の場合，Apacheでは仕

様上VirtualHost単位で設定できる項目は限られている．例えば，高負荷ホストへの最大同時接続数を設定するために非常に重要なMaxClientsをVirtualHost単位で設定することはできない．また，VirtualHost上でコンテンツ単位の同時接続数を柔軟に設定する方法がないのも高負荷ホストの制限を困難にしている．

VirtualHostは複数のホストを単一のサーバプロセスで管理しているため，サーバプロセスが高負荷で停止してしまうと全てのホストの機能が停止する．そのため，サーバプロセスが高負荷でサービス停止しないように制限することが必要となる．NASやSANによるストレージの統合化に伴い，CPU使用，特にファイルシステムとのI/Oが大量に発生することによるCPUのI/Owaitによって高負荷になる場合が多い．例えば，Apacheが標準で備えるリソースの制限のための設定としては，メモリやプロセス数等があるが，CPU使用の制限の設定は限られており，RlimitCPUという設定のみである．RlimitCPUでは，クライアントからリクエストを受けてレスポンスを生成するまでに，設定したCPU使用時間を超過した場合，カーネルによって処理が強制的に切断される．そのため，ミドルウェアやWebコンテンツの実装によらず中断され，信頼性が低くなる．以上より，サーバ自体の負荷状況に合わせて，クライアントからリクエストを受けた際に，レスポンスを返すための処理を継続するかを判断する仕組みが必要だと考えられる．

2.5.3 ホストの新規設定・追加に伴うコスト

2.5.2節で述べたとおり，VirtualHost方式においてはWebサーバプロセスが停止すると，全てのホスト機能が一時的に停止する．そのため，大規模化に伴う設定の増加を考慮すると，設定に追加あるいは変更があってもできる限りWebサーバプロセスのリロードやリスタートを実施しないようにするべきである．新規ホストの追加設定やチューニングを行う場合に，Webサーバのリロードを実施しないでよいようにするためには，mod_vhost_alias[68]を用いる手法がある．mod_vhost_aliasはApacheモジュールで実装されており，Dynamically Configured Mass Virtual Hosting（DCMVH）という設定記述方法を提供する．通常，Apacheでは，ホスト追加時に新規ホスト用のVirtualHost設定を追記する．しかし，VirtualHostの設定はホスト名やドキュメントルート名等が異なるだけで，その他の設定は同じ場合が多い．そこで，DCMVHの設定記述法を利用すると，ドキュメントルートにホスト名を含んだパスになるようにディレクトリを作成しておけば，VirtualHostの設定においてパスのホスト名部分を変数で記述することができる．その記述によって，VirtualHostの設定を1つ書いておけば，アクセスのあったホスト名で設定を動的に読み替え，該当のドキュメントルートにアクセスできるようになる．また，設定の数がホストの数に依存し

ないため設定読み込みの負荷も少なくできる．しかし，DCMVH固有の問題として，VirtualHost毎に個別の設定が必要であるsuEXECのようなアクセス制御モジュールの設定をDCMVHでは動的に扱うことができない問題や，VirtualHost毎に環境変数に保存されるドキュメントルートが変数を読み替えたパスにならない問題がある．

以上が，マルチテナントアーキテクチャであるVirtualHostを用いて，大規模対応のWebホスティング基盤を構築する際に生じる運用面の課題である．

2.6 マルチテナントアーキテクチャのセキュリティの課題

Apacheは，歴史的にマルチテナントアーキテクチャのために様々な運用上の課題を解決し，改善した上で運用可能なレベルで機能追加を行ってきていることから，以下では，Apacheを例に具体的な課題をまとめることで，従来技術の現実的なセキュリティおよび実運用上の課題を整理することができると考える．

2.6.1 システム領域や他ホスト領域の覗き見

ApacheのVirtualHostを採用した構成では，一般にOSのシステム領域でWebサーバプロセスを起動するため，Webサーバプロセスのユーザ権限や，動的コンテンツが実行される際のユーザ権限であっても，システム上の一般ユーザ権限で閲覧可能な/etc等のシステム領域のファイルを覗き見することができる．また，閲覧できないようにすべてのシステム領域の権限を修正するコストは非常に高い．

VirtualHostで動作しているApacheは，サーバプロセス権限で全てのリクエストを処理する必要があり，コンテンツファイルやディレクトリをサーバプロセス権限で操作可能にしなければならない．そのため，単純なマルチテナントアーキテクチャでは，異なるユーザが管理する他ホスト領域を覗き見することができてしまう．図2.4に，他ホスト領域のファイルを覗き見するための一般的な仕組みと権限設定を示す．図2.4では，index.cgiはWebサーバプロセスの権限であるuid500，gid101で実行され，/var/www/hosts/host1.example.com/ディレクトリはgid101からの読み取り権限がある．さらに，ディレクトリ配下のホスト領域内部のファイル群はWebサーバプロセス権限でアクセスできるように全ユーザに読み取り権限を与えている．そのため，host1を管理するユーザは，/var/www/hosts/host2.example.com/ ディレクトリの下のコンテンツには直接アクセスできないが，host1のindex.cgi内でシェル等の外部コマンドを実行することで，host2のindex.cgiのソースコードなどを閲覧できてしまう．また，CGIプログラムを経由せずとも，シンボリックリンクを他ホスト領域のファイルに対して別名で設置するだけで，Webサーバプロセスを介した覗き見が

可能となる．これを防ぐために，CGIプログラム実行時に利用できるsuEXEC[75]のようなアクセス制御モジュールを用いて，コンテンツの権限でCGIプログラムを実行し，適切に各ホスト領域の権限を設定することで，覗き見できないようにする方法が用いられる[43]．同時に，Apacheにおいてシンボリックリンク経由で他ホスト領域へ辿れないようにする設定も利用される．

2.6.2 suEXEC採用に伴うCGI実行方式の課題

ApachesuEXEC　機能を用いると，VirtualHostを採用していても他ホスト領域を閲覧できなくする構成をとることができる．図2.5にsuEXECの利用例を示す．図2.5では，図2.4と同様のパーミッション設定をしている．suEXECを採用すると，クライアントからCGIプログラムにアクセスがあった場合，ApacheによってCGIプログラムの実行処理をsuEXECに依頼する．suEXECはindex.cgiを実行する際に，index.cgiの権限であるuid501，gid102を取得する．

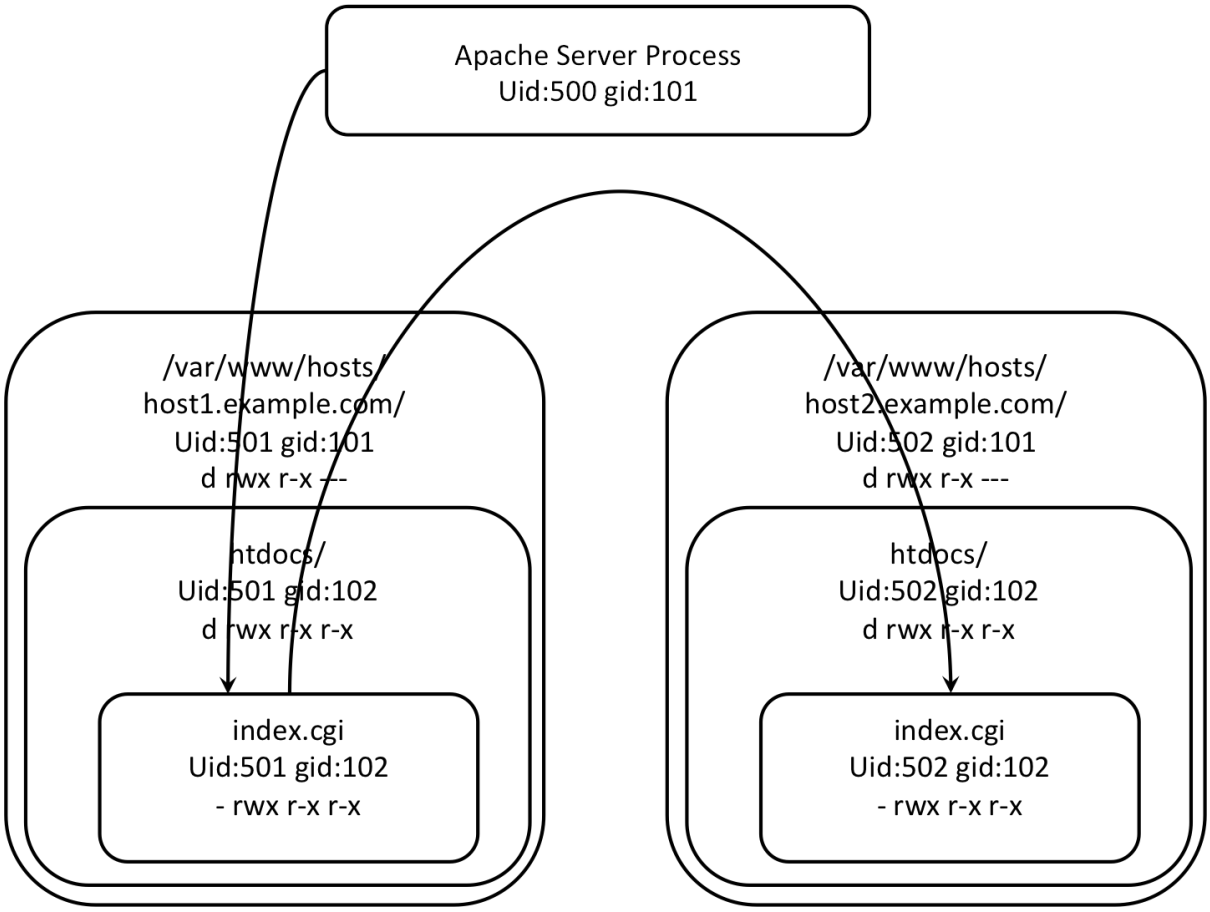

図 2.4　他ホスト領域の覗き見

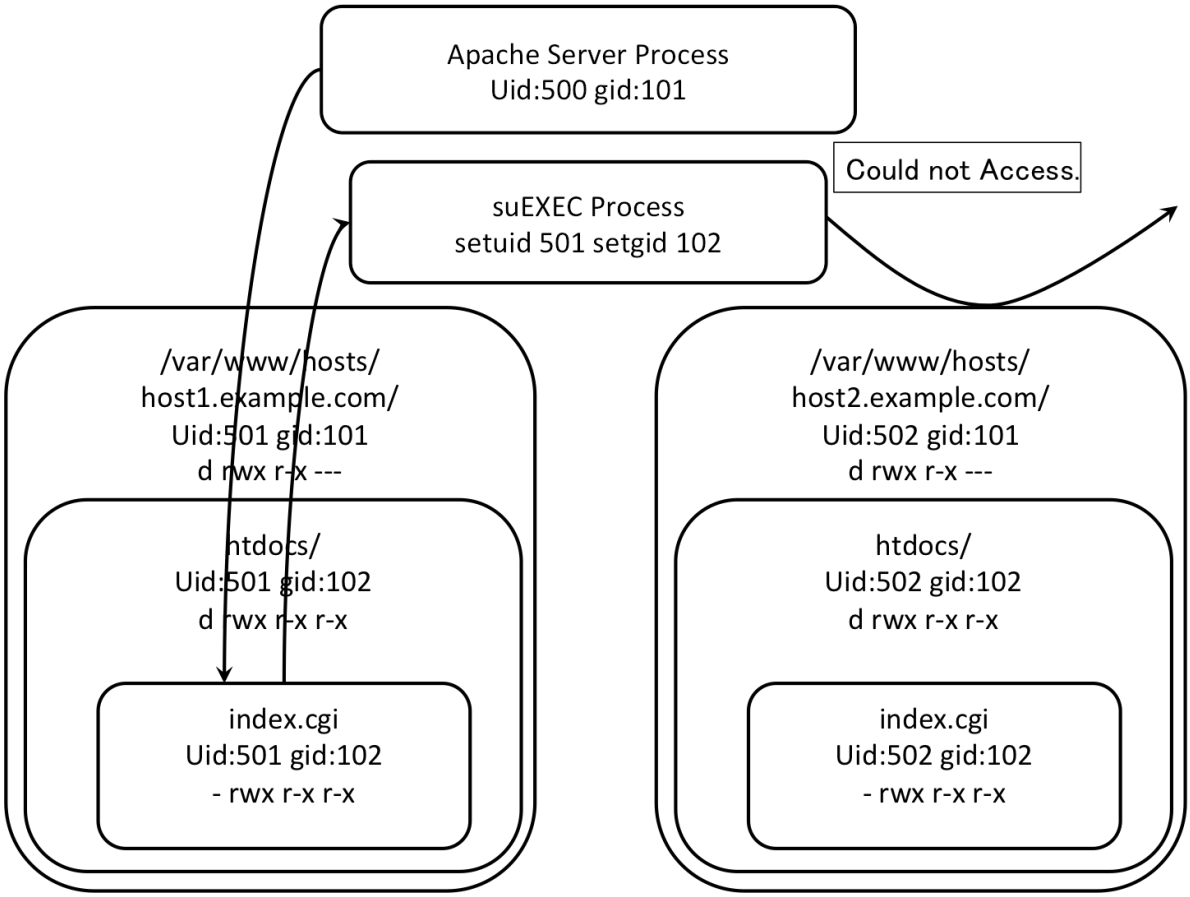

図 2.5　suEXECの利用例

そして，プロセスの権限を変更するシステムコール(setuid()，setgid()システムコールなど)を実行して，プロセスの権限を変更し，CGIプログラムを実行する．そのためuid501，gid102のプロセスは，/var/www/hosts/host2.example.com/配下での読み取り権限であるuid502，gid101に対するアクセス許可がない．このように，suEXECを採用することで，他ホスト領域にアクセスすることができず，他の領域のindex.cgiの閲覧もできない．

図2.6にサーバプロセスとsuEXECの詳細なアーキテクチャを示す．図2.6のように，suEXECはプログラムを実行するたびに，一般ユーザからの実行であってもroot権限で実行されるように設定されたラッパープログラムを実行させて一旦root権限になり，そこから実行対象のプログラムの権限にsetuid()，setgid()システムコールを実行してからプログラムを実行する．このように，CGIプログラム実行毎にプロセスの生成，破棄が必要となるため，性能が低くなるという問題がある[52]．

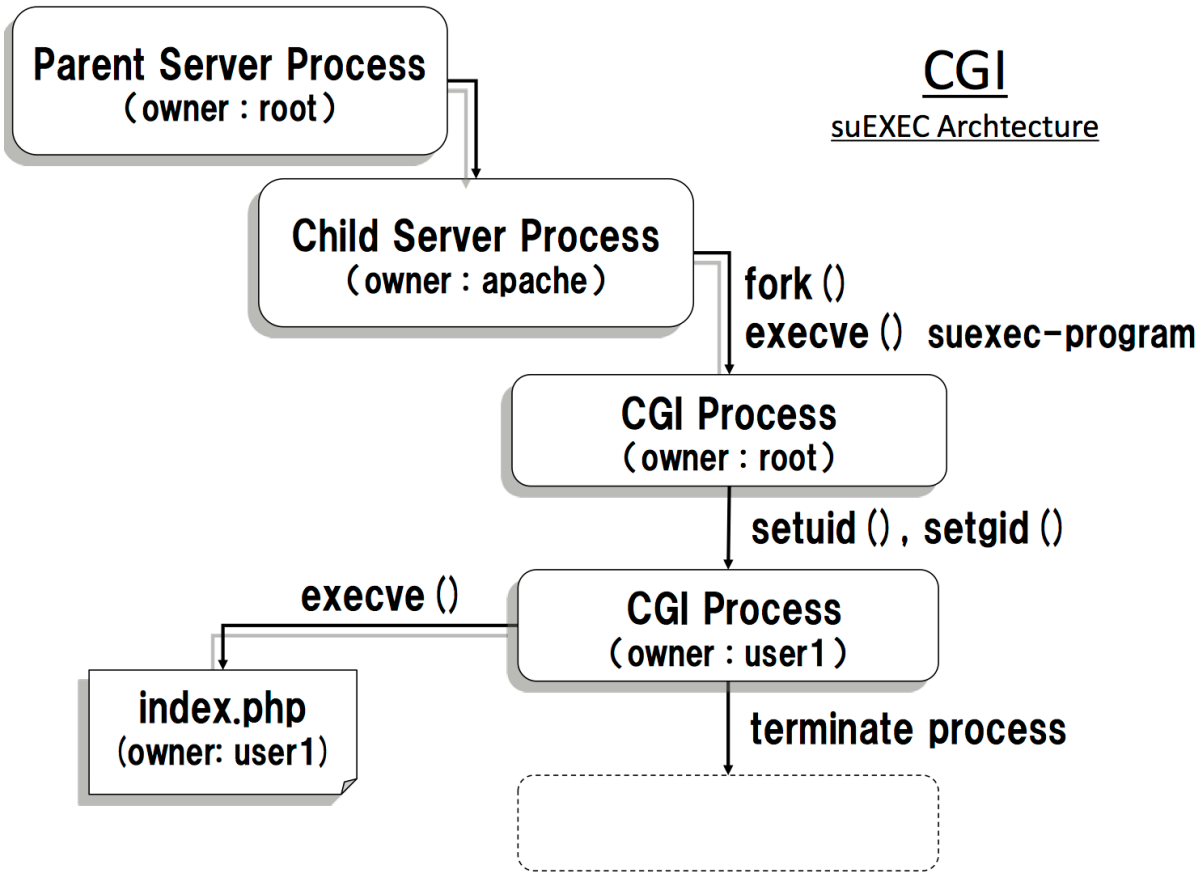

図 2.6　CGI実行方式のアクセス制御アーキテクチャ

セキュリティを高めるために，VirtualHostにおいてsuEXECと同等の機能は必須となる．しかし，suEXECを利用するためには，Webホスティングとしてのサービス仕様において様々な制約がある．制約の中，サービス仕様とセキュリティの関係をどのように解決し最善の選択肢をとるかが重要である．以降でsuEXEC採用における制約について述べる．

2.6.2.1 SuexecUserGroup設定の課題

suEXECを採用するためにはホスト単位でSuexecUserGroupという設定を記述する必要がある．SuexecUserGroupに設定されたuidとgidが，実行対象のCGIプログラムとオーナが一致するか確認し，その上でCGIを実行する際にsetuid()およびsetgid()システムコールにより，オーナを変更することでセキュリティを高めている．しかし，2.5.3節で述べた通り，mod_vhost_aliasではsuEXECの設定のuidとgidを動的に扱う記述がないため，VirtualHost単位にuidとgidを静的に記述しなければならない．そのため，新規VirtualHostを追加するたびに設定数が増加する．

設定反映にはApacheのリロードが必要であるため，ホスト数の増加と共にリロードによる設定読み込み時間も増加する．さらに，設定数の増加によってWebサーバプロセスのメモリ使用量が増加し，その状態でCGI実行方式のような，fork()システムコールやexecve()システムコールを伴う処理を行うと，ページテーブルエントリ数の増大とその複製と削除の処理に起因して処理のコストが高くなり，CPU使用時間を多く消費する．

2.6.2.2 その他のCGI実行方式の課題

suEXECを採用するためには，CGI版を利用しなければならず，シェバン行[4]の記述や適切な実行権限設定をホスティング利用者に強制する必要がある．しかし，例えばWebサイトの動的コンテンツの開発に多く採用されるPHPスクリプトのコードにおいては，一般にシェバン行は記述されない．そのため，シェバン行の記述や実行権限設定はホスティングサービス仕様上の問題となる．

mod_suphp[59]を利用すると，シェバン行の記述や実行権限設定をホスティング利用者に強制する必要なく，suEXECと同様にCGIプログラムをユーザ権限で実行できる．しかし，suEXECと同様にVirtualHost単位でuid，gidを設定ファイル内で指定する必要があり，mod_vhost_aliasでも動的に扱えない．また，他ホスト領域への覗き見を防ぐことができるが，2.5.1節で述べた，システム領域の覗き見問題を解決できていない．

mod_actions[66]を利用すると，CGI実行方式であっても，スクリプトにシェバン行や実行権限を設定しなくても実行できるラッパープログラムに渡す設定ができる．しかし，ラッパープログラムをApacheや各ホストの権限からアクセス可能なディレクトリに安全に配置する必要があり，設定や構成が煩雑になりがちである．また，ラッパープログラムはURLからアクセスできる領域に配置する必要があり，顧客のURLを一部専有することが問題となる場合もある．2.6.1節で述べた，システム領域の覗き見問題も解決できない．

2.6.3 DSO実行方式のためのセキュリティ機構

DSO実行方式はApacheモジュールとして組み込まれたインタプリタがプログラムを実行するため，一般的にCGI実行方式と比較して性能が高くなる．また，スクリプトの処理を渡すインタプリタを指定するために，スクリプトの行頭に記述するシェバ

[4] UNIXで実行されるスクリプトの1行目に，スクリプトを渡すインタプリタを指定するために書く行．#!/bin/shなどと記述される．英語表記はshebang.

ン行や権限を細かく設定する必要がない．しかし，Apacheに組み込まれて実行される以上，基本的にはApache権限で実行されるため，図2.4と同様の他ホストが覗き見される問題が生じる．以下，DSO実行方式を安全に利用するためのセキュリティ機構としてこれまでに提案されているものを紹介するとともに，それらの課題について論ずる．

2.6.3.1 PHPのセーフモード

DSO実行方式において，広く使われているDSO版PHPは，他ホストの領域を閲覧できないようにするため，セーフモードという機能があった．セーフモード機能を利用すると，DSO版PHPであっても他ホスト領域のファイルを覗き見できない．しかし，PHP特有のセキュリティ機構であり汎用性が低いこと，共有サーバ上のOSやファイルシステム上のセキュリティ問題をPHPアプリケーションのレイヤーで解決しようと試みるのはアーキテクチャ上正しくないといった理由から，PHP5.3.0で使用が非推奨となり，PHP5.4.0では削除された[76]．

2.6.3.2 DSO実行方式とmod_suid2やmod_ruid

DSO実行方式を採用した場合でも，mod_suid2[28]やmod_ruid[27]というモジュールを利用すると，他ホスト領域の閲覧を防ぐことができる．mod_suid2や関連研究[25]では，Apacheのサーバプロセスをroot権限で起動しておき，リクエストを処理する度にsetuid()およびsetgid()システムコールによりユーザ権限に降格する．これによって，Apacheの権限とは別の権限でプロセスを実行できるため，suEXECと同様，他ホスト領域を閲覧できなくなる．しかし，処理後はサーバプロセスが一般ユーザ権限であるため，権限を元のroot権限に戻すことができない．そのため，ユーザ権限に降格されたプロセスはコンテンツ処理後に破棄する必要がある．その結果，プロセスを再利用できず，DSO実行方式を利用していたとしても，suEXECよりも性能が大きく低下する．また，セキュリティの観点からは，サーバプロセスをrootで起動させていると，万一サーバプロセスそのものに任意のコマンドを実行できるなどの脆弱性があった場合や，設定ミスによってサーバプロセスの権限でコンテンツが動作した場合に，悪意のあるユーザが容易にrootが持つ全ての特権を得られるという問題がある．一方で，Apacheのプロセスをユーザ権限で起動している場合は，setuid()やsetgid()システムコールを実行することができない．アクセスするクライアントが正確に特定できるようなイントラネットの環境において，ユーザ権限であってもsetuid()システムコール等を実行可能にする手法[85]は存在するが，不特定多数のクライアントには対応していない．

mod_ruidや関連研究[88]を利用すると，2.2.2節で述べたように，一時的にユーザ権限で起動しているサーバプロセスに，rootの特権を細分化したLinux Capability[87]と呼ばれる機構の内，CAP_SETUID，CAP_SETGIDの特権を与えられる．図2.7にmod_ruid2の詳細なアーキテクチャを示す．

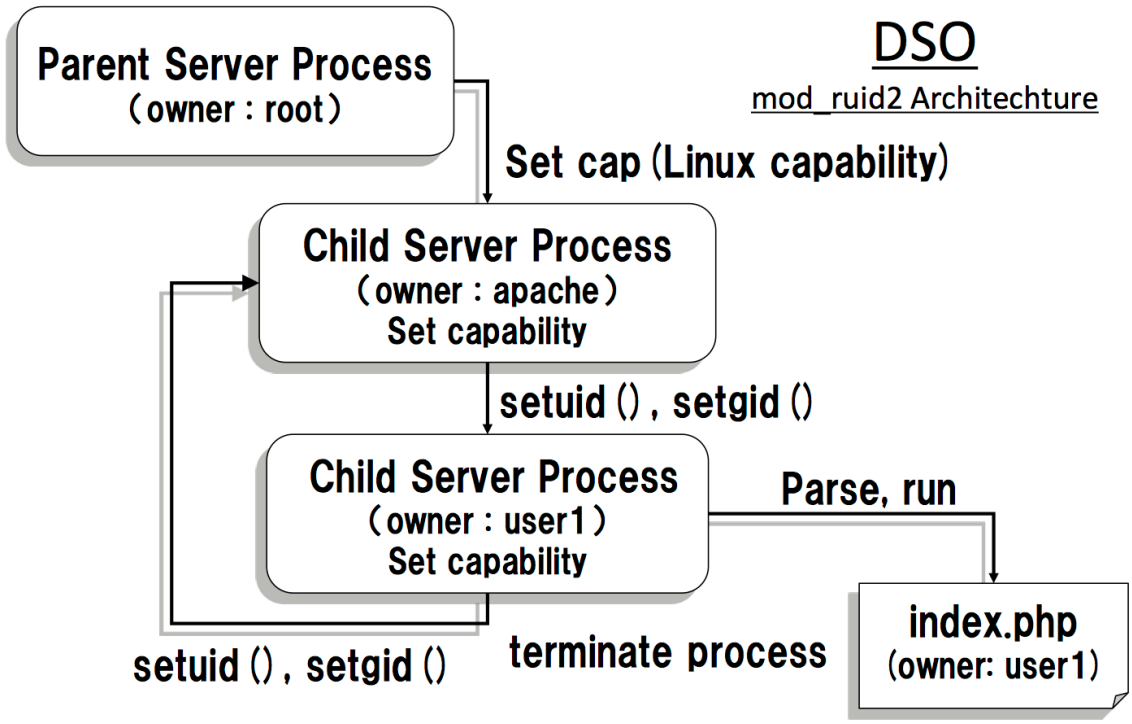

図 2.7　DSO実行方式のアクセス制御アーキテクチャ

特権を与えられたサーバプロセスは，root権限で実行されていなくても，setuid()およびsetgid()システムコールを実行可能となる．その後，mod_suid2同様にApacheのサーバプロセス自体を任意のuid，gidに権限変更してから処理を実行し，再度，元のuid，gidに戻す．この仕組みによって，DSO実行方式であっても，PHPスクリプトは他ホスト領域を閲覧できない．また，実行後でも，元のサーバプロセスの権限に戻すことで，プロセスの再利用も可能にしているため，DSO実行方式の性能を維持できる．しかし，このようなプロセスは，rootのように全ての権限を持たないものの，setuid()およびsetgid()システムコールを実行できる特権を保持している．図2.7におけるindex.phpのようなWebアプリケーションの脆弱性をつかれ悪意のある者に乗っ取られた場合，setuid()およびsetgid()システムコールによる権限変更を利用し，他ホスト領域のファイル閲覧や変更および不正プログラムの配置や配布等が可能となる．サーバプロセスに権限を変更できる特権の保持を許すことは，同時に数多くの脆弱性を許すことになる．一方で，図2.8のように，setuid()および

setgid()システムコールを実行した後にCAP_SETUIDおよびCAP_SETGIDのCapabilityを放棄し，処理後にプロセスを復帰できないように改修すれば安全であるが，やはりサーバプロセスが再利用できなくなり，mod_suid2同様性能は著しく低下する．

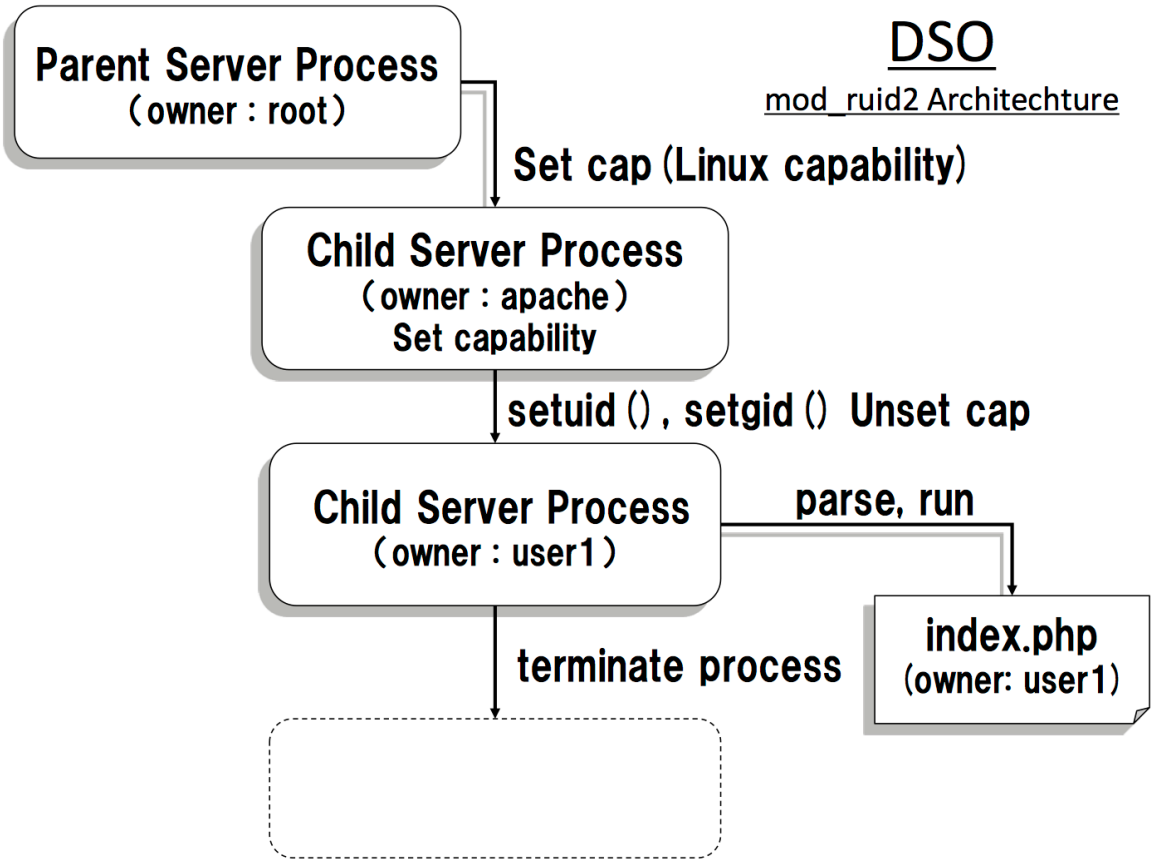

図 2.8　脆弱性を修正したDSO実行方式のアクセス制御アーキテクチャ

一般に，サーバプロセスにアクセス制御を設定後に再度解除するというアプローチは性能上の利点を得られるが，共有型の大規模Webホスティング基盤のセキュリティを考える上でリスクが非常に大きく，脆弱性をつかれた場合の利用者や閲覧者への被害は甚大であり，避けるべきと考えられる．

Webサーバからの権限変更を可逆的に変更可能にしながら，実行されるプログラムからは権限を変更されないように，プログラムから実行されるシステムコールをフック[5]してプログラムから実行される権限変更の処理を無効にしてセキュリティを担保する手法[88]が提案されている．しかし，Linuxにおいては，ライブラリ関数はLD_PRELOAD[6]によって簡単にフック可能であるが，システムコールを適切にフックす

[5] 一連の処理の中で特定の処理フェーズが呼ばれた時に，同時に，あるいは，代わりに別の処理も実行できるように予め処理を登録しておくこと

[6] http://man7.org/linux/man-pages/man8/ld.so.8.html

るためにはLinuxカーネルに直接変更を加える必要があり，可搬性が低く，カーネルやライブラリを継続的に更新することが求められる現場において運用上の問題になることが多い．

以上に述べた通り，現状，高集積マルチテナントアーキテクチャのWebホスティング基盤において，運用上の課題を考慮しながら，DSO実行方式を安全かつ高速に利用する手法は存在しないと考えられる．

2.7 マルチテナントアーキテクチャのリソース制御の課題

限られたコンピュータリソースで複数のホストをできるだけ高集積に管理・運用するためには，リソースの分離と，各ホストが安全にコンテンツを配置できるような権限分離の管理が重要である．従来手法では，リソース分離のために，ファイルやホスト単位で同時接続数を制御したり[12]，OSの負荷によってリクエストを拒否したりする[73]．しかし，CPUやDisk I/O等のコンピュータリソースは共有のため，特定ホストへのアクセス集中や，多くのリソースを消費するアプリケーションへのたった一つのリクエストによってリソースを占有し，他のホストに影響を与える問題があった．そのような問題を解決するためには，リクエスト単位でリソースを分離し，各リクエストが互いに影響を受けないようなリソース制御アーキテクチャが必要である．また，リクエスト単位でリソース制御ができれば，収容されている各ホストに対するリクエストを制御することにより，ホスト単位でのリソース分離も可能となる．

単一のサーバで複数のホストを管理する場合，セキュリティを担保するための権限分離を考慮しながら，複数のホストでサーバのハードウェアリソースを共有する構成が一般的である．これまで，マルチテナントアーキテクチャにおけるWebサーバのハードウェアリソースを仮想的に分離する手法は，主に以下の3種類が提案されている[9][35]．

(1) KVM[23][34]やVMware等の仮想マシンで複数のOSでリソースを分離する手法
(2) OpenVZ[5]のようなコンテナ方式でカーネルを共有しながらプロセス権限でOSリソースを分離する方式
(3) 仮想ホスト方式のように単一のサーバプロセスで複数のホストを扱う手法

ハードウェアリソースが潤沢にあり，サーバの運用面やセキュリティおよび可用性を重視した場合は，手法(1)の，ホストそれぞれに対して，個別の仮想マシン[21][19][18]を割り当てる構成がとられてきた．しかし，これらはコンピュータリ

ソースの面で非常にコストが高く，オーバーヘッドも大きいため，限られたリソースで，高集積にホストを収容するには不向きである．

(2)の方式では，ホスト単位でchroot()システムコールによるファイルシステムの隔離や2.2.3節で述べたようなプロセスリソース管理技術を組み合わせたコンテナ環境[62][13][83]を構築し，その環境毎にサーバプロセスを起動させる必要があるため，プロセス数がホスト数に依存する．プロセス数がホスト数に依存すると，収容数が搭載メモリ容量により制約されるため，高集積にはむいていない．通常Webサーバプロセスは性能を担保するために数十個のプロセスを起動させるが，5万ホスト収容する場合にホスト毎に20個プロセスを起動させたとすると，100万プロセス起動させる必要がある．その収容設計で，32GBのメモリを搭載したサーバに収容する場合を考えると，1プロセス最大32KB程度しか割り当てることができず，Apacheの1プロセスが通常数十MB程度消費されることを想定すると，安定稼働は到底不可能である．

(2)の方式のメリットは，ホスト毎にサーバプロセスを起動させる方式であるため，(3)の仮想ホスト方式では設定できないような，サーバプロセス全体の設定や，2.2.3節で述べたようなプロセス単位でのリソース制御や隔離機能を利用することができる．また，プロセスとしては完全に他のホストのプロセスと分離しているため，特定ホストのリクエスト処理に時間がかかっていたとしても，プロセスレベルで他のホストは影響を受けない．しかし，サーバプロセス数がホスト数に依存するため，高集積は困難である．

(3)の仮想ホスト方式のように，単一のサーバプロセスで複数のホストを管理するアーキテクチャの場合，サーバプロセスがホスト数に依存しないため高集積が可能である．しかし，(3)の方式では，単一のサーバプロセスで複数のホストを処理しているため，特定のホストやリクエスト処理がリソースを占有した場合に，他のホストやリクエスト処理が影響を受けやすいという問題がある．

以上より，ホスト単位で適切にリソースを分離するためには(1)，(2)が適切であるが，収容効率は悪い．(2)はプロセス毎にリソース分離できるが，高集積マルチテナントアーキテクチャのリソース制御においては，少なくとも個々のプロセスに適切なリソース制約を与えて，ホスト単位のリソース分離を図る必要がある．(3)は高集積なマルチテナント環境を構築する際には適切であるが，単一のサーバプロセスで複数ホストのリクエストを処理するため，最もリソース分離に向いていない．

高集積マルチテナントアーキテクチャのリソース制御の要件をまとめる．(3)のような高集積化が可能な仮想ホスト方式を採用した上で，一つのリクエストがコンピュータリソースを大きく占有しようとしても，その他のリクエスト処理が影響を

受けないようにする必要がある．さらには，各リクエストが継続的に処理を行うためには，リクエストを処理中のプロセスを，一時的に限られたリソースの範囲内に分離してから，その制限されたリソース範囲内でリクエストを処理すれば良い．また，管理者がそのようなリソース制御ルールをプログラマブルに記述できれば，柔軟なリソース制御が可能になる．

本論文の第6章では，柔軟なリソース制御を実現するため，第5章の安全かつ高速に動作するWebサーバ機能拡張支援機構を応用し，同一のサーバプロセス上で，リクエスト毎に管理者が任意のリソース分離が可能なリソース制御アーキテクチャを提案する．

2.8 結語

2章では従来研究や基礎概念と用語の整理を行った．要点は以下の4点である．

第1に，Webホスティングサービスは，各ホスト領域に利用者が任意のWebコンテンツを置くことができるため，管理者はOSやミドルウェアといった基盤技術でマルチテナント環境のセキュリティや安定性を実現しなければならない．筆者はホスティングサービス提供事業者において，ハードウェアリソースを必要最小限に抑えるために，アクセスのあったホスト名でコンテンツを区別し，単一のプロセスで複数のホストを処理する方式であるVirtualHostを用いて，ホスト高集積型のWebホスティング基盤を構築した．しかし，高集積であればあるほどその基盤運用と性能の安定化は困難で，運用技術の改善と性能劣化の少ないセキュリティ機構の設計が課題であった．

第2に，高集積マルチテナント方式において，ホスト単位で権限を分離するための従来のアクセス制御アーキテクチャは，権限分離のためにプロセスの生成，破棄が必要となり，性能が低い．また，アクセス制御アーキテクチャの実装が，インタプリタや，CGI，DSOプログラム実行方式別に複数用意されており，サーバ管理者が扱いにくい．

第3に，運用面や安定性を考慮した高集積マルチテナント環境を構築する際に，Webサーバソフトウェアの機能拡張も必要になる場合が多い．3章，4章においても，導入障壁を下げるために，新しいアーキテクチャを既存のWebサーバソフトウェアの機能拡張として実装している．Webサーバの機能拡張において，高速かつ軽量に動作することを重視した場合，C言語による実装が主流であったが，生産性や保守性を考慮した場合はスクリプト言語で機能拡張を行う手法も提供されている．しかし，従来手法は，Webアプリケーションの実装だけでなく，Webサーバの内部処理を拡張することを主目的とした場合，高速性，メモリ使用量，安全性の面で課題が残る．

第4に，高集積マルチテナント環境を実現するWebサーバソフトウェアにおいて，CPUリソースやDisk I/Oを多く消費するリクエストがあった場合，その処理をリクエスト単位で制御する必要がある．しかし，既存のWebサーバのリソース制御は，リクエストの同時接続数や単位時間当たりのリクエスト数が指定の閾値を超えた場合や，管理者があらかじめ設定したCPU使用時間やメモリ使用量の閾値を超えた場合にリクエストを強制的に切断，あるいは，拒否するようなアーキテクチャになっている．そのため，処理を継続しながらもリソースを制御することができない．

本論文では，以下の各節にて上記の4点の課題を解決するための手法を提案する．

3 高集積Webホスティング基盤のセキュリティと運用技術の改善

3.1 緒言

高集積マルチテナントアーキテクチャであるApacheのVirtualHost機能を用いて大規模対応基盤を構築する場合に，運用面とセキュリティ面の課題が生じることを，2.4節，2.5節，2.6節で体系的に明らかにした．従来の手法において，これらをすべて同時に解決している手法は存在しない．そこで，3章では，高集積マルチテナントアーキテクチャであり，特に基盤技術による制御が必要不可欠な高集積型Webホスティングサービスに対して，2章で述べた仮想ホスト採用と大規模対応にともなって生じる運用面とセキュリティ上の課題に対して，リソース制限を実現するApacheモジュールの開発と既存のアクセス制御手法であるsuEXECの改善によって，それらを解決する手法を提案する．本手法によって，従来技術の枠組みを大幅に変更することなく改善が可能となり，ホスティングサービス事業者が信頼性と運用性の高い大規模Webホスティング基盤を容易に構築することができる．

3.2 運用面の課題の解決

3.2.1 リクエスト毎でのリソース消費量測定機能

2.5.1節において述べたように，プロセス情報からの高負荷対象の特定や，リソースの測定が困難である．そこで，クライアントからのリクエストをApacheが受け付け処理を行ってからレスポンスを返すまでに，プロセスが消費したリソース量を測定するモジュール，mod_resource_checker[7]を開発した．このモジュールによって，実行されたプログラムが，管理者によって設定されていあたシステムCPU時間，ユーザCPU時間，メモリ使用量の閾値を超えていた場合に，プログラムの絶対パス，VirtualHost名，システムCPU使用時間，ユーザCPU使用時間，メモリ使用量が計測されファイルに記録される．そのデータをもとに，高負荷ホストやスクリプトをリアルタイムで調査，検知できる．また，この機能はコンテンツを設置する側の開発者にとっても，スクリプトのリソース消費量を知る上で役立つと考えられる．

[7] https://github.com/matsumotory/mod_resource_checker

3.2.2 コンテンツへの同時接続数の制限

人気サイトでは，しばしば同一スクリプトやメディアファイル，ホストに対して，複数のクライアントから同時に大量のアクセスが発生する．その結果，高負荷となってサーバのレスポンスが大きく低下し，場合によってはサーバダウンへと繋がる．しかし，2.5.2節で述べた通り，高集積マルチテナントアーキテクチャにおけるホスト単位での制限設定では，ホスト単位やコンテンツ単位で同時接続数を制限することができない．そこで，高負荷ホスト全体や高負荷スクリプトへのアクセス数を制限するために，リクエスト対象への最大同時接続数を設定するモジュールmod_vlimit[8]を開発した．設定対象は，任意のホストやファイル名，ファイルへの絶対パス，任意のディレクトリ，正規表現にマッチしたファイルやフォルダ等である．絶対パスのパラメータが必要である理由として，ホスト領域で利用者がシンボリックリンクを使って，本来制限が設定されているべきコンテンツのパスを複数作成することがあるためである．例えば，シンボリックリンクによって，負荷のコンテンツのパスが複数あらわすことができた場合，一つのパスにだけ制限をかけていたとしても，別のバスからは制限なくコンテンツにアクセスすることができる．図3.1と図3.2に設定例を示す．

```
<Directory "/path/to/host/">
     VlimitIP 5
</Directory>

<Files "a.txt">
    VlimitIP 10 /path/to/a.txt
</Files>

<FilesMatch "^.*\.txt$">
    VlimitIP 10
</Files>
```

図 3.1　IPアドレスとコンテンツ単位の同時接続数制限例

[8] https://github.com/matsumotory/mod_vlimit

```
<Files "a.txt">
    VlimitFile 10 /path/to/a.txt
</Files>

<FilesMatch "^.*\.txt$">
    VlimitFile 10
</Files>
```

図 3.2　コンテンツへの同時接続数制限例

図3.1は，接続元のIPアドレスが，コンテンツに対して同時に接続する数を制限している．例えば，二番目の設定は，ファイル名がa.txtのファイルに対して，同一IPアドレスから最大10接続できることを意味する．図3.2では，特定のWebコンテンツに対して，IPアドレスによらない，不特定多数からの同時接続数を制限する設定例を示している．一番目の設定では，a.txtに対して，不特定多数のクライアントから同時に合計10リクエストまで受け付けることを意味する．また，二番目の設定では，正規表現を用いて，マッチするコンテンツを複数しており，そのコンテンツそれぞれで同時接続数制限を設定している．同時接続数の設定値を超えた場合は，HTTPのステータスコード503（Service　Unavailable）を返す．また，同一クライアントIPからの同時接続数も設定できるようにした．

3.2.3 リソース変化に対応した同時接続数の制限

2.5.2節において，なんらかの原因でサーバプロセスが停止した場合は，全てのホストが停止することになると述べた．典型的には，リソースを多く占有するスクリプトにアクセスが集中し，サーバが高負荷となってOSが停止してしまうようなパターンが挙げられる．そこで，CPU処理やI/O処理の負荷の目安となるロードアベレージ[81]の数値に着目し，リクエストを受けた後，ロードアベレージの数値によってレスポンスを返すかどうかをApacheが判断するモジュールmod_lalimit[9]を開発した．ロードアベレージの数値を閾値として設定し，その数値を超えた場合はステータスコード503を返すことで安全に処理を中断させる．1分平均のロードアベレージを閾値に設定できる．また，制限対象として3.2.2節の記述方法と同様に，特定のファイ

[9] https://github.com/matsumotory/mod_lalimit

ル名や，正規表現を用いたマッチ記述も可能である．これにより，比較的大きいリソースを必要とするコンテンツに対して制限しておくことにより，少なくとも高負荷時にそのリクエストに対する処理が並列実行されることがなくなるため，リソース占有によるサービス停止の可能性を低減させることができる．図3.3は，ロードアベレージが30を超えていた場合に，.cgi拡張子を持つすべてのファイルに対するリクエストを拒否する設定である．

```
<FilesMatch ".*\.cgi$">
    LAlimit 30
</FilesMatch>
```

図 3.3　コンテンツへのロードアベレージによる制限例

ロードアベレージによる制限のメリットとして，高負荷時にサーバ管理者が改めて設定を記述することなく，負荷が高ければ自動的に制限がかかる点が挙げられる．

3.2.4 mod_vhost_aliasと.htaccessによる動的設定

2.4.3節で述べたmod_vhost_aliasのDCMVH機能を利用すると，複数のホストの設定を一つの設定で記述することができる．図3.4に，通常のVirtualHostの設定を示す．

```
<VirtualHost *:80>
    DocumentRoot /path/to/vhosts/host1.example.com/
    ServerName host1.example.com
    SuexecUserGroup host1 host1
</VirtualHost>
<VirtualHost *:80>
    DocumentRoot /path/to/vhosts/host2.example.com/
    ServerName host2.example.com
    SuexecUserGroup host2 host2
</VirtualHost>
```

図 3.4 通常のVirtualHostの設定例

Webホスティングのように，ホスト単位で厳密に権限分離が必要な場合は，SuexecUserGroupのようにホスト毎のオーナを静的に設定ファイルに記述する必要があるため，SuexecUserGroupの動的設定に対応していないDCMVHでは記述することができない．そこで，mod_vhost_aliasおよびsuEXECを改良し，mod_vhost_aliasでsuEXECの動的設定ができない問題を，各VirtualHostのsuEXECのユーザ名とグループ名といったオーナ情報を同一のダミーとして設定し，suEXECプログラム内部で実行ファイルからユーザ名とグループ名を解釈できるようにsuEXECを改修することで解決した．この改修とDCMVHによって，ホスト単位で権限分離が必要な場合でも，全てのVirtualHostの設定を一つの設定に書き直すことができるようになった．図3.5に，mod_vhost_aliasによる統一的設定例を示す．

```
VirtualDocumentRoot "/path/to/vhosts/%0"
SuexecUserGroup dummy dummy
```

図 3.5 mod_vhost_aliasとsuEXECの改修による統一的設定例

また，オーナの読み替えの際には，単純な実装では，仮にファイルのオーナがrootのuidとgidであった場合はrootで実行されてしまうため，rootのような特権でWebコンテンツが処理される危険性を排除するためのエラー処理を実装した．さらに，サーバ管理者が運用において不適切なオーナのファイルを配置してしまう可能性をできる限り排除するために，実行対象プログラムと実行対象プログラムが存在するディレクトリ，ドキュメントルートディレクトリのuidとgidがそれぞれ互いに一つでも異なっていた場合にもエラーを返すような厳密なオーナチェックを実装した．顧客ホスト領域毎にchroot()システムコールにより権限分離する方法と併用することで，2.6.2.2節で述べたmod_suphpによりユーザ毎の個別設定を省略する方法よりも，PHPプログラム経由でシステム領域を覗き見できない点においてセキュリティ面で優れており，複雑な実行権限設定の問題も同時に解決した．

また，mod_vhost_aliasでは，環境変数に保存されるドキュメントルートが，本来保存されるべき変数を読み替えたユーザ毎の絶対パスにならない問題があり，ホスティングサービスの利用者が環境変数を使ったプログラムを配置する可能性を考慮し，正しいドキュメントルートで保存されるようにmod_vhost_aliasを改良した．さらに，リクエスト毎に読み込まれる各ホスト専用の制限設定が書かれたファイルを，各ホストのドキュメントルートディレクトリの一つ上位のディレクトリに配置でき

るようにした．ドキュメントルートの外に配置することによって，サービス利用者がWebコンテンツとしてアップロードするファイルと重複しない．

3.2.1節，3.2.2節および3.2.3節で提案したApacheモジュールは，各ホスト専用の制限設定が書かれたファイルに設定を記述することができ，Apacheのリロードを実行すること無くリクエスト毎に新たな設定を反映させることができる．その結果，ホスト数に依存した煩雑な設定無く，また，Apacheのリロードをせずに制限の設定や新規に追加するホストの設定を反映させることができ，大幅に運用性とサービス性が向上したといえる．

3.3 セキュリティの課題の解決

2.6.2節で述べたsuEXEC採用に伴うCGI実行方式のセキュリティの課題を解決するために，suEXECを改修した．以降で，suEXECの改修内容を詳細に述べる．

3.3.1 suEXEC時にホスト領域でCGIプログラムを隔離する機能

システム領域や他ホスト領域を覗き見できないように，suEXEC時に各ホスト環境でchroot()システムコールにより，ルートディレクトリを各ホスト領域に移動し，隔離してからスクリプトを実行するように改修した．図3.6はsuEXECプログラム内部でchroot()システムコールを実行する仕組みの概要図である．これにより，CGIプログラムはホスト領域内の隔離された領域で実行されるため，利用しているホスト領域外のファイルを閲覧することができない．一方，隔離された領域でプログラムを実行するため，ホスト単位で個別にライブラリを含んだ実行環境をドキュメントルートディレクトリ配下に事前に用意しておく必要がある．ただし，複数の実行環境のファイル間をリンクし参照のみにすることにより，実行環境の構築や使用容量のコストを下げることは可能である．

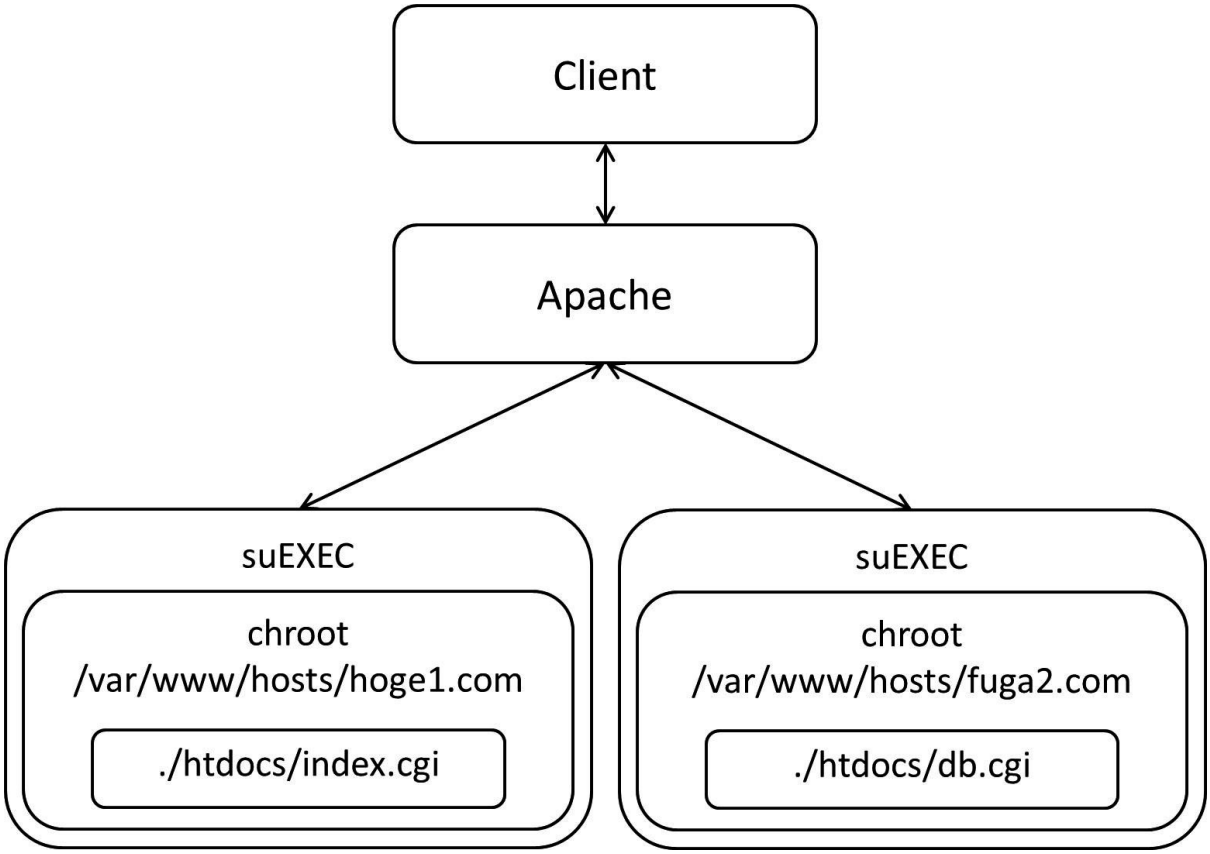

図 3.6　suEXEC実行時にchroot()システムコールを実行する仕組み

3.3.2 suEXEC時のCGIプログラムとインタプリタの紐付け

2.6.2.2節で言及したように，Webホスティングサービスで広く使われるPHPでは，通常シェバン行をスクリプトの先頭に記述しない．そのため，CGI実行方式によってPHPプログラムを実行するためには，適切にプログラムをインタプリタに渡す仕組みが必要である．そこで，PHPプログラムに対するリクエスト時は，suEXECプログラム内部でexecve()システムコールを実行し，CGI実行方式用のPHPインタプリタファイルにリクエストされたプログラムパスを引数に渡して実行するように改修した．これによって，PHPプログラムのシェバン行や実行権限の有無をサービス利用者が意識することなく，CGI実行方式のPHPプログラムをsuEXECで実行できる．

改修によって，複数のモジュールやラッパー等を組み込む必要がないという点で非常にシンプルな構成になっている．また，ホスティング利用顧客がDSO版PHPを扱うために，シェバン行を記述しなかったPHPスクリプトに対しても，suEXEC内部でCGI版PHPとして実行されるため，CGI版かDSO版かをホスティング利用顧客が気にする必要が無い．一方で，PHPプログラムとインタプリタファイルの紐付けをsuEXECで行う必要があるため，PHPのようなシェバン行を通常書かないようなプログラミング言語

を提供する場合には，別途ホスティングサービス提供者による紐付けが必要である．

3.4 性能評価

単一のApacheサーバプロセスで複数のホストを処理する方式をとる場合に，仮想ホストの追加や設定変更時にサーバプロセスをリロードすることなく，カーネルを書き換えずに動的コンテンツ実行時のセキュリティを担保するためには，DSO実行方式とmod_suid2又はmod_ruid2を採用し，サーバプロセスそのものを動的コンテンツ処理毎に破棄する以外に方法はなかったことを2.6.3節で述べた．サーバプロセスそのものの生成破棄は，CGI実行方式におけるプロセスの生成破棄と比べて非常に処理に時間がかかる．一方，3.2.4節で示したmod_vhost_aliasによる統一的設定手法により，仮想ホストの設定に応じてsuEXEC内部で動的コンテンツファイルの権限情報から動的に権限分離を行うことで，通常のCGI実行方式と同程度の性能を維持しつつ，仮想ホストの追加や設定変更時にサーバプロセスのリロードが必要ない運用が実現できた．

以上の考察に基づき，CGI実行方式と，DSO実行方式でサーバプロセスを生成破棄する方式との間で性能の比較を行って，提案手法の優位性を評価した．使用する言語はCGI実行方式およびDSO実行方式両方で実行可能なPHPとし，実行方式の差をより顕著にするために，プログラム上の処理はサーバ情報を文字出力するだけの簡易な処理で比較を行った．表3.1に実験環境を示す．

表 3.1　CGI実行方式のアクセス制御の性能評価のためのテスト環境

	Client Machine	Server Machine
CPU	Intel Core2Duo E8400 3.00GHz	Intel Xeon X5355 2.66GHz
Memory	4GB	8GB
NIC	Realtek RTL8111/8168B 1Gbps	Broadcom BCM5708 1Gbpz
OS	CentOS 5.6	CentOS 5.6
Middleware	-	Apache/2.2.3

ApacheのVirtualHost方式では，アクセスのあったホスト名に紐づくファイルパス名を導出する．単一のホストに同時接続する場合と，複数のホストに同時に接続する場合で，アクセス先のファイルのパスが違うだけで，本質的な差は無いと考えられる．また，本論文で改良したsuEXECにおいては，VirtualHost方式そのもののアーキテクチャは変更していない．そのため，変更した実装であるsuEXECの内部処理における性能劣化の評価は，一つのホスト名に対してのみ行っても十分であると考えた．

以上から，それぞれの実行方式に対して，Apacheのベンチマークコマンドを用い，文字列を出力するPHPスクリプトに対して，以下の3つの方式での比較を行った．

(1) DSO実行方式でサーバプロセスそのものを破棄する方式
(2) 本手法でsuEXECを改良したCGI実行方式
(3) 通常のsuEXECを使ったCGI実行方式

表3.1の環境において，同時接続数50，総接続数10000で負荷をかけたところ1秒間に処理できるリクエスト数は，(3)の方式の場合は約102.56リクエストであった．しかし，この構成では仮想ホスト追加毎にApacheのリロードが必要となったり，仮想ホストの数に比例して設定行数が増加したりするため，運用性が低い．また，2.6.2.1節で述べた通り，設定数の増加によりサーバプロセスのメモリ使用量が増加するため，CGI実行方式の性能が低下する．一方で，(1)の方式の場合は，Apacheのリロードは必要なく，設定も動的に記述できるものの，約24.88リクエストにまで低下していた．

次に，改修後のsuEXECをCGI版PHPに対して適用した(2)の方式について，同様のテストを行った．その結果，1秒当たりの処理可能リクエスト数は102.44で，(3)の方式である改修前のsuEXECとほぼ同等の数値となった．(2)の方式は運用を考慮した手法となっており，仮想ホスト追加時のApacheのリロードが不要であり，設定も動的に記述できるため，(1)の方式と同程度に運用性が高い．また，同時接続数を変動させても，(3)の方式と同等の性能が得られ，chroot()システムコールやその他のエラー処理の追加実装による性能劣化はほとんど見られなかった．

図3.7に同時接続数を変化させた場合の評価結果をグラフで示した．横軸は同時接続数，縦軸は1秒間にサーバ側が処理できたリクエストの数を示している．

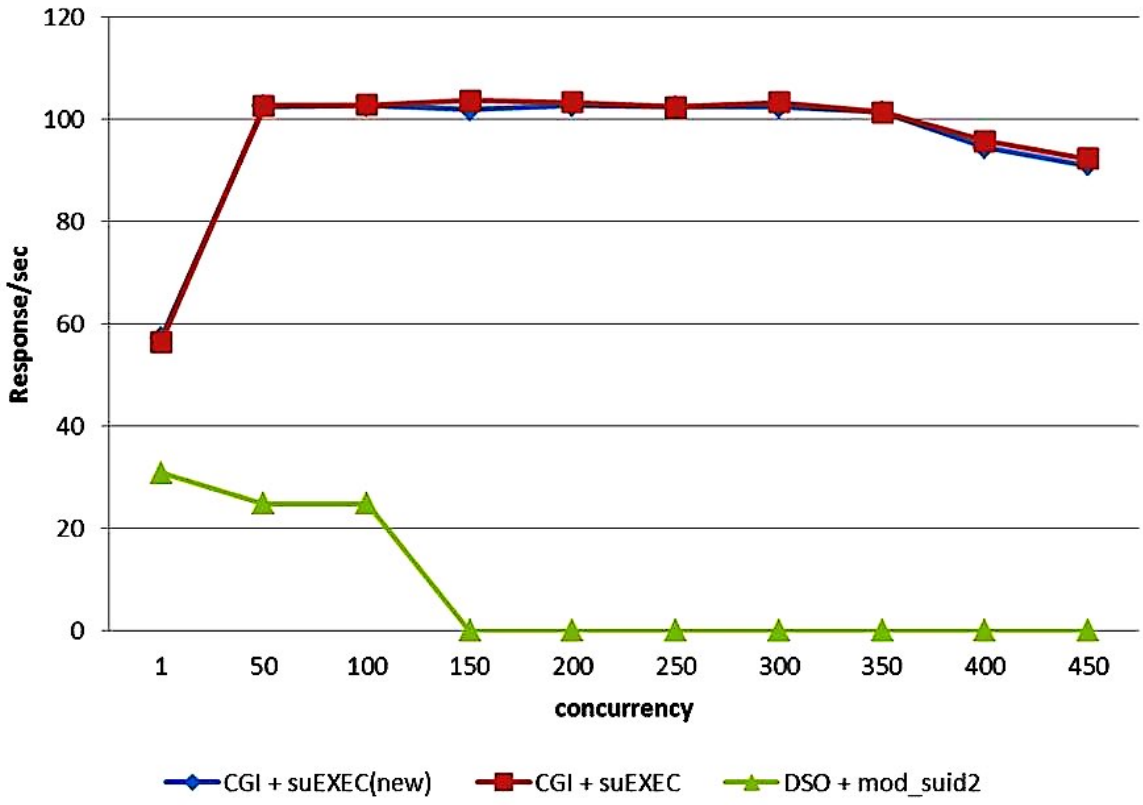

図 3.7　実行方式別の性能比較

(1)の方式（図3.7でのDSO + mod_suid2）では，同時接続数150以降ではコネクションタイムアウトとなりレスポンスを返すことができなくなった．これは，リクエスト毎にサーバプロセスそのものの生成・破棄が必要となり，権限分離のコストが非常に高いためである．一方，(2)の方式（図3.7でのCGI + suEXEC(new)）と(3)の方式（図3.7でのCGI + suEXEC）では，(1)の方式と比較して，同時接続数を増加させても安定的に処理できている．

共有サーバにおいてDSO実行方式で動的コンテンツを実行する場合，セキュリティを担保するためにmod_suid2などによる権限分離が必要であるが，mod_suid2を適用した結果，CGI実行方式よりも大きく性能が落ちることがこの結果からも分かる．3.3.2節で示したように，提案手法では，ホスティング利用顧客がDSO版PHPで動作するように，シェバン行を書かずにPHPスクリプトを記述している場合でも，Apacheを経由してsuEXECによりCGI版PHPとしてそのまま動作するように改修されていることで，顧客が管理するコンテンツを修正することなくCGI版PHPへと移行できる設計となっており，運用上の有効性があると考えられる．

以上のようにして，これまで高集積型のWebホスティング基盤を運用した場合に生じていたセキュリティと運用上の課題を解決できた．これによって，運用者の負担

が大幅に低減され，実運用に耐えうる大規模構成を取ることができると考えている．

3.5 結語

本章では，汎用性の高い大規模Webホスティングサービスの基盤技術として，高集積マルチテナントアーキテクチャを実現する際にApacheのVirtualHostを採用した場合に生じていた運用面とセキュリティの課題を改善する手法を提案した．2章で明らかにしたVirtualHostの課題に対して，既存の手法が数多くあるが，現状，汎用性が高くスケーラビリティがあり，2.5節，2.6節で挙げた課題をすべて同時に解決している手法を見つけることができなかった．これに対し本研究では，既存の手法や関連研究を元に，それらの中で最善のものを選択し，さらにそれに新たな機能追加や改修を行うことによって課題をすべて解決することができた．性能も，単位時間あたり処理可能なリクエスト数に関して，実績のあるsuEXECと同等であり，実用に耐えうると考えられる．加えて，CGI実行方式では，インタプリタの複数バージョンを用意しておくことで，バージョンを切り替えて使えるというメリットもある．DSO実行方式はインタプリタをWebサーバプロセスに組み込むため，複数バージョンの提供は原理的に困難である．サービス仕様を検討する上では，CGI実行方式による複数バージョン提供が有用となる場合が多い．

本手法を採用することで，今後増えるであろうクラウドサービスや低価格ホスティング等に対応するために，共有ストレージを用いたスケールアウト型の負荷分散構成をとることができる．そして，コンピュータリソースの効率化や運用業務の省力化が促進され，最小限の人員リソースでの運用の仕組みが構築できると考えている．本章で提案したWebホスティング基盤は，今後のクラウドや低価格ホスティングを支えるWebサービスプラットフォームとしての，現時点における一つの答えであると考えている．

4 スレッド単位で高速に権限分離を行うWebサーバのアクセス制御アーキテクチャ

4.1. 緒言

単一のサーバプロセスで，複数のホストを仮想的に処理するマルチテナント型の仮想ホスト方式[70]を採用した場合に，ホスト間でファイルの閲覧や書き換えを防ぐためにアクセス制御が必要となる．これまで，Webサーバにおいて動的コンテンツを生成するプログラムを介したセキュリティ侵害から保護するためのアクセス制御手法としてsuEXEC[75]等が利用されてきた．さらに，Web APIの普及に伴ってWebサーバはプログラマブルな基盤になってきており，ホスティングサーバだけでなく，各システム連携もWeb APIを利用する状況で，Webサーバを実行する権限とコンテンツ生成のプログラムを実行する権限等，プログラムの役割によって明確に権限を区別し適切にアクセス制御を行う必要がある．それによって，セキュリティ上のリスクを低減し，また，同一権限による他システムへの干渉やプログラムの不具合による情報漏えいを減らすことができる．

2章でまとめた従来研究と，3章で提案した運用技術を考慮したアクセス制御アーキテクチャの提案によって，Webサーバ上でのプログラム実行方式の一つであるCGI実行方式[69]に対するアクセス制御で十分な権限分離がなされるといえる．しかし，CGI実行方式が，プログラム実行毎にプロセスを生成し，実行後にプロセスの破棄を必要とするものであるため，サーバプロセスが直接プログラムを実行する場合と比較して，性能が低くなることは否めない．CGIプロセスを仮想ホストの権限毎に起動させておく手法もあるが，高集積マルチテナントアーキテクチャのWebホスティングを想定した場合，ホストの数に依存したプロセスを事前に起動させておかなければならないためリソース効率が悪い．一方，サーバプロセスがプログラムを直接実行できるようにするために，サーバプロセスにインタプリタを直接組み込み，高速にプログラムを実行可能とするDSO実行方式[72]に対しては，2章で論じたように，汎用性が高く，安全で，性能劣化の少ないアクセス制御手法は存在しない．さらに，現状は，実行方式やインタプリタの種類によって複数のアクセス制御手法が存在し煩雑な状態になっているため，サーバ管理者（サービス提供者）にとって権限分離が扱いにくいものとなっている．この煩雑さゆえにしばしばサーバ管理者がアクセス制御の必要性を認識できていない場合も多い．

今後，Webサーバを高集積マルチテナントアーキテクチャに耐えうるシステムとして利用するためには，ハードウェアリソースを複数のシステムで共有する必要があ

る．そのためには，単一のサーバプロセスで複数のホストを処理する手法が必要となる．さらに，ホスティングシステムや複数のシステムが共存している環境において，DSO実行方式のようにプログラムを高速に動作させながらも，CGI実行方式で用意されているのと同等のアクセス制御が適用できることが求められる．また，サーバ管理者にWebサーバ上でアクセス制御を適用するように促すためには，より使いやすく簡潔で，CGIかDSOか等のプログラム実行方式に依存せず，性能劣化の少ないアクセス制御アーキテクチャでなければならない．

本章では，Webサーバにおいて，Webコンテンツの処理にLinuxのスレッドを用いて権限分離を行うアクセス制御アーキテクチャを提案する．Webコンテンツを処理する際にサーバプロセスにスレッドを生成させ，スレッド単位で権限を分離した上で，スレッド上でコンテンツの処理を行う．そのため，プログラム実行方式によらない統一的なアクセス制御アーキテクチャが実現可能となり，開発者が扱いやすく汎用性も高い．また，スレッドの生成，破棄は処理が軽量であるため，DSO実行方式を採用した場合でも性能劣化を少なくできる．

提案するアクセス制御アーキテクチャは，LinuxかつApache HTTP Server[16]（以降Apacheとする）上で動作することを前提としており，Apacheモジュールとして実装した．実装したApacheモジュールmod_process_security[10]は，オープンソースとして公開しており，既に数社で利用されている．

以降，本章の構成は以下の通りである．2章の従来研究にもとづいて，4.2節ではmod_process_securityのアーキテクチャについて説明する．4.3節で性能評価を行い，4.4節でまとめとする．

4.2. 提案するアクセス制御アーキテクチャ

2章のセキュリティとアクセス制御に関する考察および3章の提案にもとづいて，性能を重視した場合に必要となるアクセス制御アーキテクチャの要件をまとめる．単一のサーバプロセスで，多数のホストを高速に処理するためには，fork()システムコールを行うことで性能に限界のあるCGI実行方式ではなく，DSO実行方式でセキュリティを担保するためのアクセス制御アーキテクチャが必要である．suEXECのように別ファイルを，fork()システムコール後にexecve()システムコールを実行するようなアーキテクチャは，DSO実行方式においては性能に大きく影響を与えるため採用しない．また，サーバ管理者が扱いやすいアーキテクチャにするために，DSO実行方式の高速処理という特徴を生かしつつ，プログラム実行方式によらない統一的なア

[10] https://github.com/matsumotory/mod_process_security

クセス制御アーキテクチャである必要がある．さらに，システムへのアクセス制御適用方法がサーバ管理者にとって容易であることである．それらを満たすために，アクセス制御モジュールmod_process_securityを開発した．

4.2.1 Linuxスレッド単位で権限分離を行うアーキテクチャ

DSO実行方式の利点は，プログラムを高速に実行できることである．そのため，DSO実行方式のアクセス制御アーキテクチャを設計する上では，性能劣化を十分考慮しなければならない．3.4節の性能評価でも述べた通り，suEXECのようなプログラム実行時に新たに子プロセスを生成し，コンテンツ処理後にプロセスを破棄するアーキテクチャは，性能を大幅に低下させる．また，mod_ruid2のように，プロセスを生成せずにサーバプロセスに権限変更の特権を与えてプロセスを再利用すれば高速に実行できるが，2.6.3.2節で述べた通り，脆弱性が生じる．

そこで，mod_process_securityにおいては，Linux上で動作することを前提とし，Linuxにおけるスレッドをpthread_create()関数によって一時的に生成し，そのスレッド上で権限分離を行った後，スレッド配下でプログラムの処理を行い，最後にスレッドを破棄するアーキテクチャをとった．Linuxにおけるスレッドはプロセス内の同一メモリ空間上で実行でき，メモリ消費量等が軽減できる．また，Linux上では，スレッドの生成・破棄はプロセスの生成・破棄よりも処理が軽い[22]．スレッドの生成・破棄を利用することにより，サーバプロセスを破棄する必要も無い．

図4.1にDSO実行方式にmod_process_securityを適用した場合の，処理の流れを示す．

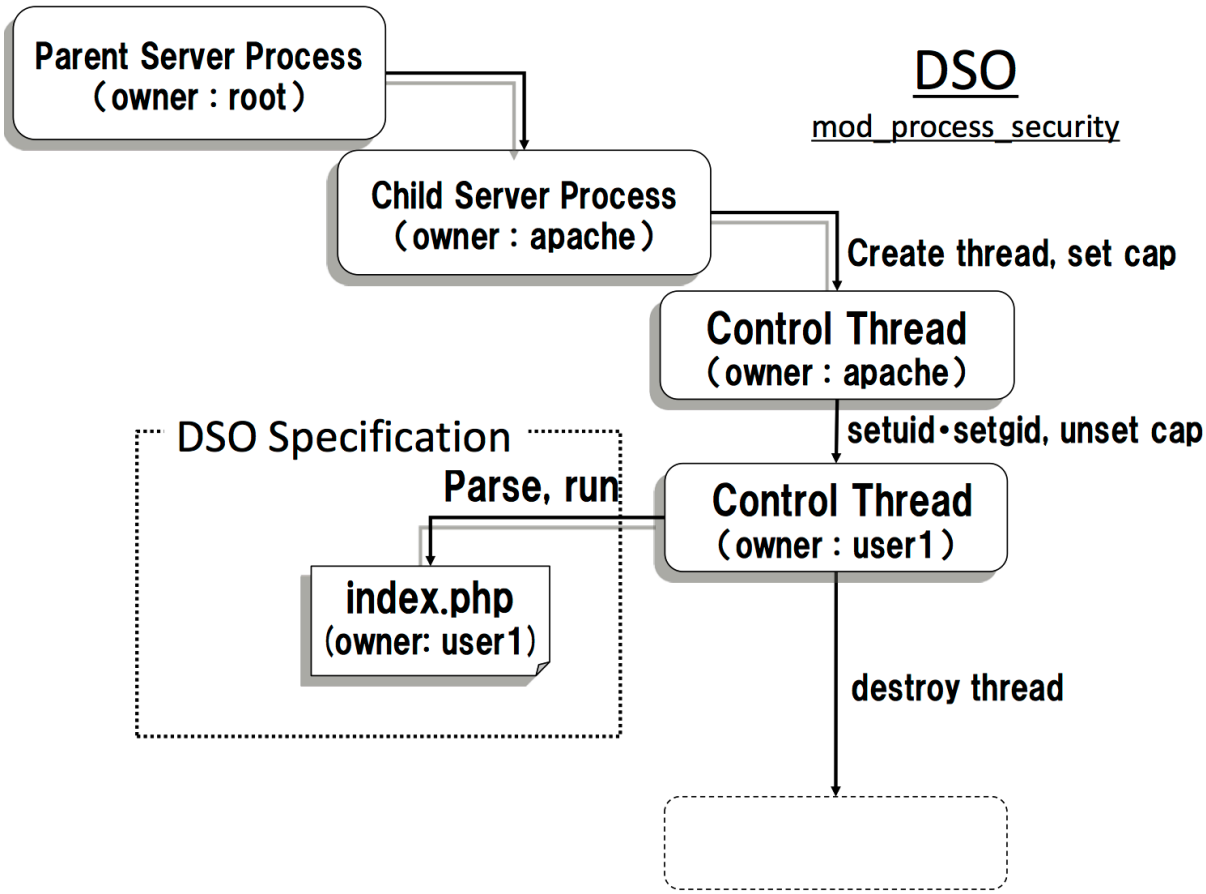

図 4.1　mod_process_securityのアクセス制御(DSO)

Linux上で動作するApacheは，親サーバプロセス（Parent Server Process）から事前にforkされた複数の子サーバプロセス（Child Server Process）がリクエストを受け付けるために待機している．リクエストを受け付けると，子サーバプロセス上で一時的にスレッド（Control Thread）を生成する．一時的に生成したスレッドに対し権限変更の特権であるLinux CapabilityのCAP_SETUID, CAP_SETGIDを付与する．この特権によって，スレッドは任意のuid，gidに権限変更可能となる．その後，3.2.4節で提案したアクセス制御アーキテクチャ同様に，実行対象のプログラムのuig，gid等の権限情報を動的に取得して，その権限にスレッドの権限変更を行う．スレッドの権限変更を行った後は，プログラムを実行する前にスレッドに付与された特権を破棄しておく．これによって，mod_ruid2で生じたような，プログラム経由での権限変更を防止する．スレッド上で直接プログラムを実行した後は，スレッドを破棄して，スレッドが属した子サーバプロセスは再度リクエスト受け付けに再利用される．

これによって，既存のDSO実行方式のアクセス制御アーキテクチャのように，サーバプロセスの生成破棄をすることなく，安全にアクセス制御を行える．また，スレッドの生成，破棄の処理時間の短さから性能劣化を低減し，DSO実行方式の特徴である高い性能を維持できる．性能評価に関しては4.3節で行う．

図4.2にCGI実行方式の場合のmod_process_securityのアーキテクチャを示す．

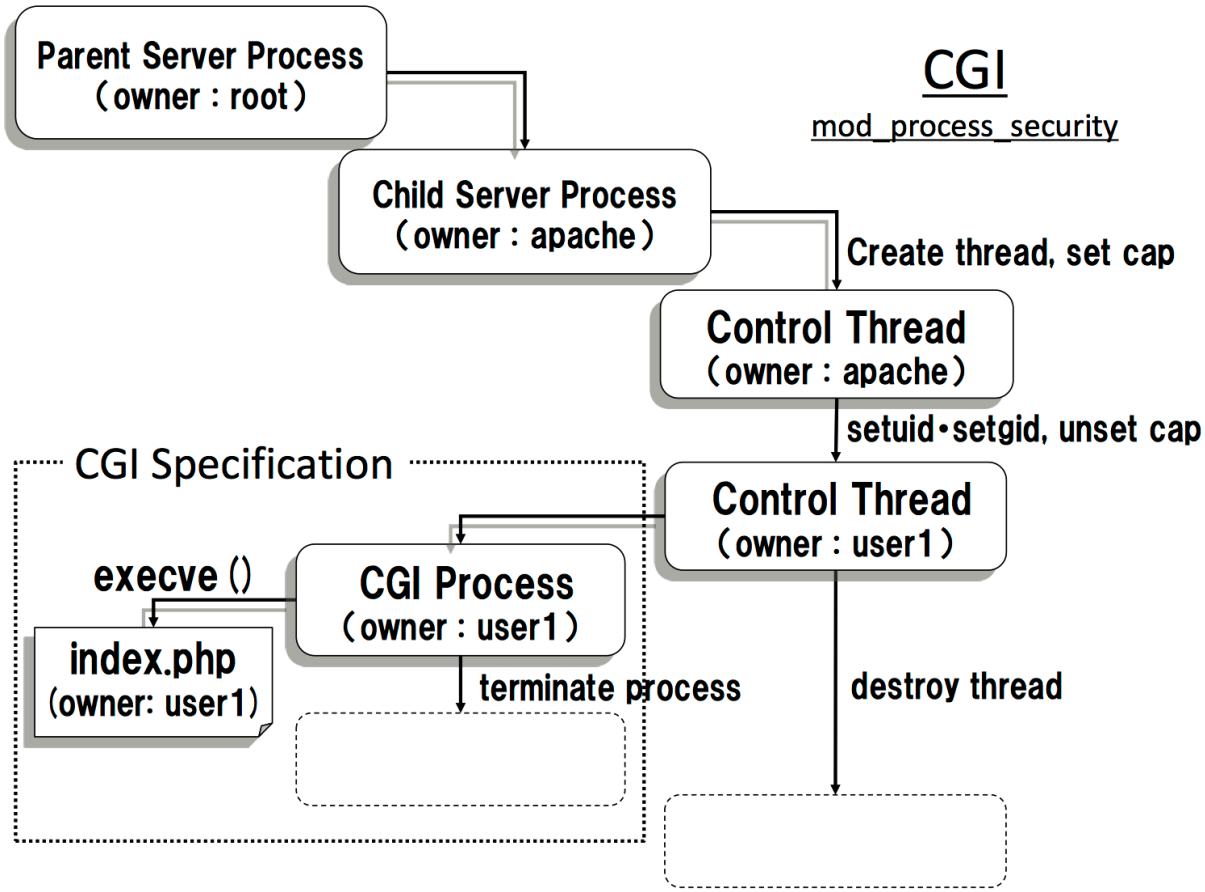

図 4.2　mod_process_securityのアクセス制御(CGI)

DSO実行方式とCGI実行方式が混在した環境であっても，一時的にスレッドを生成し，スレッドそのものの権限をプログラムの権限に変更する処理までのアーキテクチャは同じである．そのため，CGI実行方式の場合は，図4.2のようにスレッドからCGI用の一時プロセス（CGI Process）が生成され，プログラムがプログラムのファイルと同じ権限でexecve()システムコールにより実行されることになる．スレッド以降の処理は実行方式特有の仕様となるが，上位のスレッドで権限分離を行っているため，mod_process_securityでは実行方式を意識することなくアクセス制御を行うことができる．このように，mod_process_securityはスレッドを利用したアーキテクチャをとることで，プログラム実行方式によらないアクセス制御が実現可能となる．

4.2.2　リクエストされたプログラムの権限情報を取得

3.2.4節で提案した，リクエストのあったプログラムから動的に権限情報を取得する方式を採用により，新規顧客領域追加時や別権限のAPI群設置時において，Apacheプロセスの再読み込みや面倒な設定が必要無くなる．そのため，サーバ管理者にとって扱いやすくサービスの質が向上する．

mod_process_securityはApacheモジュールとして実装しているため，モジュールファイルを1つ組み込むだけで，アクセス制御が簡単に実現できる．図4.3に，すべてのファイルに対してアクセス制御行う場合の組み込み設定例を示す．サーバ管理者は，OSレベルで特別な設定等をすることなく，役割に沿ってプログラム群に適切に権限を設定し，図4.3の設定を書くだけで，プログラムの権限でプログラムが動作する．mod_process_securityはプログラムだけでなく静的コンテンツを含めたすべてのファイルに対して設定したり，任意の拡張子に対して設定したりすることができる．このように，DSOかCGIか等の実行方式の違いやインタプリタの種類によってアクセス制御手法を区別する必要が無くなり，非常に簡単にアクセス制御を適用することが可能となる．

以上のアーキテクチャによって，mod_process_securityを組み込めば，DSO実行方式に対応したインタプリタ（PHP，Perl，Python，Ruby等）はDSO実行方式を採用して，高速にプログラムを動作させ，その他シェルスクリプト等の場合はCGI実行方式で扱う，というように，プログラム実行方式を気にすることなく，より柔軟に安全なシステムを設計可能になる．

```
LoadModule process_security_module modules/mod_process_security.so
PSExAll On
```

図 4.3　mod_process_security設定例

4.3. 性能の測定と評価

mod_process_securityをアクセス制御に適用した場合の性能評価を行った．その際に，既存のアクセス制御手法を適用した場合との比較を行った．表4.1にテスト環境の詳細を示す．実験においては，一台のサーバ計算機を用意し，別の一台のクライアント計算機から通信を行うことで評価を行った．ベンチマークソフトはhttperf0.9.0[46]を利用し，クライアントにおいて1秒間に生成するリクエスト数を変動させ，サーバ側で1秒間に処理が完了したリクエスト数を計測した．使用する言

語は，CGI実行方式及びDSO実行方式両方で実行可能なPHPとし，実行方式の性能差を顕著にするために， PHPの設定情報をphpinfo()関数で表示するだけの簡易なコードで比較を行った．

表 4.1　mod_process_securityの性能評価のためのテスト環境

	Client Machine	Server Machine
CPU	Intel Core2Duo E8400 3.00GHz	Intel Xeon X5355 2.66GHz
Memory	4GB	8GB
NIC	Realtek RTL8111/8168B 1Gbps	Broadcom BCM5708 1Gbpz
OS	CentOS 5.6	CentOS 5.6
Middleware	-	Apache/2.2.3

また，実験対象の仮想ホストの数は，3.4節と同様の理由で，1ホストとした．本章で提案したmod_process_securityにおいては，仮想ホスト方式そのもののアーキテクチャは変更していないため，アクセス制御手法の性能差を比較する場合には，1ホストに対してのみを実験対象に行っても十分であると考えられる．

mod_process_securityをDSO実行方式とCGI実行方式それぞれに適用した場合の性能を評価する．評価には，アクセス制御手法を適用していない場合，既存のアクセス制御手法を適用した場合，mod_process_securityを適用した場合の3パターンについて比較を行う．グラフにおいて“nac”はアクセス制御手法を適用していない場合，“ps”はmod_process_securityを組み込んだ場合，“suEXEC”はsuEXECを組み込んだ場合，“ruid2_custom”は2.6.3.2節で述べたmod_ruid2の脆弱性を持った実装箇所を，サーバプロセスを破棄する方式に修正したモジュールを組み込んだ場合を示している．縦軸はサーバが1秒間に処理できたレスポンスの数，横軸はクライアントが1秒間に要求したリクエストの数を示している．

4.3.1 CGI実行方式に適用した場合

CGI実行方式の既存のアクセス制御手法はsuEXECを採用した．図4.4にCGI実行方式での評価結果を示す．CGI実行方式は，プログラム実行毎に必ず行うプロセスの生成，破棄に大きな処理時間を要する．そのため，アクセス制御適用の有無にかかわらず，

クライアントのリクエスト数を変動させていっても，図4.4において，CGI(nac)やCGI(suEXEC)と比較しても，CGI(ps)による大きな性能の差は見られなかった．また，同様の理由で，suEXECとmod_process_securityの性能差もほとんど見られなかった．また，アクセス制御を適用していないサーバが処理できたレスポンス数から，各種アクセス制御を適用した時のレスポンス数がどれだけ低下したかを，100リクエスト増加するごとに算出し，その平均値を性能劣化率として算出した．それぞれのアクセス制御を適用しない場合の性能に対して，アクセス制御適用後の性能劣化率は，suEXECの場合平均2.15%であったのに対し，mod_process_securityは平均1.48%であった．

4.3.2 DSO実行方式に適用した場合

DSO実行方式の既存のアクセス制御はmod_ruid2を採用した．ただし，mod_ruid2は通常の使い方においてroot昇格の脆弱性を持っているため，2.6.3.2節の図2.7で示した脆弱性を修正した方式で性能を計測した．図4.5にDSO実行方式での評価結果を示す．

脆弱性対応版mod_ruid2の性能はクライアントが1秒間に要求したリクエストの数の値に関わらず，1秒間に約4.5レスポンス程度と非常に低く，DSO実行方式にも関わらず，図4.4に示したCGI実行方式の性能のうち，横軸300から800の間で示している縦軸160前後の値よりも大きく低下した．2.6.3節で述べた通り，既存のDSO実行方式のアクセス制御アーキテクチャは，安全にアクセス制御を行うために，プログラム実行毎にサーバプロセスの生成，破棄を必要とする．サーバプロセスの生成，破棄はCGI実行方式におけるプログラム実行用のプロセス生成，破棄よりも処理に時間がかかるためだと考えられる．

mod_process_securityを組み込んだ場合は，アクセス制御を適用しない場合と比較してほとんど性能劣化は見られず，平均0.19%の性能劣化であり，DSO実行方式はCGI実行方式と比較して約10倍性能が高いことがわかった．既存のアクセス制御手法では，権限分離のためにCGIプロセスやサーバプロセスの生成，破棄が必要となる．その処理は，Apacheがリクエストのあったプログラムを実行してレスポンスを返す処理と比較し，無視できない程度に処理に時間を要したため，性能が大きく低下していた．一方，mod_process_securityでは，権限分離にスレッドの生成，破棄を利用しており，その処理はプロセスの生成，破棄と比較して非常に軽量である．そのため，リクエストを受けたApacheがプログラムを実行してレスポンスを返す処理と比較し，ほとんど無視できる程度にまでアクセス制御に必要な処理を低減できており，性能劣化が見られなかったと考えられる．

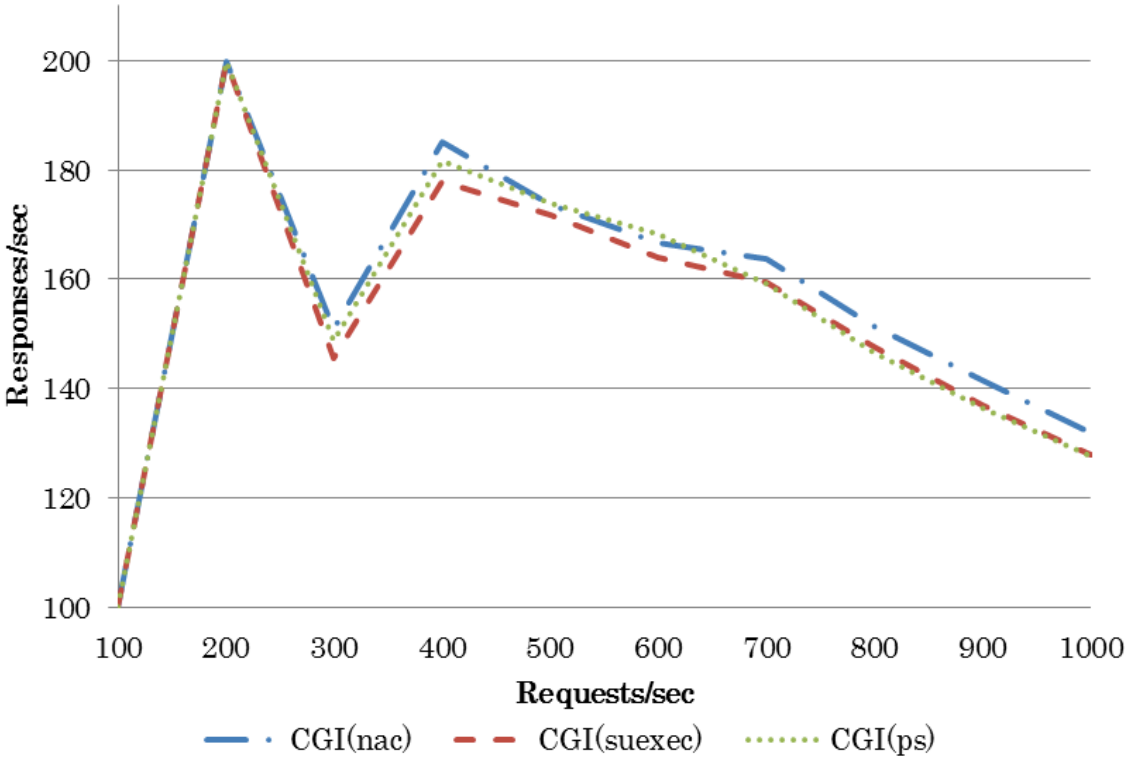

図 4.4　CGIのアクセス制御における性能比較

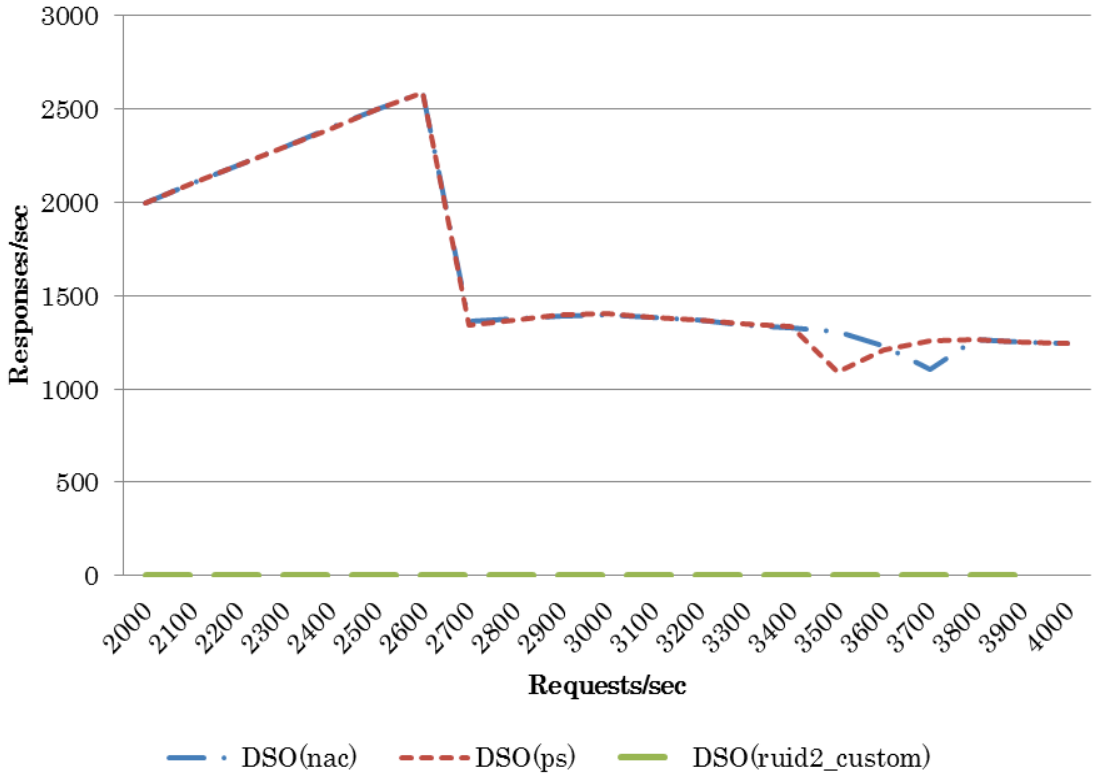

図 4.5　DSOのアクセス制御における性能比較

以上の性能評価より，mod_process_securityはDSO実行方式において，アクセス制御を使わない場合と比較して遜色ない性能を引き出すことができ，同時に，高集積マルチテナント環境において，カーネルの実装を変更することなく，コンテンツに脆弱性があったとしても他のホスト領域へセキュリティ侵害が起きないように権限分離を行うことができるはじめての手法だといえる．また，CGI実行方式においては，3章で提案したアクセス制御と同様，アクセス制御を使わない場合と遜色ない性能を引き出した上で，必要に応じてDSO実行方式だけでなくCGI実行方式に対してアクセス制御を適用でき，サーバ管理者が統一的に扱うことが出来るため，利便性が大幅に向上したと考えられる．

4.4. 結語

高集積マルチテナントを想定した仮想ホストを利用したWebサーバ上のアクセス制御において，既存手法では，DSO実行方式の権限分離を行えたとしても，CGI実行方式よりもはるかに劣る性能になっていたことが問題であったが，Linuxスレッド単位で権限を分離することにより，DSO実行方式でアクセス制御を使わない場合と比較しても遜色ない性能を引き出すことができた．また，実行方式によって区別されて考えられていたアクセス制御を，DSO実行方式に対応した上で，必要であれば同一の枠組みの中で，CGI実行方式でも分離できるような汎用性の高いアクセス制御アーキテクチャを実現できた．

広く使われているApacheのモジュールmod_process_securityとして実装し，これまでの一般的なApacheの設定や構成を大きく変えることなく導入できるように，リクエストのあったプログラムのファイルの権限から動的に権限分離を行えるようにした．そのため，導入も容易になっており，サーバ管理者が扱いやすい仕様になっている．ただし，性能を重視してDSO実行方式を採用することにより，CGI実行方式のメリットである，インタプリタの複数バージョンの利用可能性が失われる．

一般に，同一プロセス上のLinuxスレッドはメモリ領域を共有しているため，別のスレッドのメモリ領域を読み書き可能であるが，Linuxスレッドはスレッド単位でsetuid()およびsetgid()システムコールによって権限を変更できるため，その特性を利用して，少なくともファイルやプロセスのオーナの権限は分離できる．そのようなLinuxの使用にもとづいて，mod_process_securityは，Apacheのpreforkモデルが一つのリクエストに対して一つのプロセスを専有する仕様であるという前提のもと，リクエスト単位に一つのスレッドを生成して権限を分離し，同一プロセス内に複数のリクエスト処理が並行で動作しないように実装されている．また，Webサーバプロセスから生成されたスレッドがWebサーバプロセス自体のメモリ領域を読み

書きできる点についても，本来DSO実行方式はWebサーバプロセスが直接プログラムを実行する仕様になっており，通常Webサーバやインタプリタの実装によってプログラムからWebサーバのメモリ領域を直接操作できないようにしているため，Webサーバプロセスがスレッドを作ってからプログラムを実行することによるセキュリティレベルの低下は生じない.

5 スクリプト言語で高速かつ軽量に拡張可能なWebサーバの機能拡張支援アーキテクチャ

5.1 緒言

Webサービスを安定して提供するために，Webサーバソフトウェアそのものの内部機能の拡張が必要となる場合が多い．Webサーバソフトウェアの拡張を実装しやすくするため，Apacheではプラグイン形式の機能拡張機構を採用している[71]．Webサーバソフトウェアの内部機能を拡張することにより，Webコンテンツ処理前に，リソース制御やアクセス制御等の緻密な制御を実現できる[25]．2.4.3節で述べたように，Webサーバの拡張は，高速かつ軽量に動作することを重視してC言語による実装が主流であったが，生産性や保守性を顧慮して，近年はスクリプト言語で機能拡張を行う手法が提供されている[36][60][33]．しかし，従来手法のmod_perl[36]やmod_ruby[60]は，高速に処理するためにインタプリタを複数のスクリプトで共有するが，アプリケーションとして全く別々のスクリプト間でもグローバル変数名が干渉し合い，拡張機能を安全に実装できない．また，PerlやRubyのインタプリタやライブラリが巨大であるため，Webサーバそのものの内部拡張を高速にかつ省メモリで処理するには適していない．

The Apache Software Foundationが開発しているmod_lua[67]は，Lua[55]が高速かつ軽量な組み込みスクリプト言語であることを利用して，スクリプト実行毎にインタプリタを用意することにより安全に実装できるようにしている．しかし，スクリプト実行毎にインタプリタの読み込み・解放とライブラリの読み込みが必要となり，C言語による実装に比べれば依然として処理効率が悪い．そこで，提案手法ではWebサーバプロセスから内部処理としてスクリプトが呼び出された際，高速に処理するために，インタプリタやスクリプトに関する情報を保存しておく領域（以降，状態遷移保存領域とする）を，サーバプロセス起動時のみに生成しておいて，それを複数のスクリプトで共有し実行するアーキテクチャをとった．また，メモリ増加量を低減するために，スクリプト実行後に状態遷移保存領域からメモリ増加の大きな原因となるバイトコードのみを解放するようにした．さらに，状態遷移保存領域を共有することにより生じるスクリプト間のグローバル変数や例外処理の干渉を防止するために，スクリプト実行後は，状態遷移保存領域から任意のグローバル変数・例外フラグを解放できるようにした．このアーキテクチャにより，稼働し続けることが前提のサーバプロセスに対し，高速性・省メモリ・安全性の観点から最適化し

てインタプリタを組み込み，スクリプト言語でも効率よく内部機能拡張を実装できる．

提案する機能拡張支援アーキテクチャを実現するために，Apacheに組み込みスクリプト言語mruby[47]を組み込むことにより，Apacheの内部機能を，Rubyスクリプトによって容易に拡張できるように実装した．このApacheの機能拡張支援機構をmod_mruby[11]と名付けた．mod_mrubyは，Rubyスクリプトで内部機能を拡張できるように，C言語と連携が容易な組み込みスクリプト言語mrubyの特性を生かして，Apacheの内部処理やデータを制御するRubyメソッドが実装されている．また，mod_mrubyの実装にあたり，アプリケーションにmrubyを組み込むために必要な機能が不足していたため，開発を通して機能の提案や不具合をフィードバックし，mrubyに沢山の貢献を行った[12]．

mod_mrubyでは，Rubyスクリプトを変更することで，即時にApacheの内部機能が変更や拡張可能となり，コンパイルやApacheの再起動を必要としない．また，さらに高速に処理したい場合は，都度Rubyスクリプトを呼び出さずに，サーバ起動時にバイトコードまではコンパイルしておいて，処理を呼び出された時にバイトコードから実行することも可能である．バイトコードから実行するアーキテクチャにより，C言語でApacheモジュールを実装した場合と遜色ない性能が得られる．

さらに，Apacheと並ぶ代表的なWebサーバソフトウェアであるnginx[50]にも，提案するアーキテクチャをngx_mruby[13]として実装した．これにより，少なくとも共通している機能において，複数のWebサーバソフトウェアの違いを意識することなく，Webサーバの機能拡張を実装できるようにしている．

本章の構成について述べる．5.2節ではmod_mrubyのアーキテクチャと適用例について説明する．5.3節でRubyスクリプトをApacheの内部機能として実行した場合の性能評価を行い，5.4節でまとめとする．

5.2 mod_mrubyのアーキテクチャと適用例

Webサーバの実装や拡張機能追加は，2.4.3節で述べたように，C言語での開発が主流であり，かつ，Webサーバソフトウェアの内部仕様を詳細に理解している必要があった．スクリプト言語の普及に伴い，Webサーバソフトウェアをスクリプト言語で拡張する手法も提案されてきたが，これまでの手法は，生産性や保守性を優先した場合

---

[11] https://github.com/matsumotory/mod_mruby

[12] https://github.com/mruby/mruby/pulls?q=is%3Apr+is%3Aclosed+author%3Amatsumotory 本研究を通したmrubyへのフィードバック一覧．

[13] https://github.com/matsumotory/ngx_mruby

に，リクエスト毎に処理されるWebサーバの内部処理をスクリプトで実装するには，高速性，省メモリ，および，スクリプト間のグローバル変数の干渉の防止を同時に満たせていない．

本研究では，2.3.2節で言及したように，現状のWebサーバソフトウェア拡張における問題が組み込みソフトウェア開発における問題と共通であると捉え，その解決策として，スクリプト言語で機能拡張可能で，高速にかつ省メモリで動作し，スクリプト間で干渉しあわないWebサーバの機能拡張支援機構mod_mrubyを提案する．

5.2.1 mod_mrubyのアーキテクチャ概要

mod_mrubyは，高速性と省メモリの観点からWebサーバプロセスに軽量に動作する組み込みスクリプト言語mrubyのインタプリタを組み込み，その上で，スクリプトが互いに影響を与えないように分離するアーキテクチャを取った．また，本章のアーキテクチャはWebアプリケーションではなく，Webサーバソフトウェアそのものの内部処理を制御することを主目的としている．2.4.3節で述べたように，Webアプリケーションの実装を主目的とする場合は，言語の軽量さよりも，ライブラリや実装記述方法が充実していることの方が重要であり，そのような用途においては，現行のRuby on Rails[57]やmod_ruby[60]，mod_php[51]，mod_python[20]およびJava Servlet[31]等の方が適している．mod_mrubyでは，mrubyの特徴であるモジュールやクラスの取り外しによって不必要な機能を省略し，mrubyの組み込みAPIを利用することで，生産性や保守性を優先しながらも，省メモリなWebサーバ拡張機構を提供できる．

mod_mrubyは，Webサーバ管理者が迅速にリソース管理やアクセス制御等の機能拡張を実装できるように，予めApacheとRubyスクリプトの処理を連携するインターフェイスをApacheモジュールとして実装し，組み込んでおく．図5.1に，mod_mrubyの仕組みを示す．

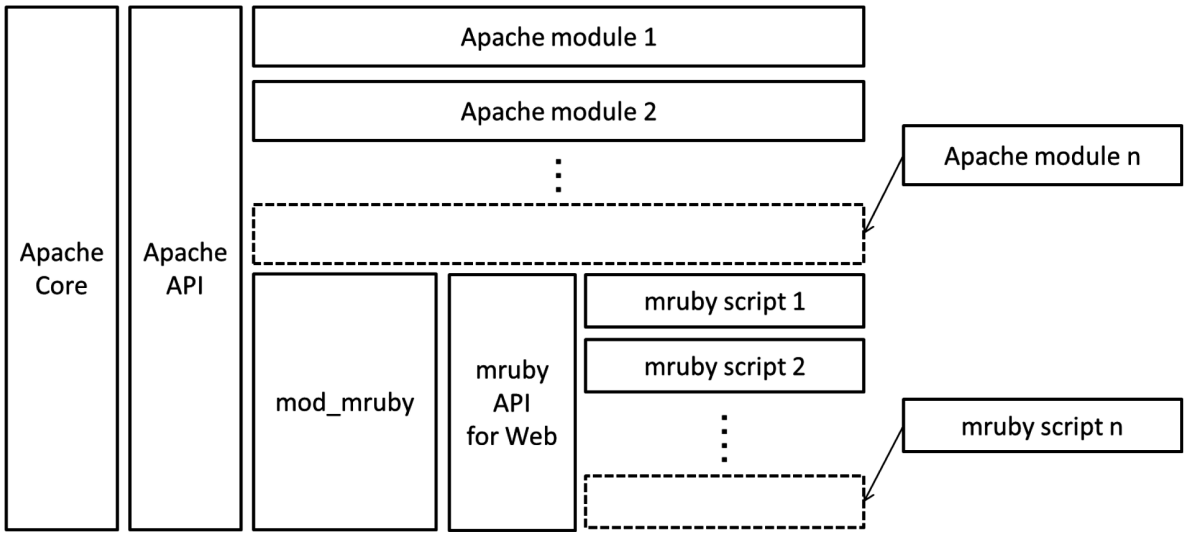

図 5.1　ApacheとMod_mrubyの仕組み

mod_mrubyをApacheに組み込むことによって，Apacheの内部処理をRubyスクリプトとして実装できる．mrubyとApacheは専用のRubyメソッドを介して連携する．mruby上で記述できないような複雑な処理の場合は，ApacheモジュールとしてC言語で実装し，Rubyスクリプトと共存しながら組み込むことも可能である．Rubyスクリプト内の処理の変更は，スクリプトを書き換えることで，以降のリクエストが書き換え完了後のスクリプトで処理される．また，スクリプトの変更がそれほど多く必要とされない状況では，リクエストの都度Rubyスクリプトを呼び出す代わりに，サーバ起動時にバイトコードまではコンパイルしておくことも可能である．

以降，高速性および省メモリとスクリプト間の干渉の観点からmod_mrubyのアーキテクチャの詳細を述べる．

5.2.2 状態遷移保存領域の再利用

mod_luaは，Apacheの内部処理をLuaスクリプトで実装できるため，スクリプトとしての保守性や開発効率を生かしたまま，Luaの特徴を活かして，高速かつ省メモリにApacheの内部処理を実装することが可能となる．図5.2にLuaスクリプト実行時のmod_luaの実行フローを示す．

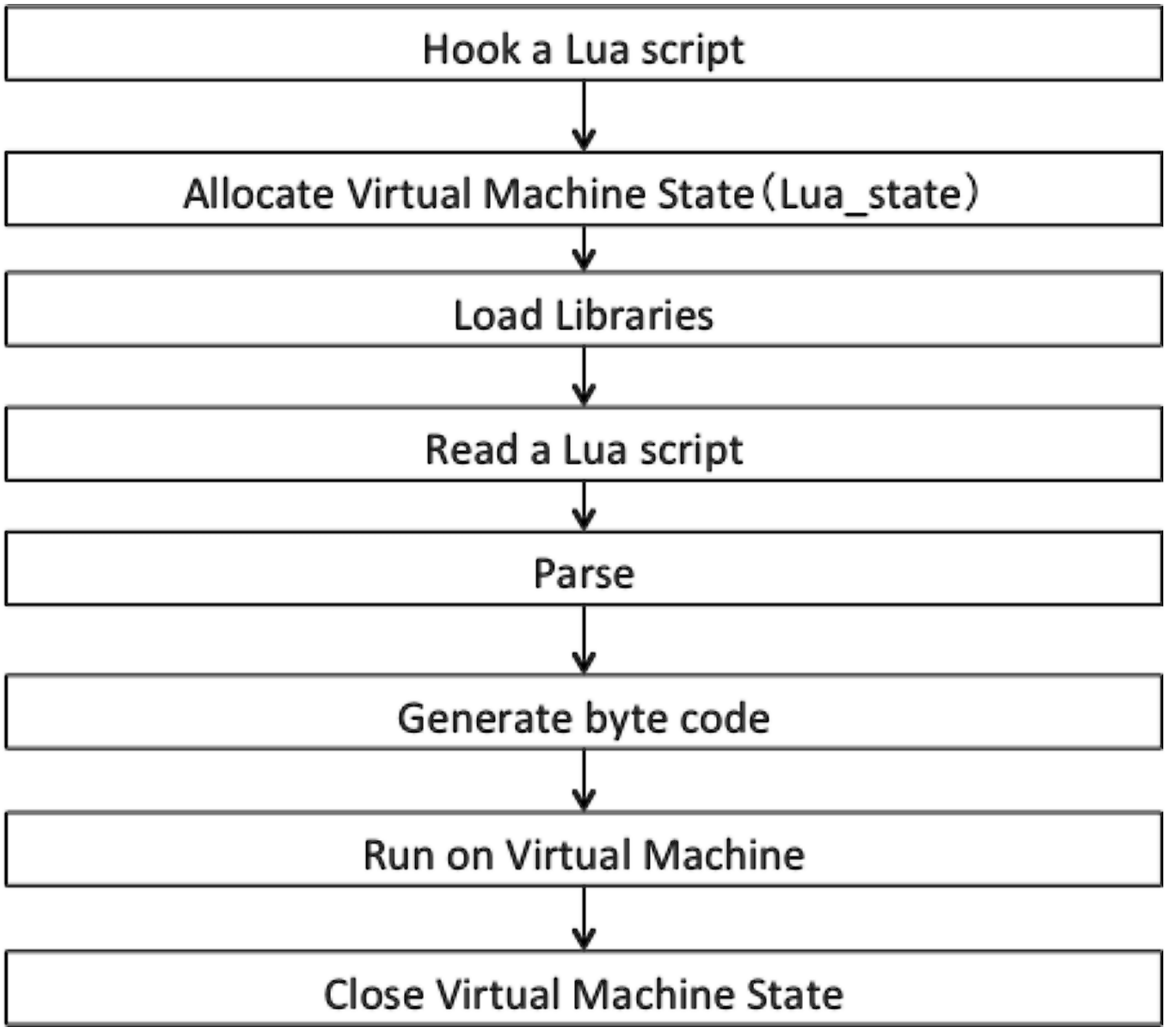

図 5.2　mod_luaのアーキテクチャ

5.1節で述べた通り，mod_perl[4][38]やmod_rubyは，複数のスクリプトで同一の状態遷移保存領域を使用するため，別々のアプリケーション間で各スクリプトのグローバル変数を共有してしまう問題があった．しかし，mod_luaでは，それらの問題を回避するために，性能を犠牲にして，スクリプトを実行する毎に状態遷移保存領域を生成し，実行後は状態遷移保存領域を解放するアーキテクチャをとっている．これに対し，mod_mrubyは，状態遷移保存領域を複数のスクリプトで共有することで，高速処理できるアーキテクチャをとった．状態遷移保存領域には，構文解析やインタプリタが持つバーチャルマシン上で動作するバイトコードの生成，その他，バーチャルマシンそのものや変数を保存するハッシュテーブル等，スクリプト実行に必要な情報のほとんどが保存されている．そのため，その領域の確保処理は非常に高コストである．図5.3にmod_mrubyの実行フローを示す．

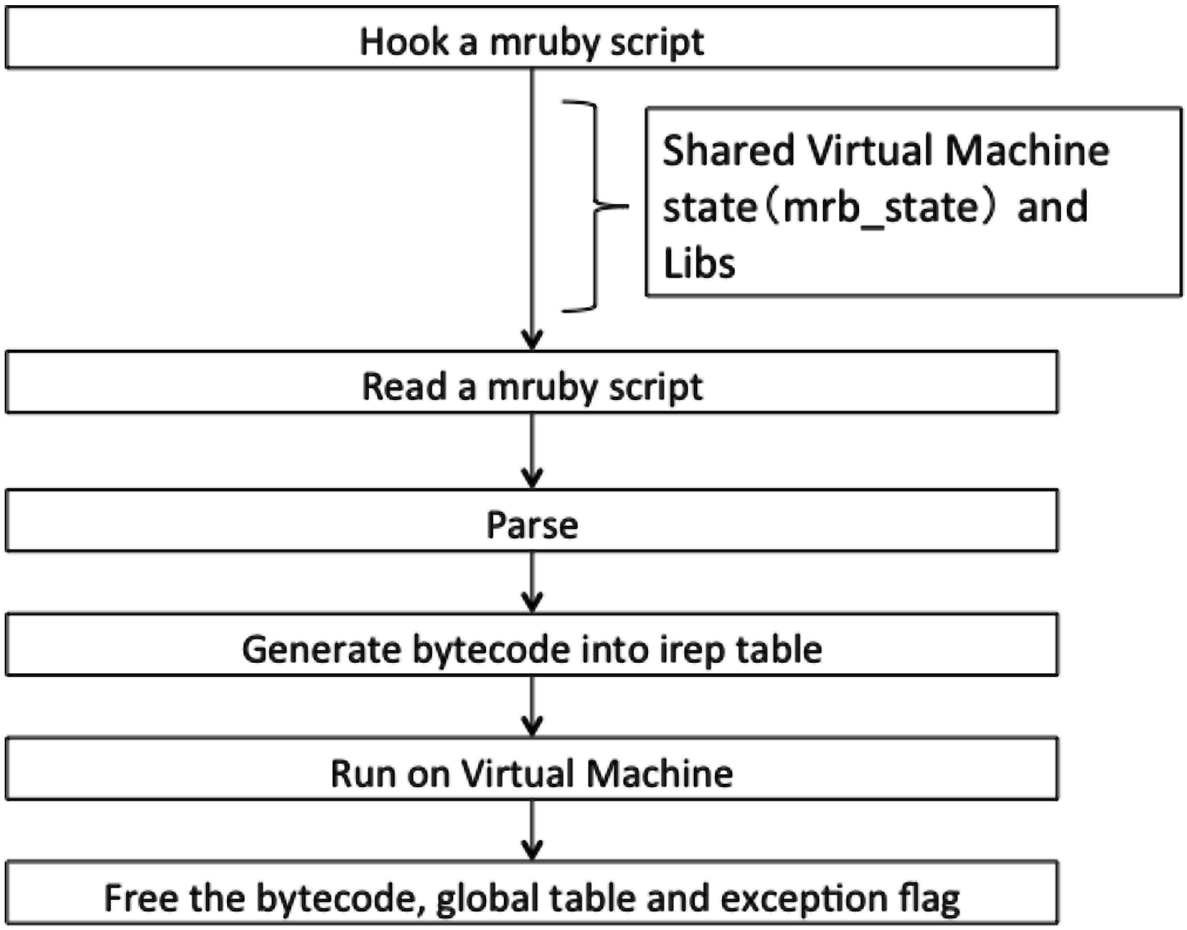

図 5.3　mod_mrubyのアーキテクチャ

リクエスト処理時に，処理コストの高い状態遷移保存領域の再確保やライブラリの読み込み等を実行しないようにするために，Apacheのサーバプロセス起動時に，状態遷移保存領域をあらかじめ生成し，ライブラリを読み込んでおく．そして，Apache起動後，クライアントからリクエストがあると，図5.3のように，リクエストの各種処理フェーズでフックされているRubyスクリプトがApacheから呼び出され実行される．そこで，事前に生成されている状態遷移保存領域を呼び出して，その領域上でスクリプトの処理を開始する．リクエスト処理毎に，Rubyスクリプト自体の構文木解析を行い，バイトコードを生成する．そのバイトコードは，状態遷移保存領域内に存在するバイトコードテーブルに保存される．その結果，スクリプト単位で処理コストの高い状態遷移保存領域の確保やライブラリの読み込みが必要であるmod_luaと比較して，高速に処理を行うことができる．このアーキテクチャをmrubyで実現するためには，mrubyの持つ機能だけでは実現が困難であった[14]が，mruby本体にバイ

[14] 2013年3月6日のcommitで取り込まれた． https://github.com/mruby/mruby/commit/2e95f638cd01a92a83f3808d745f69f46b64db10

トコードテーブルからバイトコードを簡単に取り出すAPIや，不必要になったバイトコードを安全に解放するAPI等を実装してフィードバックし，mruby本体にも取り込まれた．また，リクエスト毎にコンパイルするのではなく，バイトコードをWebサーバ起動時にコンパイルしてバイトコードまで生成しておき，リクエスト時にバイトコードを直接実行することで，より高速に動作する．Webサーバプロセス起動中に，サーバ管理者がRubyスクリプトの変更を必要としない場合は，このバイトコードキャッシュ機能を使えるように実装した．

5.2.3 メモリ効率とスクリプト間の干渉の改善

上記の状態遷移保存領域を再利用するアーキテクチャをとった場合の問題を解決する方法を述べる．mrubyは，バイトコード生成前後でメモリ消費量が大きく増加する．メモリ消費量が肥大化する主な原因は，コンパイルによって生成されたバイトコードを状態遷移保存領域内のバイトコード保存領域（以降irepテーブルと呼ぶ）に保存し，次に新たにスクリプトを実行する際には，irepテーブルのインデックス番号をインクリメントして，新しいインデックス番号が指し示す領域にバイトコードを保存していたためであった．mod_mrubyでは，Apache起動中であってもフックしているRubyスクリプトの内容を変更することで，振る舞いを次のリクエストから変更することができるように，リクエスト毎にRubyスクリプトをコンパイルしてバイトコードを新しく生成する．つまり，過去のコンパイル済みの古いバイトコードをirepテーブルに保存しておく必要は無い．その特徴を考慮して，スクリプト実行終了後に自動的にバイトコードを解放するようにした．バイトコード生成後，irepテーブルに保存する際にそのバイトコードを保存する領域のインデックス番号を保存しておく．バイトコードを実行した後は，状態遷移保存領域上のirepテーブルから，保存しておいたインデックス番号を元に対象のバイトコードを解放し，インクリメントされた数だけデクリメントするようにした．この処理をmrubyの組み込みAPIに追加することで状態遷移保存領域を再利用しながらも，スクリプト実行前後に増加していたプロセスのメモリ消費量を大幅に低減することができる．また，mruby自体が機器組み込みのために使われていたため，アプリケーションへの組み込み用途では機能が不足していた．mrubyの組み込みAPIだけでなく，本研究でその他数多くのフィードバックを行い，mrubyにコードが取り込まれた．

状態遷移保存領域を共有することによるスクリプト間の干渉問題については，以下のような実装で回避した．状態遷移保存領域内には，例外処理フラグとグローバル変数テーブルが存在する．これらが，スクリプト干渉における主な原因となるため，スクリプト実行終了時にはバイトコードに加えて例外処理フラグも解放し，さ

らにスクリプト中においてグローバル変数テーブルから任意のグローバル変数を解放できるようにする．これによって，新たなRubyスクリプトが同一の状態遷移保存領域上で処理されても，それ以前に実行されていた古いスクリプトの情報と干渉が起きないようにできる．また，Webサーバの特性上，マルチスレッドによるリクエスト処理等を考慮して，バイトコードやグローバル変数テーブルの保存と解放の間を排他処理することにより，インデックス番号の齟齬が起きないようにした．グローバル変数の解放は，mod_rubyや既存手法からの移行を考慮して，プログラマが手動で記述することにより行われるように実装した．図5.4にグローバル変数$gvを解放するための記述例を示す．

```
$gv = 1

Apache::global_remove :$gv
```

図 5.4　グローバル変数解放の記述例

平易な記述でグローバル変数を解放することができるため，記述コストは大きく増えない．なお，グローバル変数の衝突を完全に防ぐためには，全てのスクリプト上において，使用したグローバル変数を解放する上記処理を追加する必要がある．2.4.3節のWebサーバの機能拡張の従来研究で言及した通り，状態遷移保存領域には，バーチャルマシンに必要な情報や，その他，シンボルや変数のための多くのハッシュテーブルが存在する．グローバル変数用のハッシュテーブルはそれらの一つに過ぎず，さらにはグローバル変数の解放はテーブル内の特定のカラムを解放するのみの処理となるため，状態遷移保存領域全体を解放するのと比較した場合には相対的に非常に小さい処理となり，性能上の問題は生じず状態遷移保存領域の共有の利点には影響ない．一方で，スクリプト間で変数を受け渡ししたい状況，例えば，サーバ起動時にデータベースサーバに接続しておいて，その接続情報を別のフェーズで再接続することなく再利用したい場合に，それらの情報を，mod_mrubyの機能によりグローバル変数を介して安全に受け渡しすることも可能である．

グローバル変数の解放は，開発者が手動で記述する必要があるため，現実的に適切にグローバル変数の共有を完全に避けることは難しい．特定のグローバル変数を実行毎に自動的に解放することも原理的には可能であるので，グローバル変数の自動解放機能を追加する予定である．

5.2.4 mod_mrubyのクラスとメソッドの仕様

mod_mrubyによってApache内部の処理をRubyスクリプトで実装するために，Ruby上からApache内部の関数や構造体を操作できるクラスを設計した．これによって，Rubyスクリプト上から様々なWebサーバソフトウェア拡張が可能となる．図5.5と図5.6に，Rubyによる機能拡張実装例を示す．図5.5は，Apacheがリクエストを受けた際に，アクセスのあったURIとファイルを紐づけるフェーズで，Rubyスクリプトをフックするための Apacheの設定例である． 図5.6は，アクセスのあったURIと特定のファイルを紐づけて，レスポンスにそのファイルの内容を返す処理をRubyで実装した例である．

また，Apacheと並ぶ代表的なWebサーバソフトウェアであるnginx[50]にも，提案するアーキテクチャをngx_mrubyとして実装した．他のWebサーバソフトウェアにも同様の機能を実装予定である．これにより，Webサービス開発の技術者は，Apacheやnginx等，共通の処理については複数のWebサーバソフトウェアの実装の違いを意識することなく，Webサーバの機能拡張を実装できる．

```
LoadModule mruby_module modules/mod_mruby.so
mrubyTranslateNameMiddle /path/to/sample.rb
```

図 5.5　mod_mrubyのApache設定例

```
r = Apache::Request.new

r.filename = "/path/to/redirect.html"

Apache.return Apache::OK
```

図 5.6　Rubyスクリプト例

5.3 性能評価

mod_mrubyが実用に耐えうるかを評価するために，Apacheの内部処理を実装したRubyスクリプトを，mod_mrubyを介して実行した場合の性能を評価した．

5.3.1 サーバプロセスのメモリの増加量に関する評価

mod_mrubyは，高速性を得るために，状態遷移保存領域を共有しながらも，メモリが増加しないようにバイトコードのみを解放するようにしたアーキテクチャをとった．そのアーキテクチャによるメモリ増加量の低減を確認するため，Apacheにmod_mrubyを組み込み，図5.5と図5.6で示す処理を行うようにした．そして，バイトコードを解放しない場合と，解放する場合において，それぞれ5万リクエスト処理するまでの，サーバプロセスのメモリ増加量を複数回測定し，その平均値を算出した．図5.7にメモリ増加のグラフを示す．図5.7より，バイトコードの解放をしない場合は，リクエスト処理毎にApacheの内部でRubyスクリプトが実行され，そのバイトコードが状態遷移保存領域に保存されてしまい，サーバプロセスの消費メモリが単調増加していることが分かる．それを防ぐために，バイトコードのみを解放することで，サーバプロセスのメモリが増加していないことが分かる．これにより，バイトコードのみを解放するアーキテクチャで，メモリ使用量が増加しないことが分かった．また，図5.7に示すようにApacheサーバプロセスのメモリ使用量は約5000kBであるが，mod_mrubyを組み込む前のApacheのサーバプロセスのメモリ利用量が約4400kBであり，mod_mruby組み込みに要する最小限のRubyスクリプトをフックした場合のメモリ使用量は約600kBであった．ただし，Rubyスクリプトの実装内容によってメモリ使用量は大きく変わる．

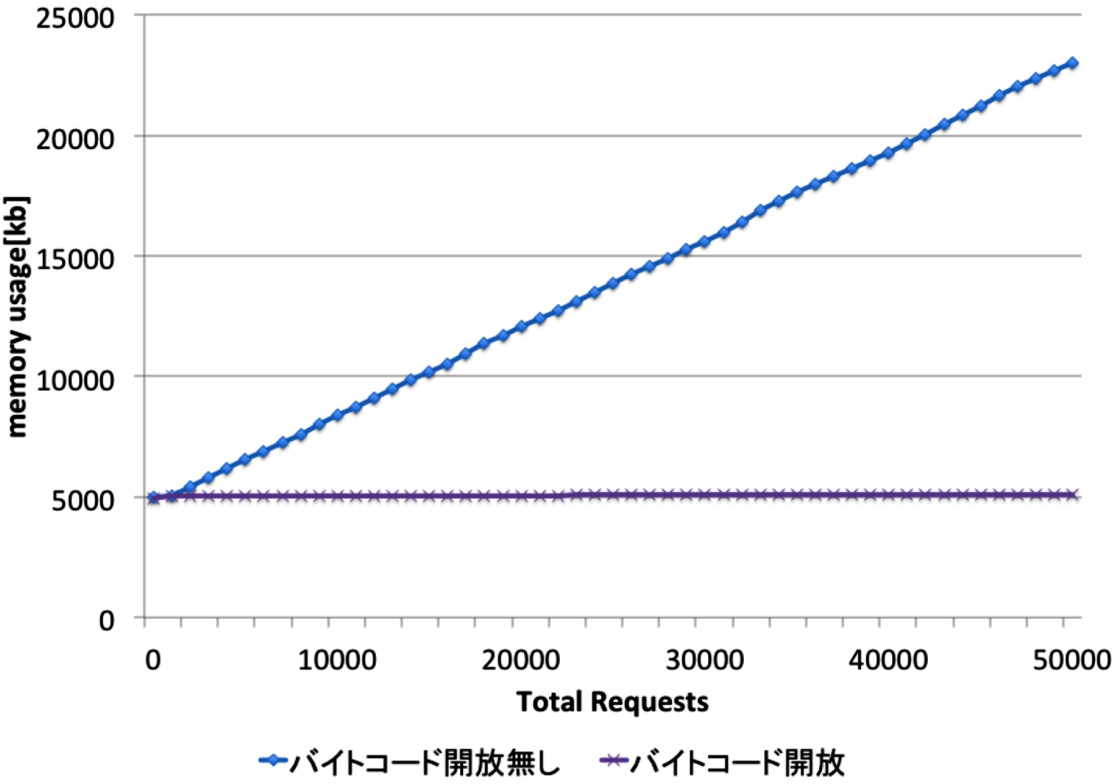

図 5.7 バイトコード解放によるメモリ増加量

5.3.2 高速性に関する評価

次に，既存の機能拡張手法と高速性の比較を行った．状態遷移保存領域が作成され，コンパイルされることによる処理の影響を最大化するため，実行するスクリプトはクライアントがどのようなURIにアクセスしても，全てのアクセスに対してhello worldの文字列を返すようなコストの低い処理を実装したものとした．評価においては，同様の処理を従来のようにC言語によって実装したApacheモジュールmod_helloと，mod_perlによるPerlスクリプトでの機能拡張実装，mod_rubyによるRubyスクリプトでの実装，mod_luaによるLuaスクリプトでの実装との比較を行った．表5.1に，評価に用いたクライアントならびにサーバマシンの性能を示す．

表 5.1　mod_mrubyの性能評価のためのテスト環境

	Client Machine	Server Machine
CPU	Intel Core2Duo E8400 3.00GHz	Intel Xeon X5355 2.66GHz
Memory	4GB	8GB
NIC	Realtek RTL8111/8168B 1Gbps	Broadcom BCM5708 1Gbps
OS	CentOS 5.6	CentOS 5.6
Middleware	-	Apache/2.2.3

クライアントマシンからサーバマシンに対して，abコマンドにより同時接続数100，総接続数10万のパラメータでリクエストを送信し，サーバマシンが1秒間に処理できたレスポンス数を複数回計測しその平均値を算出した．同時接続数のパラメータは，サーバのApacheやOSのチューニングがボトルネックにならないように，予備実験により適切だと思われるパラメータを設定した．

表5.2は1秒間に処理できたレスポンス数（responses/sec）の結果を有効数字6桁により示している．

表 5.2　Webサーバの機能拡張手法の性能評価と各手法の特徴

	Programing Language	Response/sec
mod_hello	C	9861.17
mod_perl	Perl	3346.38
mod_ruby	CRuby	4769.04
mod_lua	Lua	5209.11
mod_mruby	mruby	9021.54
mod_mruby byte-code cache	mruby	9752.37

性能評価の結果，C言語により機能拡張したmod_helloが9861.17　response/secであるのに対して，RubyスクリプトにApacheの内部処理を実装し，mod_mrubyを介して実

行しても，9021.54 response/secであり，さらにmod_mrubyにおいてバイトコードをキャッシュした場合には，9752.37 response/secと，C言語による機能拡張にも遜色ない結果が得られた．

mod_perlやmod_rubyは，mod_mrubyと同様リクエストごとに状態遷移保存領域を解放せず初期化して再利用するアーキテクチャをとっていながら，mod_mrubyよりも性能が低い．これは，mod_perlやmod_rubyで使われているインタプリタは非常に高機能で規模が大きく（数MByteのバイナリサイズ），mrubyインタプリタ（数百Kbyte）と比較して多くのクラスやメソッドを使用可能であり，構文解析やバイトコードの生成時（以降コンパイルとする）のクラスやメソッドの検索のコストが非常に大きくなるためである．C言語で書かれたWebサーバの実装の一部をスクリプト記述で代替するために，CRubyやPerlのような高機能で規模の大きいインタプリタを組み込むのは，リクエスト毎にスクリプトのコンパイルが必要となることを考えると非常にコストが高い．そのため，mod_rubyやmod_perlは状態遷移保存領域を共有しているが，コンパイル処理がボトルネックととなり，大幅に性能が低下する．

mod_rubyがmod_luaよりも性能が低い理由は，CRuby程度のインタプリタの規模になった場合，状態遷移保存領域の確保・解放よりも，コンパイル時の処理がボトルネックとなるためと考えられる．一方で，mod_luaのインタプリタの規模は，mod_mrubyと同等かやや小さいものであり，インタプリタの規模が同等の状況においては状態遷移保存領域を共有するかどうかが性能に大きく影響を与えている．そのため，mod_mrubyがmod_luaよりも高速に動作する．比較として，mod_mrubyの実装をリクエスト毎に状態遷移保存領域の確保を行うように変更して同様の実験を行った場合，約2000 response/sec程度の値しか出なかった．このことからも状態遷移保存領域の確保と解放のオーバーヘッドが大きいことがわかる．

以上より，hello world文字列を出力するのみの軽量な処理においては，mod_mrubyのアーキテクチャによって，インタプリタの処理部分は動的コンテンツを扱う上でほとんどボトルネックにならないと想定される．スクリプトの保守性や開発のし易さを考慮した場合，サーバ拡張のための現時点で最良のの選択肢になり得ると考えられる．

5.4 結語

本章では，Webサービスの大規模・複雑化を考慮して，Webサーバの機能拡張を支援するために，スクリプト言語で機能拡張が可能でありながら，高速かつ省メモリで動作するWebサーバの機能拡張支援機構mod_mrubyを提案した．また，高速性を優先した場合に生じる，スクリプト間でグローバル変数が干渉し合う問題を同時に解

決した．mod_mrubyによって，Rubyスクリプト上にApacheの内部処理を記述することで，Webサーバプロセスを再起動することなく，書き換えた以降のリクエストから処理の拡張が適用される．その結果，Webサービス開発に関わる技術者がWebアプリケーション等をRubyで開発する延長で，容易にWebサーバ自体を拡張できるようになると考えている．また，nginxをmrubyによって機能拡張できるngx_mrubyにより，共通の処理については，ApacheやnginxのWebサーバソフトウェアの実装の違いをRubyスクリプトで吸収しながら機能拡張が可能になる．

6 リクエスト単位で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

6.1 緒言

2.4.2節で高集積マルチテナントアーキテクチャについて言及したように，個人利用向けのホスティングサービスを提供するにあたっては，サービスの提供価格を月額数百円程度に下げるために，一つのサーバに数万から最大10万サイト程度を収容するような，高集積Webホスティングサービスの基盤構築が要求される．Webサーバに効率よく高集積に複数のホストを収容するためには，プロセス数がホスト数に依存しないように，単一のサーバプロセスで複数のホストを処理する必要がある．そのような環境で，大量のリクエストを安定して処理するためには，コンピュータリソースを多く消費するリクエストがあった場合，その処理ができるだけ他のリクエスト処理に影響を与えないように制限する必要がある．さらには，管理者がそのようなリソース制御ルールを自由に記述することができれば，多種多様なリソース制御の問題を解決することができる．

2.7節で述べたように，既存のWebサーバの一般的なリソース制御[12][73]は，リクエストの同時接続数や単位時間当たりのリクエスト数が指定の閾値を超えた場合や，管理者があらかじめ指定したCPU使用時間やメモリ使用量の閾値を超えた場合に，リクエストを切断，あるいは，拒否するようなアーキテクチャになっている．そのため，処理を継続しながらもリソースを制御することができない．また，たった一つのリクエスト処理がサーバリソースを占有する場合にも，強制的に処理を中断するような対応しかとれない．さらに，Webサーバ運用時は，事前に適切な閾値設定が困難であり，サーバ負荷が高まったことを監視で気付いた後に対応するといった，事後対応前提のアーキテクチャになっている．

Apacheのpreforkモデルの場合，一つのサーバプロセスが同時に複数のリクエストを処理しないことを利用し，一つのリクエストがコンピュータリソースを大きく占有しようとしても，その他のリクエスト処理が影響を受けず，さらに，制限されたリクエストも含めて，各リクエストが継続的に処理を行えるようにするためには，リクエストを処理中のプロセスを，一時的に限られたリソースの範囲内に分離してから，その制限されたリソース範囲内でリクエストを処理すれば良い．また，管理者がそのようなリソース制御ルールをプログラマブルに記述できれば，柔軟なリソース制御が可能になる．そこで，本章では，同一のサーバプロセス上で，リクエスト毎に任意のリソース分離が可能なリソース制御アーキテクチャを提案する．このアー

キテクチャによって，継続的にリクエストを処理しつつも，各リクエスト処理は一時的に仮想的なリソースを割り当てられ，そのリソースの範囲内で処理が行われる．そのため，各リクエストに対する処理が，特定のコンテンツに対する突発的なアクセス集中や，特定のリクエスト処理のリソース使用超過による影響を受けにくくなる．さらに，大量にリソースを消費するようなリクエストであっても，リクエストを拒否あるいは切断することなく，割り当てたリソースの範囲内で処理を継続させることが可能となる．また，リクエストを処理する時点で，そのリクエストがどのホストに対するリクエストなのかが識別できるため，ホスト単位でリソースを分離することも可能である．従来では高集積に収容すればするほど，互いにリソースの影響を受け合うためにサーバが不安定になりがちであり，それを解決するためにはリクエストを拒否，あるいは，切断という制限手法を取る必要があったが，本提案手法によって高集積であっても，高負荷時にクライアントにエラーを返すことなく任意のリソース制限下で安定的にサービスを提供できるようになる．

実装に関して，Linuxカーネルのバージョン3.10で動作するApacheに第5章で提案し実装したソフトウェアであるmod_mrubyを組み込み，mod_mruby上で動作するモジュールとして，Linuxの仮想化技術であるcgroupsを用いたリソース制御モジュールを実装した．制御ルールは，Rubyで記述できるようにした．

本章の構成は以下の通りである．6.2節では提案するリソース制御アーキテクチャについて説明する．6.3節で実装の詳細について述べ，6.4節でリソース制御アーキテクチャの精度評価を行い，6.5節でまとめとする．

6.2 提案するリソース制御アーキテクチャ

2.4.2節で述べた単一のサーバプロセスで複数のホストを高集積に管理するアーキテクチャにおいて，リソース占有の問題を解決するためには，リクエスト処理単位で，リソースを分離することにより，一つのリクエスト処理が，例えばCPUリソースを占有しようとしても，分離されたリソースの範囲内で動作するため，他のリクエスト処理に影響を与えない．また，管理者がリソース制御ルールをプログラマブルに記述できれば，柔軟なリソース制御が可能になる．それによって，サーバ管理者にとってこれまで難しかった多種多様なリソース制御に関する問題に対し，対応できる可能性が向上する．さらに，単一のサーバプロセスで高集積にホストを管理していることを考慮すると，可能な限りサーバプロセスを再起動することなく，リソース制御の変更をできるようにするべきである．

リクエスト処理時に，管理者が記述した内容に従って仮想的に分離されたリソース領域を作成し，サーバプロセスをそのリソース領域内で動作させることで，リク

エスト単位で任意のリソース制御が可能なWebサーバのリソース制御アーキテクチャを提案する．クライアントからサーバプロセスに対してリクエスト処理があると，そのリクエストが制御対象であった場合，サーバプロセス上で動作しているリソースコントローラが，リソース制御ルールからリソースに関する設定値を取得する．そして，そのリソース設定値を元に確保された仮想リソース領域がなければ，新規で領域を作成する．例えば，任意のリクエストに対し，CPU使用率は最大10%，ディスクへの書き込みは最大5MB/secに制限したいとする．その場合は，制御ルールを設定ファイルに記述する．記述後，新しいリクエストを受けた際に，リソースコントローラは制御ルールを解釈し，ルール通りにリソース領域を生成する．そして，サーバプロセスを，作成したリソース領域に割り当てた後，リクエストをそのリソース範囲内で処理する．処理後は，レスポンスをクライアントに返し，リソース領域への割り当てを解除してから，次のリクエスト処理に備える．また，リソースコントローラは，複数のWebサーバソフトウェア上でも同様に扱えるように，複数のWebサーバソフトウェアに汎用的な機能拡張インターフェイスを用意した上で，そのインターフェイス上に実装する．これによって，Webサーバソフトウェアの違いを気にすることなく，リソースコントローラ自体の拡張・修正が容易となる．このようなアーキテクチャを取ることにより，リクエストに含まれる情報，例えば，ホスト名やHTTPメソッド，ユーザ情報等を条件に，管理者が柔軟にリソース制御を行える．

6.3 アーキテクチャの実装

提案するアーキテクチャをLinux上で動作するApacheに実装した．仮想リソース領域の作成にはcgroupsを利用した．2.7節で言及した(2)優先順位の機能を使い，CPUやI/Oの最大使用率をリクエスト単位で分離する．汎用的なWebサーバの機能拡張I/Fには，5章で実装したmod_mrubyを利用した．提案するアーキテクチャでは，リクエスト毎に制御ルールを読み込むため，mod_mrubyによるリクエスト単位で高速に動作するWebサーバ機能拡張は，本アーキテクチャの実装に適している．図6.1にApacheとmod_mrubyとcgroupsの関係を示す．

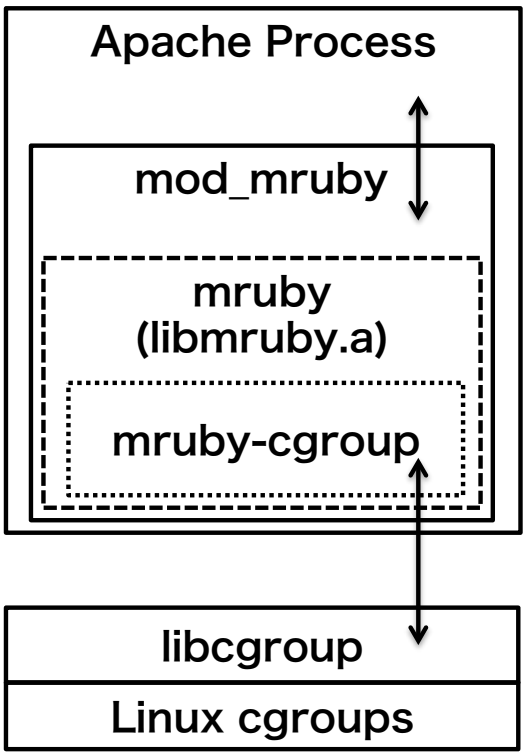

図 6.1　ApacheとMod_mrubyとcgroupの関係

RubyスクリプトからApacheの内部APIを操作できるように，ApacheにMod_mrubyを組み込む．同様に，Rubyスクリプトからcgroupを操作するために，cgroupsを操作するためのシステムライブラリであるlibcgroupをRubyで制御できるようにmruby-cgroupという拡張モジュールを実装した．これによって，RubyスクリプトからApache内部の情報を取得したり，その情報を元にlibcgroupを操作したりすることが可能になる．

図6.2に，提案するアーキテクチャをLinux上で動作するApacheに実装する場合のアーキテクチャの構成を示す．

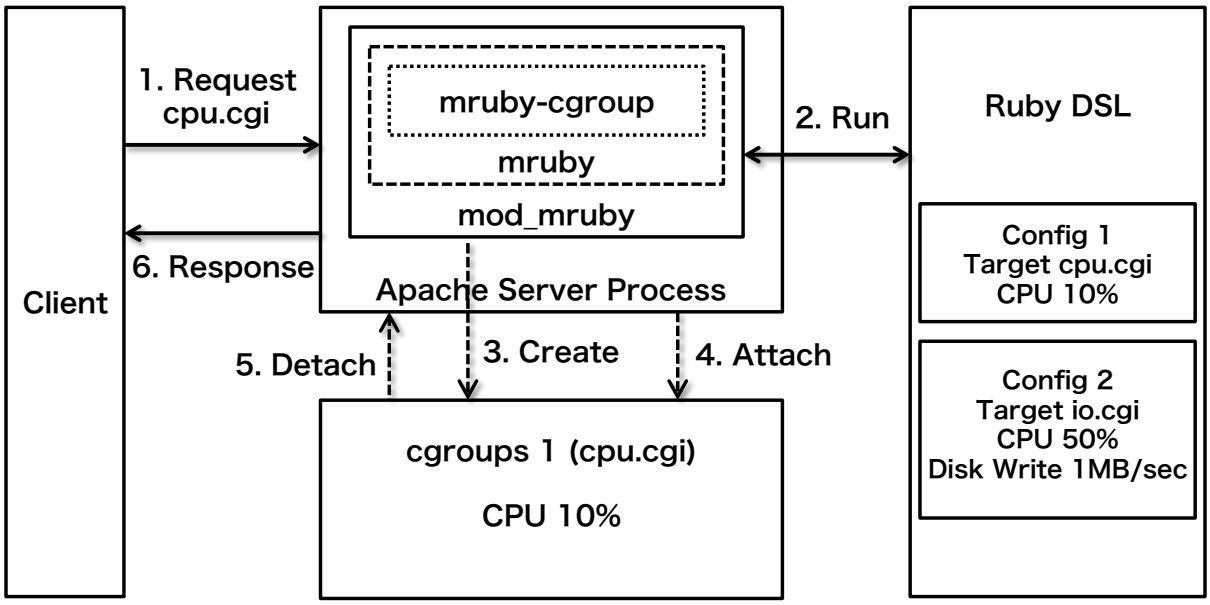

図 6.2　Linuxの実装例

リソースコントローラはmrubyとmruby-cgroup[15]が担い，その制御をmod_mrubyが担う．制御ルールはRubyの構文で記述し，ファイル，ディレクトリ，ホスト単位等，様々な条件から，CPUの使用率を制限可能である．また，サーバのCPUコア全てが100%近く使用されている状況において，複数のプロセスにCPU使用率を均等に分配して処理することもできる．cgroupsによるリソース領域は複数設定することが可能で，リクエスト時に既に存在する場合はその領域を利用し，存在しない場合は新規で領域を確保する．Rubyで記述された各種リソース制御ルールに従って，クライアントからのリクエスト処理のリソースを制御する．図6.3に，cpu.cgiにリクエストがあった場合に，そのリクエスト処理を，CPUを1000msのうち100msだけスケジューラが割り当てるように設定することで，CPU使用率を最大10%に制限する場合のリソース制御ルール例を示す．

[15] https://github.com/matsumotory/mruby-cgroup

```
r = Apache::Request.new

if r.filename = "/path/to/cpu.cgi"
  cpu = Cgroup::CPU.new "cpu_group"
  if ! cpu.exist?
    cpu.cfs_quota_us = 10000
  end

  cpu.create
  cpu.attach
end
```

図 6.3 リソース制御ルール記述例

図6.3に示したとおり，Rubyで記述可能な機能拡張インターフェイスによって，Ruby言語により，リソース制御ルールを柔軟に記述することが可能となる．また，制御ルールをフックしておくためのフェーズは，サーバのレスポンスを生成するまでのリクエスト解析，アクセスチェック，レスポンス生成直前，レスポンス生成後などである．

図6.4にリソース制御ルールの応用例を示す．図6.4では，予め高負荷ユーザが所属するホストを対象に，リクエストが対象のホストであれば高負荷ユーザと判定し，全体のCPUの25%，Disk I/Oの読み書きを最大30MBytes/sec使えるようにリソースを分離する．また，対象外のホストに対してはCPUを最大75%，Disk I/Oの読み書きを最大100MBytes/sec使用できるように設定した例である．対象のホストのリストについては，ホスト数が大きくなってき場合は，ハッシュやデータベースなどに保存することにより，スケーラビリティを維持できる．Webサーバソフトウェア内部の情報やOSの負荷状況を考慮した記述，仮想ホスト，ファイル，ディレクトリ単位の制御もRubyの制御構造によって柔軟に記述でき，汎用性が高い．

```
targets = %w(

target1.example.com
target2.example.com
target3.example.com
target4.example.com

)

r = Apache::Request.new

if targets.include? r.hostname
  c = Cgroup::CPU.new "httpd-limiterd"
  c.shares = 25
  if c.exist?
    c.modify
  else
    c.create
  end
  c.attach

  b = Cgroup::BLKIO.new "httpd-limited"
  b.throttle_write_bps_device = "8:0 30000000"
  b.throttle_read_bps_device = "8:0 30000000"
  if b.exist?
    b.modify
  else
    b.create
  end
  b.attach
else
  c = Cgroup::CPU.new "httpd"
  c.shares = 75

  if c.exist?
    c.modify
  else
    c.create
  end
  c.attach

  b = Cgroup::BLKIO.new "httpd"
  b.throttle_write_bps_device = "8:0 100000000"
  b.throttle_read_bps_device = "8:0 100000000"
  if b.exist?
    b.modify
  else
    b.create
  end
  b.attach
end
```

図 6.4　ホスト単位でのリソースの分離

Rubyスクリプトを変更することで，サーバプロセスを再読み込みすることなく，変更後の次のリクエストから条件を変更できる仕様であるため，運用性も高い．一方，スクリプトの変更を必要としない場合は，事前に中間コードにコンパイルしておいて，リクエスト時に中間コードを実行することでより高速に動作させることも可能である．なお，mod_mrubyは高速かつ省メモリで動作することも本機構に適合しており，5.3節の性能評価からも，バイトコードキャッシュ時のスクリプト実行のオーバーヘッドは非常に少ないことが分かっている．

6.4 リソース制御アーキテクチャの精度評価

提案するリソース制御アーキテクチャのCPU制御機能を組み込むことによるオーバーヘッドやリソース制御の精度評価を行った．性能評価として，本アーキテクチャの導入前後でリソース制御対象でないリクエストの性能にどの程度違いがあるか，リクエスト処理時間の違いによってリソース制御の精度に差がどれ程生じるのかを評価した．表6.1にテスト環境のマシンスペックを示す．

表 6.1　リソース制御精度を測るためのテスト環境

	Client Machine	Server Machine
CPU	Intel Core2Duo E8400 3.00GHz	Intel Xeon X5355 2.66GHz
Memory	4GB	8GB
NIC	Realtek RTL8111/8168B 1Gbps	Broadcom BCM5708 1Gbps
OS	CentOS 5.6	CentOS 5.6
Middleware	-	Apache/2.2.3

まず，本アーキテクチャを導入することで，リソース制御対象でないリクエストの性能にどの程度差異があるかを評価した．評価方法として，リソース制御アーキテクチャの処理の影響を最大にするため，リクエスト対象のファイルはhello worldを出力するだけの静的なHTMLファイルとした．そのファイルに対し，表6.1のテスト環境のリソースを十分に使用できるように，予備実験から同時接続数100，総接続数10万のリクエストパターンを決定し，評価を行った．ベンチマークソフトウェアにはabコマンド[74]を利用した．その結果，リソース制御アーキテクチャを導入して

いない場合は，1秒間に32915.46リクエスト処理できていたのに対し，リソース制御アーキテクチャを導入している場合は，32322.07リクエスト処理できていた．この結果から，リソース制御アーキテクチャを導入することによるボトルネックはほとんどないと考えられる．

次に，リクエストの処理時間の違いによって，リソース制御の精度にどれほど差異が生じるのかを評価した．評価方法として，単純なループを行うCGIプログラムを作成し，そのループ回数を変化させることで，リクエスト処理時間を変化させた．そのCGIプログラムに対して，リソース制御アーキテクチャによりCPU使用率を最大50%に制限し，表6.1のテスト環境のリソースを十分に使用できるように予備実験から決定した同時接続数10，総接続数1000の条件で，CGIプログラムにリクエストを送信した．CGIプログラムのリクエスト処理時間を変化させながら，1リクエストの処理にかかった時間が，リソース制御をしていない場合にかかった処理時間と比較して，性能がどれだけ低下しているかを測定した．その値を性能制御率と呼ぶことにする．50%にリソース制御している場合，性能制御率が50%に近ければ近いほど，正確にリソース制御できていることになる．図6.5に1リクエストの処理時間を変化させた場合の，リソース制御アーキテクチャの性能制御率を示す．図6.5のように，1リクエストの処理時間が0msecから6msec程度の場合は，表6.1のテスト環境において，本来制御したい50%よりも，より低く性能が制御されていることがわかる．これはリソース制御の際に，リクエストがリソース制御対象であった場合に，仮想的に作成したリソース分離領域にプロセスを参加させる処理が，リクエスト処理時間とくらべて無視できない程度の処理時間となり，オーバーヘッドになっているためだと考えられる．一方で，6msec以上リクエスト処理に時間がかかるような処理は，性能を50%前後に制御できていることがわかる．

以上より，CPU使用率をリソース制御する場合において，リクエスト処理時間が，リソース制御アーキテクチャによってリソースを分離するための処理時間よりも十分大きい場合において，本手法が有効であることが分かった．また，単一のリクエストが大きくCPUリソースを占有するような場合においては特に，本手法が有効に機能すると考えられる．

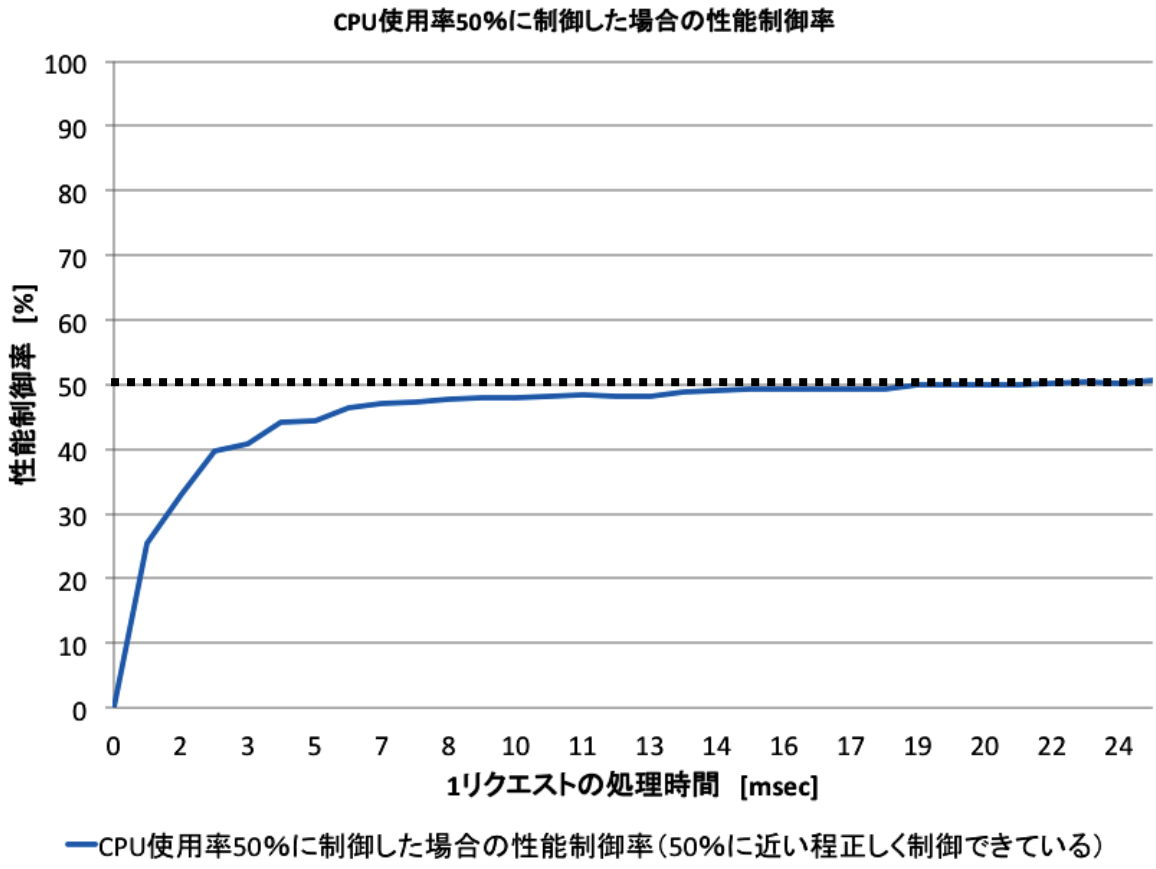

図 6.5　リクエスト処理時間の違いによるリソース制御精度

制度評価により，高集積マルチテナント環境において，リソースを占有するようなプログラムに対してリクエストがあったとしても，管理者が設定したCPU使用率を超えることはない．すなわち，高負荷時でも他のリクエストへ与える影響は，リクエスト処理時間の大小にかかわらず限定できる．また，1.2節で述べたように，従来の制限手法は処理を継続しながらリクエスト単位でリソースを制御することが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課題があったが，リクエストを拒否，あるいは，切断することなく継続的な処理とリソースの制限が実現できるため，本研究の目的を達成できる．

6.5 結語

本章では，限られたリソースで高集積にホストを収容する際に，単一のサーバプロセスで複数のホストを管理する場合に生じるリソース制御の問題を解決するために，サーバプロセス単位で仮想的にリソースを分離するWebサーバのリソース制御機

構を提案した．本手法により，一つのリクエストがコンピュータリソースを大きく占有しようとしても，他のリクエストが処理できなくなるようなことが生じない．また，リクエスト処理を切断・拒否することなく継続的にリソースを制御できる．さらに，サーバ管理者がRubyでサ柔軟にリソース制御ルールを記述可能であるため，多種多様なリソース制御に関する問題に対し，対応できると考えている．

本章のリソース制御手法と3章のCGI実行方式のためのアクセス制御手法や4章のDSO実行方式のためのアクセス制御手法は併用可能である．

7 結論

本論文では，Webサーバの高集積マルチテナントアーキテクチャにおいて，Webホスティングサービスのように，各ホストのWebコンテンツをサービス事業者が管理できない状況における，高品質でありながらハードウェアや運用管理コストを低減させるための最適なアーキテクチャについて述べた．

Webコンテンツをサービス事業者が管理できないような高集積マルチテナントアーキテクチャのWebサービスでは，ホスト単位でサービス利用者を収容するため，ホスト間で適切に権限分離を行い，セキュリティを担保する必要がある．その上で，高集積のために，ハードウェアリソースを適切に共有し，最大限の性能を提供しつつも，各ホスト間でのリソース競合を低減しなければならない．また，そのようなアーキテクチャ実装を容易にするための機能拡張支援機構が必要である．

1章と2章では本研究の背景と目的，立ち位置を示し，高集積マルチテナントアーキテクチャの研究の重要性をまとめた．本研究について述べるための前提として，LinuxやApacheに関する基礎概念や用語を整理した上で，高集積マルチテナントアーキテクチャにおける，運用技術，セキュリティ，性能，機能拡張，リソース制御に関する関連研究と課題を体系的にまとめた．

3章では高集積マルチテナントアーキテクチャにおけるセキュリティと性能，および，運用技術の改善手法について述べた．高集積マルチテナントアーキテクチャを実現するために，WebサーバソフトウェアとしてApacheが広く使われている．Apacheでは，高集積マルチテナントアーキテクチャを提供するためにVirtualHostという機能を有しており，VirtualHostにおいては，CGIプログラムをホスト単位で権限分離するために，通常，suEXECと呼ばれるアクセス制御機能が使われる．suEXECは，ホスト間の権限分離は可能であるが，システム領域へのアクセスを考慮できていない．また，ホスト単位の権限設定を静的に設定ファイルに記述する必要がある．そのため，高集積にホストを収容した状況においては，ホスト数に依存して設定数が増加し，Apache起動時にはメモリを多く消費した状態で起動する．その結果，CGIプログラムの性能が低下したり，メモリを多く消費するという課題があった．そこで，リクエストのあったファイルから動的にオーナを取得し，そのオーナに基いて，該当するホストのドキュメントルートでchroot()システムコールを実行してから，chroot環境配下で権限分離を行うアクセス制御アーキテクチャを提案した．このアーキテクチャでは，リクエストから動的にオーナを取得するため，設定ファイルに静的な設定の記述を要求しない．そのため，通常のsuEXECよりもサーバプロセスのメモリ使用量が低減できる．

4章では，動的コンテンツの実行速度を維持しながら，セキュリティを担保するための性能劣化の少ないアクセス制御アーキテクチャについて述べた．3章では，CGI実行方式のメリットや運用技術を活かしたアクセス制御アーキテクチャを提案した．CGI実行方式のメリットとして，複数のインタプリタや同一のインタプリタであっても複数のバージョンを切り替えて使えることが挙げられる．しかし，リクエスト毎にプロセスの生成・破棄やCGI用の比較的大きなバイナリファイルやsuEXECのバイナリファイルをexecve()システムコールによって実行す必要があり，権限分離のためのコストが高く性能が低かった．fork()やexecve()システムコールを必要とせず，インタプリタを直接Webサーバプロセスに組み込んでプログラムを実行できるDSO実行方式は，その性能をいかしながらも適切に権限分離するアーキテクチャがなかった．そこで，DSO実行方式の性能をいかしながらも，スレッドの生成・破棄程度のコストのみで権限分離できるアクセス制御アーキテクチャを提案した．このアーキテクチャによって，CGI実行方式で権限分離を行う場合と比べて，大幅に高速に権限分離しつつ，プログラムを実行することが可能になった．また，提案したアーキテクチャによって，CGI実行方式とDSO実行方式を区別せずに，統一的に権限分離することも可能であり，サーバ管理者に非常に扱いやすい仕様になっている．

5章では，高集積マルチテナント環境における運用技術やWebサーバの振る舞いを容易に改善するために，スクリプト言語によって平易に実装可能でありながら，従来の手法よりも性能劣化の少ないWebサーバの機能拡張支援機構アーキテクチャについて言及した．これまで，Webサーバの機能拡張は，一般的に性能を重視してC言語で実装する場合がほとんどであった．一方で，生産性や保守性を考慮して，スクリプト言語で機能拡張する手法も提案されてはいるが，性能面や安全面で課題があった．そこで，スクリプト言語で機能拡張が可能ながら，高速かつ省メモリで動作し，グローバルの状態を共有しない安全な機能拡張支援アーキテクチャを提案した．このアーキテクチャによって，リクエスト毎にWebサーバにフックされたスクリプト言語をコンパイルする場合でも，従来の手法よりも高速に動作する．さらに，Webサーバ起動時にスクリプトをバイトコードまでコンパイルしておくバイトコードキャッシュ方式では，リクエスト毎にコンパイルを必要としないため，C言語による機能拡張に遜色ない性能が得られた．

6章では，高集積マルチテナント環境におけるリクエスト単位でのリソース競合を解決するために，従来のプロセス単位のリソース制御を，Webサーバへのリクエスト単位で実現できるアーキテクチャを提案した．Webサーバの従来のリソース制御は，リクエストを受け付けるか拒否するかの単純な手法であり，一つのリクエストがサーバリソースを占有するような状況を考慮できていない．高集積マルチテナントアー

キテクチャのWebサーバでは，収容ホストに対するリクエストを，収容ホストよりも少ない数のプロセスで同時に処理するため，リクエスト間でリソース競合が起き，ホスト単位でリソースを均一に分割して使うことが難しい．提案するアーキテクチャは，多数のリクエストやリソースを占有するリクエストを同時並行で処理する状況で，各リクエストを制限された仮想リソース領域に隔離することにより，リクエスト間で干渉度の低いリソース競合を実現できる．アーキテクチャの実装では，5章の機能拡張支援機構を応用することによって，リソース制御そのものが性能劣化にならず，スクリプト言語で容易に開発・保守可能なWebサーバのリソース制御アーキテクチャを実現した．

今後の課題や展望として，HTTP/2プロトコルへの対応，mrubyによるHTTP/2サーバやSSLハンドシェイクの制御，Webサーバのリソースの自律制御があげられる．

本研究の実装は一つのリクエストに対して一つのサーバプロセスが処理を専有するような一般的なHTTP/1プロトコルのWebサーバのアーキテクチャに基づいている．HTTP/2の普及や，一つのサーバプロセスで複数のリクエストを非同期で処理するWebサーバのアーキテクチャの普及にともなって，新しいWebサーバのアーキテクチャに対応した実装を今後検討する必要がある．例えばmod_process_securityは，一つのプロセスが並行でリクエストを処理しないことを前提に，スレッド単位で権限を分離しているが，プロセスが並行にリクエストを処理するアーキテクチャでは，スレッド間でメモリ領域の共有が可能になり適切に権限分離できない．スレッド単位ではなく，関数の実行単位での権限分離が必要となる．

Webサーバの機能拡張支援アーキテクチャmod_mrubyやngx_mrubyは，世界中のWebフレームワークの性能を競うTechEmpower Web Framework Benchmarksにおいて動的にPlaintextを生成する分野で137個のWebサーバの中で12位[16]，Ruby部門では世界1位[17]となるなど，OSS公開後から注目されるようになってきた．2014年には，HTTP/2プロコトルの実装であるnghttp2[18]や，HTTP/2サーバの実装であるH2O[19]などがOSSとして公開され，HTTP/2の実用的な実装が普及し始めている．筆者は，HTTP/2プロトコルのサーバプッシュや優先度制御といったHTTP/2の新しい機能の複雑さに着目し，mrubyによってHTTP/2サーバの拡張を行えるように，H2Oにmrubyによる機能拡張機能

16 https://www.techempower.com/benchmarks/#section=data-r12&hw=ph&test=plaintext

17 https://www.techempower.com/benchmarks/#section=data-r12&hw=ph&test=plaintext&l=4fteyn

18 https://nghttp2.org/

19 https://h2o.example.net/

を提案し実装が取り込まれた[20]．それによって，nghttp2もmrubyによる機能拡張が実装されはじめ，現在のHTTP/2プロコトルで通信できる主要なWebサーバソフトウェアのApache，nginx，nghttp2，H2Oは全てmrubyで機能を拡張できるようになった．今後は，企業でも徐々にHTTP/2サーバの導入が進んでいくことが予想されるため，mrubyによってHTTP/2プロトコルを最大限活かす実装を企業に向けてフィードバックしていく必要がある．また，mod_mrubyやngx_mrubyはApacheやnginx自体の仕様に大きく影響を受けるため，機能面が限定されるという課題があったが，筆者自身でTrusterd[21]という名前のHTTP/2サーバを実装しOSSとして公開している．Trusterdは，全てのサーバ機能をmrubyで設定できる仕様にしており，mod_mrubyやngx_mrubyのように各フェーズのみでRubyスクリプトを実行するよりもプログラマブルに設定を記述できるソフトウェアになっている．Webサービスの高度化と基盤技術の複雑化にともない，Webサーバソフトウェアの設定を各ソフトウェア実装者による独自設定で記述するよりも，mrubyやLuaといった組み込みスクリプト言語で汎用的に記述できる方が生産性が高く，性能もそれほど問題にならないことを伝えるために，mod_mrubyやngx_mruby，Trusterd，H2O，nghttp2のmruby拡張機能などの普及活動を継続的に行っていく予定である．

HTTP/2はTLSによる通信が前提となっており，高集積マルチテナントアーキテクチャにおいては，ホストの収容数に伴って管理する証明書の数が増えるため，単一のサーバプロセスでいかに効率よく証明書を扱うかが課題である．これまでのWebサーバの実装では，設定に各ホストに該当する証明書のパスを記述しておき，Webサーバ起動時に読み込むようになっている．しかし，高集積マルチテナントアーキテクチャでHTTP/2かつTLSが前提の場合，事前に読み込んでおく証明書の数が膨大となってサーバプロセスのメモリ使用量が大きくなり，2.6.2.1節で述べたように性能が低下する．そこで，ngx_mrubyにTLSの通信における証明書の扱いをmrubyで制御できるように実装し，Server Name Indication[22]と組み合わせることによって，証明書を事前に読み込むことなくリクエストのあったホスト名から該当の証明書をデータベースから動的に選択し，SSLハンドシェイクを行えるようにした．今後は，ngx_mrubyによる大量証明書の動的読み込みの実装の評価をもとにその有用性を企業にフィードバックしていくことによって，HTTP/2やTLSによる通信を普及させていく予定である．

---

[20] https://github.com/h2o/h2o/pull/378

[21] https://github.com/matsumotory/trusterd

[22] SSL/TLS拡張の一つで，SSLハンドシェイク時にクライアントがアクセスしたいホスト名を通知し，サーバがホスト名によって証明書を選択できるようにする機能．SNIと省略されることが多い．

最後に，Webサーバのリソースの自律制御に対する展望を述べる．Webサーバの高集積マルチテナントアーキテクチャによって構築された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，複数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していくため，適切な調査と対策に要するコストが非常に高くなる．6章では，リクエスト単位で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．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について，刻々と変化する状況下で人力による調査に頼って判断することは高コストである．適切な閾値を，いかにシステム管理者の運用コストをできるだけかけずに把握し，適用するかという，ホスト単位で精細なリソース制御を行う際の課題が残されている．

そこで，Webサーバのリソース特徴量を時系列データとして抽出した上で変換点検出を行い，原因となるホストと変化らしさの重み付けを都度解析した上で，サーバ全体のリソース逼迫時には，解析結果に基いて自律的に原因となるリクエストをリソース分離するアーキテクチャを検討したい．時系列データには，レスポンス生成に使用したCPU使用時間をホスト毎に一定期間保存しておき，保存した時系列データを変化点検出アルゴリズムにより変化点スコアを計算する．リクエスト時のホスト名に基いて，ホスト単位で計測したスコアをスコアリストに登録しておき，同一ホストのスコアが既に存在する場合は，スコアを加算していく．これをリクエスト単位で繰り返し登録していくことにより，リソースの傾向変化に寄与したホストのランキングリストが抽出できる．そして，サーバ全体のリソースが逼迫した場合に，リストのランキングに基づいて，ランキング上位のホストへのリクエストのみを，リソース使用量が限定された隔離環境内でレスポンスを処理するようにする．閾値を下回った場合は，その後一定の時間を設けた後に隔離環境下で処理していたホストの制限を解除する．これらを，Webサーバのレスポンス生成処理に組み込むことができれば，Webサーバは自律的にリソースの高負荷状態の原因を解析し，必要な時に原因に対して対処できるのではないかと考えている．

謝辞

本論文は，2008年にファーストサーバ株式会社に入社後に，ホスティングサービスをはじめ，沢山のWebサービスを運用・保守する中で生じた問題意識を解決するために，取り組んできた研究開発をまとめたものです．2012年に，京都大学大学院情報学研究科の岡部研究室に配属後，研究室の皆様をはじめ，多くの方々に多大なるご支援とご指導を頂くことで博士論文としてまとめることができました．ここに改めて，博士論文執筆および学位取得に向けてお世話になった皆様に，誠意を持って感謝の意を表します．

博士論文を執筆にあたり，岡部寿男教授には，多大なるご指導や数多くの貴重なご助言を頂きました．また，それだけではなく，これからのインターネット技術に対してどう取り組むべきであるかといった，現在の著者の研究開発に対する核となく考え方や行動指針を持つに至るまでに，沢山の貴重なご助言を賜りました．心から感謝を申し上げます．本審査に至るまで，中島浩教授，五十嵐淳教授には，沢山の貴重なご助言を頂きました．心から御礼申し上げます．また，私が2015年入社後，現在も所属しているGMOペパボ株式会社の諸氏，特に，本研究の執筆にあたり，沢山のご支援をはじめ，業務時間内に研究開発することに対して全面的に協力頂いた，佐藤健太郎社長，栗林健太郎取締役CTOをはじめ，博士論文執筆中にもチームとして支えてくださった技術部技術基盤チーム，ペパボ研究所には，感謝の意を表します．特に，博士論文執筆の際には，力武健次さん，廣川優さん，奥村晃弘さん，田平康朗さん，久米拓馬さんには沢山のご支援を頂きました．ここに御礼申し上げます．また，本研究の問題意識としてまとめた内容は，2008年から2012年まで所属していたファーストサーバ株式会社の諸氏と共に，多くの課題や問題に対して，妥協なく協力しながら取り組めたために整理できました．心から御礼申し上げます．

最後に，大学卒業後に一度会社に入社しながらも，博士課程で研究を行うために会社をやめるという選択に対して常に肯定的に支持してくれて，その後のあらゆる私的な面で多大なる支えとなってくれた妻と子供，そして，博士論文執筆に至るまでに，幼少期から今に至るまでに，博士論文執筆という選択肢を現実的なものにまで考えられるように教育し，あらゆる面で影響を与え，強い支えとなった両親と兄弟に心から感謝の意を表します．

参考文献

[1] Amazon EC2, Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2/.

[2] Bacarella M, Taking Advantage of Linux capabilities. Linux Journal, 2002.

[3] Banga G, Druschel P, Mogul J C, Resource Containers: A New Facility for Resource Management in Server Systems, OSDI, Vol. 99, pp. 45-58, February 1999.

[4] Bekman S, Cholet E, Practical mod_perl, O'Reilly Media, Inc., 2003.

[5] Ben-Yehuda M, Mason J, Xenidis J, Krieger O, Van Doorn L, Nakajima J, Wahlig E, Utilizing IOMMUs for Virtualization in Linux and Xen, The 2006 Ottawa Linux Symposium (OLS'06), pp. 71-86, July 2006.

[6] Bhardwaj S, Jain L, Jain S, Cloud Computing: A Study of Infrastructure as a Service (IAAS), International Journal of engineering and information Technology, Vol. 2, No. 1, pp. 60-63, 2010.

[7] Bhattiprolu S, Biederman E W, Hallyn S, Lezcano D, Virtual Servers and Checkpoint/Restart in Mainstream Linux. ACM SIGOPS Operating Systems Review, Vol. 42, No. 5, pp. 104-113, 2008.

[8] Cecchet E, Chanda A, Elnikety S, Marguerite J, Zwaenepoel W, Performance Comparison of Middleware Architectures for Generating Dynamic Web Content, The ACM/IFIP/USENIX 2003 International Conference on Middleware, pp. 242-261, June 2003.

[9] 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.

[10] Checconi F, Cucinotta T, Faggioli D, Lipari G, Hierarchical Multiprocessor CPU Reservations for The Linux Kernel, The 5th International Workshop on Operating Systems Platforms for Embedded Real-time Applications (OSPERT 2009), pp. 15-22, June 2009.

[11] Chen H, Wagner D, MOPS: An Infrastructure for Examining Security Properties of Software, The 9th ACM Conference on Computer and Communications Security, pp. 235-244, November 2002.

[12] David J, mod_limitipconn, http://dominia.org/djao/limitipconn.html.

[13] Felter W, Ferreira A, Rajamony R, Rubio J, An Updated Performance Comparison of Virtual Machines and Linux Containers, IEEE International Symposium Performance Analysis of Systems and Software (ISPASS), pp. 171-172, March 2015.

[14] Ferdman M, Adileh A, Kocberber O, Volos S, Alisafaee M, Jevdjic D, Falsafi B, Clearing the clouds: a study of emerging scale-out workloads on modern hardware, ACM SIGPLAN Notices, Vol. 47, No. 4, pp. 37-48, March 2012.

[15] Fielding R T, Architectural Styles and The Design of Network-based Software Architectures, Doctoral Dissertation, University of California, Irvine, 2000.

[16] Fielding R T, Kaiser G, The Apache HTTP Server Project, IEEE Internet Computing, Vol. 1, No. 4, pp. 88-90, 1997.

[17] Fitzpatrick B, Distributed Caching with Memcached. Linux journal, No. 124, pp. 72-76, 2004.

[18] Garfinkel T, Pfaff B, Chow J, Rosenblum M, Boneh D, Terra: A Virtual Machine-based Platform for Trusted Computing, ACM SIGOPS Operating Systems Review, Vol. 37, No. 5, pp. 193-206, October 2003.

[19] Garfinkel T, Rosenblum M, A Virtual Machine Introspection Based Architecture for Intrusion Detection, NDSS, Vol. 3, pp. 191-206, February 2003.

[20] Goerzen J, mod_python, Foundations of Python Network Programming, pp. 393-416, 2004.

[21] Goldberg R P, Survey of Virtual Machine Research, Computer, Vol. 7, No. 6, pp. 34-45, 1974.

[22] Gu Y, Lee B S, Cai W, Evaluation of Java Thread Performance on Two Different Multi threaded Kernels, Operating Systems Review, Vol. 33, No. 1, pp. 34-46, 1999.

[23] Habib I, Virtualization with KVM, Linux Journal, No. 166, pp. 8, 2008.

[24] Hara D, Fukuda R, Hyoudou K, Ozaki R, Nakayama Y, Hi-sap: Secure and Scalable Web Server System for Shared Hosting Services, International Conference on Broadband Communications, Networks and Systems, pp. 119-137, Oct 2010.

[25] Hara D, Nakayama Y, Secure and High-performance Web Server System for Shared Hosting Service, IEEE 12th International Conference on Parallel and Distributed Systems (ICPADS' 06), Vol. 1, pp. 8-15. July 2006.

[26] Hellerstein J L, Zhang F, Shahabuddin P, Characterizing Normal Operation of A Web Server: Application to Workload Forecasting and Problem Detection, IMG Conference, Vol. 1, pp. 150-160, December 1998.

[27] Hideo N, Pavel S, mod_ruid, http://websupport.sk/~stanojr/projects/mod_ruid/.

[28] Hideo N, mod-suid2, http://code.google.com/p/mod-suid2/.

[29] ISO/IEC 30170:2012 Information technology -- Programming languages -- Ruby, http://www.iso.org/iso/iso_catalogue/catalogue_icscatalogue_detail_ics.htm?ics1=35&ics2=060&ics3=&csnumber=59579.

[30] Ierusalimschy R, De Figueiredo L H, Celes Filho W, Lua: An Extensible Extension Language, Softw Pract Exper, Vol. 26, No. 6, pp. 635-652, 1996.

[31] Java Servlet 3.0 Specification, http://jcp.org/en/jsr/detail?id=315.

[32] Josey A, The Single UNIX Specification Version 3, http://www.unix.org/version3/.

[33] Kamp P H, Watson R N, Jails: Confining The Omnipotent root, The 2nd International SANE Conference, Vol. 43, pp. 116, May 2000.

[34] Kivity A, Kamay Y, Laor D, Lublin U, Liguori A, KVM: The Linux Virtual Machine Monitor, The Linux Symposium, Vol. 1, pp. 225-230, July 2007.

[35] Kovári 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.

[36] Labrinidis A, Roussopoulos N, Generating Dynamic Content at Database-backed Web Servers: cgi-bin vs. mod_perl, ACM SIGMOD Record 29, pp. 26-31, 2000.

[37] Labrinidis A, Roussopoulos N, WebView Materialization, ACM SIGMOD Record, Vol. 29, No. 2, pp. 367-378, May 2000.

[38] Lerner R M, At the Forge: Writing Modules for mod_perl, Linux Journal, 60es, pp. 17, 1999.

[39] M Belshe, R Peon, and M Thomson, Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, 2015.

[40] Matthews J N, Hu W, Hapuarachchi M, Deshane T, Dimatos D, Hamilton G, Owens J, Quantifying The Performance Isolation Properties of Virtualization Systems, ACM The 2007 Workshop on Experimental Computer Science, pp. 6, June 2007.

[41] Maunder A, Van Rooyen R, Suleman H, Designing A Universal Web Application Server, The 2005 Annual Research Conference of The South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries, pp. 86-94, July 2005.

[42] 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.

[43] Mirheidari S A, Arshad S, Khoshkdahan S, Jalili R, Two Novel Server-side Attacks against Log File in Shared Web Hosting Servers, Internet Technology And Secured Transactions, pp. 318-323, December 2012.

[44] Mirheidari S A, Arshad S, Khoshkdahan S, Performance Evaluation of Shared Hosting Security Methods, IEEE 11th International Conference on Trust Security and Privacy in Computing and Communications, pp. 1310-1315, June 2012.

[45] Mockus A, Fielding R T, Herbsleb J, A Case Study of Open Source Software Development: The Apache Server, ACM The 22nd International Conference on Software engineering, pp. 263-272, June 2000.

[46] Mosberger D, Jin T, httperf: A Tool for Measuring Web Server Performance. ACM SIGMETRICS Performance Evaluation Review, Vol. 26, No. 3, pp. 31-37, 1998.

[47] NPO法人軽量Rubyフォーラム, http://forum.mruby.org/.

[48] National Security Agency, Security-Enhanced Linux, http://www.nsa.gov/research/selinux/.

[49] Netcraft, July 2013 Web Server Survey, http://news.netcraft.com/archives/2013/07/02/july-2013-web-server-survey.html.

[50] Nginx, http://nginx.org/ja/.

[51] PHP, http://php.net/.

[52] Prandini M, Faldella E, Laschi R, Mandatory Access Control Applications to Web Hosting, EC2ND 2006, pp. 13-22, 2007.

[53] 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.

[54] Rabinovich M, Aggarwal A, Radar: A Scalable Architecture for A Global Web Hosting Service, Computer Networks, Vol. 31, No. 11, pp. 1545-1561, 1999.

[55] Roberto I, Waldemar C, Luiz H F , The Programming Language Lua, http://www.lua.org.

[56] Rosen R, Resource Management: Linux Kernel Namespaces and cgroups, Haifux, May 2013.

[57] Ruby on Rails, http://rubyonrails.org/.

[58] Schroeder T, Goddard S, Ramamurthy B, Scalable Web Server Clustering Technologies. IEEE network, Vol. 14, No .3, pp. 38-45, 2000.

[59] Sebastian M, suPHP Homepage, http://www.suphp.org/Home.html.

[60] Shugo M, mod_ruby, https://github.com/shugo/mod_ruby.

[61] Simonson J, Berleant D, Zhang X, Xie M, Vo H, Version Augmented URIs for Reference Permanencevia An Apache Module Design, Computer Networks and ISDN Systems, Vol. 30, No. 1, pp. 337-345, 1998.

[62] Soltesz S, Pötzl H, Fiuczynski M E, Bavier A, Peterson L, Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors, ACM SIGOPS Operating Systems Review, Vol. 41, No. 3, pp. 275-287, March 2007.

[63] Stein L, MacEachern D, Writing Apache Modules with Perl and C: The Apache API and mod_perl, O'Reilly Media, Inc., 1999.

[64] Stein L, SBOX, Put CGI Scripts in a Box, USENIX Annual Technical Conference, General Track, pp. 145-155, June 1999.

[65] Sullivan D G, Seltzer M I, Isolation with Flexibility: A Resource Management Framework for Central Servers, USENIX Annual Technical Conference, pp. 27-27, June 2000.

[66] The Apache Software Foundation, Apache Module mod_actions, http://httpd.apache.org/docs/2.0/en/mod/mod_actions.html.

[67] The Apache Software Foundation, Apache Module mod_lua, http://httpd.apache.org/docs/trunk/mod/mod_lua.html.

[68] The Apache Software Foundation, Apache Module mod_vhost_alias, http://httpd.apache.org/docs/2.0/mod/mod_vhost_alias.html.

[69] The Apache Software Foundation, Apache Tutorial: Dynamic Content with CGI, http://httpd.apache.org/docs/2.2/en/howto/cgi.html.

[70] The Apache Software Foundation, Apache Virtual Host documentation, http://httpd.apache.org/docs/2.2/en/vhosts/.

[71] The Apache Software Foundation, Apacheモジュール一覧, https://httpd.apache.org/docs/2.2/ja/mod/.

[72] The Apache Software Foundation, Dynamic Shared Object (DSO) Support, http://httpd.apache.org/docs/2.2/en/dso.html.

[73] The Apache Software Foundation, Server - Wide Configuration Limiting Resource Usage, http://httpd.apache.org/docs/2.2/en/server-wide.html#resource.

[74] The Apache Software Foundation, ab: Apache HTTP Server Benchmarking Tool, http://httpd.apache.org/docs/2.2/programs/ab.html.

[75] The Apache Software Foundation, suEXEC Support, http://httpd.apache.org/docs/2.2/en/suexec.html.

[76] The PHP Group, PHP Manual Safe Mode, http://php.net/manual/ja/features.safe-mode.php.

[77] Titchkosky L, Arlitt M, Williamson C, A Performance Comparison of Dynamic Web Technologies, ACM SIGMETRICS Performance Evaluation Review, Vol. 31, No. 3, pp. 2-11, 2003.

[78] Trent S, Tatsubori M, Suzumura T, Tozawa A, Onodera T, Performance Comparison of PHP and JSP as Server-side Scripting Languages, ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing, pp. 164-182, December 2008.

[79] Van Doorn L, Hardware Virtualization Trends, ACM/Usenix International Conference on Virtual Execution Environments, Vol. 14, No. 16, pp. 45-45, June 2006

[80] Wang G, Ng T E, The Impact of Virtualization on Network Performance of Amazon EC2 Data Center, INFOCOM 2010, pp. 1-9, March 2010.

[81] Wikipedia, Load (computing), http://en.wikipedia.org/wiki/Load_average.

[82] Wu A, Wang H, Wilkins D, Performance Comparison of Alternative Solutions for Web-To-Database Applications, The Southern Conference on Computing, pp. 26-28, October 2000.

[83] Xavier M G, Neves M V, Rossi F D, Ferreto T C, Lange T, De Rose C A, Performance Evaluation of Container-based Virtualization for High Performance Computing Environments, 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 233-240, February 2013.

[84] Yi S, Kondo D, Andrzejak A, Reducing Costs of Spot Instances via Checkpointing in The Amazon Elastic Compute Cloud, IEEE 3rd International Conference on Cloud Computing, pp. 236-243, July 2010.

[85] 鈴木真一, 新城靖, 光来健一, 板野肯三, 千葉滋, ユーザ権限変更機構を利用した安全なイントラネットサーバの実現, 情報処理学会論文誌コンピューティングシステム(ACS), Vol. 44, No. 10, pp. 86-96, 2003.

[86] 富樫荘太, 大月勇人, 瀧本栄二, 毛利公一, Linux の Cgroups における CPU throttling の精度改善手法, 電子情報通信学会総合大会講演論文集 D-6. コンピュータシステム C (ソフトウェアと性能評価), pp. 62, 2014.

[87] 日本Linux協会, JM Project CAPABILITIES, http://archive.linux.or.jp/JM/html/LDP_man-pages/man7/capabilities.7.html.

[88] 原大輔, 中山泰一, Hussa: スケーラブルかつセキュアなサーバアーキテクチャ ~ 低コストなサーバプロセス実行権限変更機構, 第8回情報科学技術フォーラム (FIT 2009) 講演論文集, RB-002, 2009.

研究業績

論文誌論文

1. 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, Nov 2014.
2. 松本亮介, 岡部寿男,スレッド単位で権限分離を行うWebサーバのアクセス制御アーキテクチャ,電子情報通信学会論文誌 Vol.J96-B, No.10, pp.1122-1130, Oct 2013.
3. 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, Mar. 2013.

国際会議等発表（査読付）

1. Yoshiharu Tsuzaki, Ryosuke Matsumoto, Daisuke Kotani, Shuichi Miyazaki, Yasuo Okabe, A Mail Transfer System Selectively Restricting a Huge Amoount of E-mails, Workshop on Resilient Internet based Systems (REIS 2013), Dec. 2013.
2. Hiroki Okamoto, Ryosuke Matsumoto, Yasuo Okabe, Design of Cooperative Load Distribution for Addressing Flash Crowds Using P2P File Sharing Network, IEEE 37th Annual International Computer Software and Applications Conference (COMPSAC2013), July 2013.
3. Ryosuke Matsumoto, Yasuo Okabe, Access Control Architecture Separating Privilege by a Thread on a Web Server, The 12th IEEE/IPSJ International Symposium on Applications and the Internet (SAINT2012), pp.178-183, July 2012.

国内発表（査読付）

1. 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, インターネットと運用技術シンポジウム2013論文集, 2013, 79-86 (2013-12-5) , 2013年12月.

2. 松本亮介, 川原将司, 松岡輝夫, 汎用性の高い大規模共有型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, インターネットと運用技術シンポジウム2011論文集,2011,31-38 (2011-11-24), 2011年12月.

学会誌・商業誌等解説

1. 松本亮介・栗林健太郎, AI最前線の現場から【GMOペパボ】「なめらかなシステム」の取り組み, 連載: ディープラーニング・人工知能 最前線 2016, 2016年7月.
2. 松本亮介・栗林健太郎, 「GMOペパボ研究所」設立, ガチな学術研究でホスティングサービス差別化, 所長・栗林健太郎氏, 主席研究員・松本亮介氏, INTERNET Watch トピック 業界動向 企業 インタビュー, 2016年7月.
3. 松本亮介, 【特別企画】 快適・低価格・安全を実現した"次世代ホスティング"の秘密に迫る!, マイナビニュースIT企業 セキュリティ特別企画, 2016年5月.
4. 松本亮介, 楽しもうOSS開発 mrubyで学んだ貢献の流儀と情熱, WEB+DB PRESS Vol.85, pp.98-107, 技術評論社, 2015年2月24日.

口頭発表

1. 松本亮介, 自律制御するWebサービス基盤の権限分離と性能の両立, ペパボ・はてな技術大会@福岡, 2016年7月.
2. 松本亮介, ロリポップ！で目指すPHPのためのセキュリティと性能要件を同時に満たすサーバホスティング技術, PHPカンファレンス福岡2016, 2016年5月.
3. 松本亮介, セキュリティと性能要件を同時に満たすサーバホスティング技術の最新動向, 第39回インターネット技術第163委員会研究会 -ITRC meet39-, 2016年5月.
4. 松本亮介, cgroupとLinux Capabilityの活用 - rcon and capcon internals, 第9回 コンテナ型仮想化の情報交換会@福岡, 2016年4月.
5. 松本亮介, HTTP/2とmrubyの活用 HTTP/2時代のサーバ設定になぜmrubyが必要か, 第2回技術共有会, 2016年1月.
6. 松本亮介, PFSを考慮したTLS終端とngx_mrubyによる大量ドメイン設定の効率化 nginxにおける大量ドメインの証明書を動的に処理する方法, 第1回技術共有会, 2016年1月.

7. Ryosuke Matsumoto, The future of mruby in HTTP Server, RubyKaigi 2015, Dec 2015.
8. 松本亮介, mruby in HTTP server, connect HTTP/1 and HTTP/2 servers, 第1回 Hacker Tackle, 2015年9月.
9. 松本亮介, Middleware Configuration as Code, 第68回 Ruby関西 勉強会, 2015年8月.
10. 松本亮介, 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, Embedded Technology 2014スペシャルセッションC-2, 2014年11月.
11. Ryosuke Matsumoto, Resource Control Architecture for a Web Server Separating Computer Resources Virtually at Each HTTP Request, RubyKaigi 2014, Sep 2014.
12. 松本亮介, 今どきのWEBホスティングの高負荷対策, Hosting Casual Talks #1 at 株式会社はてな, 2014年6月.
13. 松本亮介, mod_mruby × ngx_mruby: 組込みスクリプト言語mrubyで高速かつ省メモリに拡張可能なWebサーバの機機能拡張支援機構, Ruby・mrubyビジネスセミナーFUKUOKA2014), 2014年2月.
14. 松本亮介, mod_mruby × ngx_mruby: 組込みスクリプト言語mrubyで高速かつ省メモリに拡張可能なWebサーバの機機能拡張支援機構, Ruby・コンテンツフォーラムFUKUOKA2014, 2014年2月.
15. 松本亮介, mrubyとWebサーバ技術の未来, Ruby東京プレゼンテーション2014 mrubyテクニカルセッション（パネルディスカッション2), 2014年1月.
16. 松本亮介, mrubyによるWebサーバ機能の拡張, Ruby東京プレゼンテーション2014 mrubyテクニカルセッション（パネルディスカッション1), 2014年1月.
17. 松本亮介, mod_mruby × ngx_mruby: 組込みスクリプト言語mrubyで高速かつ省メモリに拡張可能なWebサーバの機機能拡張支援機構, 第６回フクオカRuby 大賞, 2014年1月.
18. 松本亮介・岡部寿男, libcgroupとmrubyを使ったWebサーバのリソース制御アーキテクチャ, 第2回 コンテナ型仮想化の情報交換会, 2013年10月.
19. 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.
20. 松本亮介・岡部寿男, mrubyとWebサーバ, オープンソースカンファレンス名古屋2013, 2013年6月.

21. Ryosuke Matsumoto, Yasuo Okabe, Design and Implementation of Infrastructure Software for More Sophisticated Web Services, The 35th IST seminar, 11 April 2013 (in English).
22. 松本亮介・岡部寿男, mod_mruby, Ruby東京プレゼンテーション2013, 2013年3月.
23. 松本亮介・岡部寿男, Webサービスの高度化に耐えうる基盤設計に関する研究, 京都大学ICTイノベーション2013, 2013年2月.
24. 松本亮介・岡部寿男, 組み込みスクリプト言語mrubyを利用したWebサーバの機能拡張支援機構, 情報処理学会研究報告 Vol.2012-IOT-18, No.6, 2012年6月.
25. 松本亮介・岡部寿男, スレッド単位で権限分離を行うWebサーバのアクセス制御アーキテクチャ, 情報処理学会研究報告 - 第16回 インターネットと運用技術（IOT）, Vol.2012-IOT-16 No.13, pp. 1-6, 2012年3月.

受賞

1. 第8回 フクオカRuby大賞 奨励賞, 松本亮介, HTTPサーバを支えるmruby, 2016年2月.
2. 第7回 フクオカRuby大賞 企業賞（IIJ GIO賞）, 松本亮介, Trusterd: 高性能かつmrubyで設定をプログラマブルに記述可能な次世代HTTP/2 Webサーバ, 2015年3月.
3. Ruby Prize 2014 最終ノミネート, 松本亮介, HTTPサーバソフトウェアにおけるmruby活用を具現化した功績, 2014年10月.
4. 2014年度 情報処理学会 山下記念研究賞, 松本亮介, 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, 2014年10月.
5. 第9回 日本OSS奨励賞, 松本亮介, 博士課程在学中から研究成果をOSSとして公開するなどOSSの開発普及に貢献, 2014年2月.
6. 第6回 フクオカRuby大賞 優秀賞, 松本亮介, mod_mruby × ngx_mruby: 組込みスクリプト言語mrubyで高速かつ省メモリに拡張可能なWebサーバの機機能拡張支援機構, 第６回フクオカRuby 大賞, 2014年2月.
7. 情報処理学会 インターネットと運用技術シンポジウム2013（IOTS2013）優秀論文賞 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, インターネットと運用技術シンポジウム2013論文集, 2013, 79-86（2013-12-5）, 2013年12月.

8. 情報処理学会　インターネットと運用技術シンポジウム2013（IOTS2013）学生奨励賞，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，インターネットと運用技術シンポジウム2013論文集，2013，79-86（2013-12-5），2013年12月.
9. 情報処理学会 インターネットと運用技術研究会 第23回（IOT23） 学生奨励賞，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.
10. 2013年度 情報処理学会関西支部 支部大会 学生奨励賞，津崎善晴・松本亮介・小谷大祐・宮崎修一・岡部寿男，電子メールの大量送信を選択的に制限する中継システム，平成25年度情報処理学会関西支部支部大会 E-23，2013年9月.