Vol.2018-OS-144 No.9
2018/7/31

動的なコンフィグチューニングによるコンテナ型アプリケーションの性能最適化

千葉 立寛[1,a)]　中澤 里奈[1,b)]　堀井 洋[1,c)]

概要：Docker をはじめとするコンテナ型仮想化技術が広く普及してきた中で，公開レポジトリには誰でも利用可能なアプリケーションイメージが数多く登録されている．ユーザはそれらのイメージから気軽にコンテナをデプロイ出来るようになった一方，必ずしも全てのイメージで最適なコンフィグがデフォルトで設定されているとは限らないため，ユーザ側での適切なチューニングが必要となるケースも多々存在する．しかしながら，正しいチューニングを全てのユーザが行うことは困難であるため，自動的に設定をチューニングするシステムの存在が必要不可欠である．本稿では，コンテナ型アプリケーションの設定を自動的に最適化する仕組みを提案し，Kubernetes 上に実装したプロトタイプを紹介する．また様々なイメージ中に内在する性能に影響を与える設定の調査を行い，分析した結果を示す．

1. はじめに

システムソフトウェアには，ユーザがシステムの動作を後から変更出来るようにするため，数多くの設定項目 (以降，コンフィグ) が用意されている．ユーザは，ハードウェアを含む実行環境や実行するアプリケーション特性を考慮しつつコンフィグをチューニングするすることで，システムソフトウェアの最適化を自ら行うことが可能となっている．しかしながら，近年のシステム・ソフトウェアにおいては，数十から数百の単位でチューニング可能なコンフィグが存在していることも多い [1]．ユーザがそれらのシステムに対して常に十分な知識を持っていることは稀であり，無数にあるコンフィグの中から最も適したコンフィグを選び，なおかつ適切な値をセットすることは一般的には非常に困難であると言える．例えば，Hadoop においては数百ものコンフィグが存在し，それらの設定を間違えることによって発生するエラーの数は，ソフトウェアのバグよりも多いことが示されている [2]．またデータセンタにおける設定のミスが原因となり，サービスが停止するなどの大規模な障害を引き起こすケースも報告されている．例えば，Google におけるサービスが停止したイベントのうち，およそ 30%が誤ったコンフィグによって引き起こされており，その数は全体の中での 2 番目に多かったことが述べられている [3]．

このように間違ったコンフィグ（以降，ミスコンフィグ）がシステムに影響することは広く知られており，その原因や影響，修正方法についてこれまでにも様々研究されてきた [4][5][6]．十分な知識を持たずに多数のコンフィグを自らチューニングすることでミスコンフィグの発生に繋がり，その結果，システムの動作が変わり，実行性能が低下やシステムのデッドロック，Out of Memory Error によって実行そのものが出来ない状況を引き起こす可能性もある．さらには，アプリケーション実行開始時においては最適なコンフィグであったとしても，時間の経過とともにワークロードの傾向が変わるにつれて，最適でなくなる場合もある．このように，性能に影響を及ぼすコンフィグに関しては，アプリケーションの特性に応じてパラメータを調整していくことでミスコンフィグとならないようにしていく必要がある．

一方，近年のコンテナ型仮想化技術の発展により，非常に多くのアプリケーションがコンテナで動作するようになってきている．アプリケーションはコンテナイメージという可搬性が高い形式でパッケージングされており，システムを跨いでも実行環境を再現して容易に実行することが出来る．そのため，自分が作成したコンテナイメージをリポジトリを通じて他のユーザと共有したり，逆に他の人が作成したコンテナイメージをダウンロードしてすぐに利用することも可能となっている．コンテナランタイムとして最も利用されている Docker においては，そのイメージを共有するレポジトリ (DockerHub) で現在 100 万を超える

[1] 日本アイ・ビー・エム（株）　東京基礎研究所
IBM Research
[a)] chiba@jp.ibm.com
[b)] rina@jp.ibm.com
[c)] hhorii@jp.ibm.com

イメージがパブリックイメージとして公開されており，それらがダウンロード (Pull) される総数は，1 週間あたり数億回にも上る[*1]．ユーザはこれらのパブリックイメージをベースにし自分用のイメージとしてさらにカスタマイズすることも出来るが，ベースとなるイメージに存在するコンフィグやアプリケーションおよびシステム・ソフトウェアの特性を十分に理解していない場合，様々なコンフィグがデフォルトの値のままあったり，誰かが設定したミスコンフィグがそのままイメージに残り続けてしまい，結果として思わぬ性能低下を引き起こす可能性がある．コンテナとして手軽にアプリケーションが実行可能になった現在，これらに内在するシステム・ソフトウェアのコンフィグを実行前・実行後を問わず継続的かつ自動的にチューニングしていくシステムがこれからのクラウドネイティブなコンテナシステムにとって必須となっていくと我々は考えている．

本稿では，コンテナ型アプリケーションに内在するコンフィグおよびメトリクスを自動的に収集・解析し，様々なシステム・ソフトウェアに対して最適なコンフィグをユーザに提供・フィードバックするためのフレームワークを提案する．ヒューリスティクスベースおよび機械学習ベースの 2 つのアプローチでコンテナアプリケーションの状態に応じたコンフィグチューニングすることを目指したフレームワークであり，コンテナオーケストレーション基盤である Kubernetes 上に提案するフレームワークのプロトタイプ実装を行った．また，ウェブアプリケーション・サーバーの Liberty，および，NoSQL データベースの MongoDB の 2 つのコンテナにおいて，デフォルトコンフィグ値でスループット性能に影響を及ぼす例を紹介し，最適化ヒューリスティクスで用いるべきメトリクスについて検討を行った．

2. 背景

2.1 コンフィグチューニング

性能に直結するようなパフォーマンスセンシティブなコンフィグをチューニングするためには，そのシステムに対する深い知識が必要とされる．ヒューリスティクスベース [7][8]，性能モデルベース [9][10]，機械学習やベイジアン最適化などを用いてコンフィグを自動的にチューニングする研究 [11][12][13] など，状況に応じて様々なアプローチが取られている．例えば，Starfish [7] では，Hadoop 向けのコンフィグをプロファイル実行の結果を踏まえてヒューリスティクスベースで計算して，最適なコンフィグを提供している．また OtterTune [9] では，データベースシステム (MySQL，PostgreSQL) に対して，過去にチューニングしたワークロード特性をベースにモデルを作成し，データベース自体から得られるメトリクスから必要なフィーチャを取り出してガウス過程回帰を用いて最適なコンフィグを導出する仕組みを提案している．

これらの手法には一長一短がある．例えば，ヒューリスティクスベースの場合，真に最適なコンフィグに到達できない可能性があるが，条件にマッチしている場合はサンプリング等が必要ないため，高速に設定できる．一方，機械学習ベースの場合，真に最適なコンフィグまで到達できる可能性が高いが，実行プロファイルやパラメータサーチなどで時間がかかるため，コストをかけて最適なコンフィグを導いても結果的にはペイしない場合もある．そのため，コンフィグの種類やアプリケーションの状態に応じて，これらの手法を使い分けていくことが必要である．

2.2 コンテナ型仮想化

Linux におけるコンテナ型仮想化は，OS カーネル内の機能である namespace によるリソースの分離と，cgroups によるハードウェアリソースの制限を用いて，独立した環境をプロセス単位で構築することで実現されている．近年では，コンテナ型仮想化を実現するためのランタイムとして，HPC 向けの Singularity [14] や，後述する Kubernetes に特化して軽量化を目指す cri-o [15] ，ハイパーバイザを用いてコンテナ間でホスト OS のカーネルを共有しない形でコンテナを構築する frakti [16] など様々なコンテナランタイムが登場しているが，現在最もメジャーなランタイムは Docker であり，ラップトップからサーバまで幅広く使われている．

Docker が幅広く使われるようになった要因として，プロセスの隔離を行うだけでなく，そのプロセスがマウントするシステムのイメージをパッケージングして，様々な環境で確実に動作するような可搬性を用意したことが大きいと考えられる．コンテナのイメージは，UnionFS を用いて差分管理された CoW のレイヤーで構築されており，更新されたイメージを Pull した場合，以前に Pull したイメージの差分となるレイヤーだけをダウンロードするだけでよく，DevOps のサイクルと非常に相性が良い．このため，非常に低コストでコンテナイメージを共有して実行することが可能となっている．

公開されている様々なベースイメージから自分のコンテナイメージを誰でも自由に作成/公開できるようになった現在においては，実行するアプリケーションコンテナイメージの中に様々な脆弱性 (Vulnerability) が入り込む余地は以前にも増して高くなってきている．コンテナイメージにおけるセキュリティ脆弱性を検出するサービスが様々あるが [17][18][19] ，性能に影響を与えるようなミスコンフィグを検出し修正するようなサービスは我々が知る限り存在していない．また，マルチアーキテクチャ対応の Docker Image が登場したことにより，ユーザ視点では同一のイメージを複数のアーキテクチャ (amd64, ppc64le, arm64, etc.) で実行することが出来るようになった．コンフィグと

[*1] https://blog.docker.com/2018/06/day-1-keynote-highlights-dockercon-san-francisco-2018

いう視点では，アーキテクチャの違いまで意識してイメージを作成することは稀であるため，実行されるアーキテクチャに応じたコンフィグの最適化も求められる．

2.3 Kubernetes

Kubernetes[*2] は，クラスタ上にデプロイされた多数のコンテナアプリケーションを管理するためのオーケストレーションシステムである．Kubernetes は，Google でコンテナ型アプリケーションやサービスを管理するために利用されていた Borg [20] を前身とするシステムであり，現在では CNCF[*3]のもとでオープンソースで開発が進められている．コンテナのスケジューリングやロードバランス，CPU 負荷の状況に応じたオートスケール，何らかの原因で停止したコンテナの再配置やリスタート，サービスを無停止でコンテナイメージの更新を可能にするロールアウト機能など，実際にコンテナ型アプリケーションを用いてシステムを構築する際に必要な多くの機能を有しており，コンテナシステム全体に対する OS のような役割を担っている．Docker Swarm や Mesos なども同様の用途で用いられているが，現在では Kubernetes が最もメジャーなオーケストレーションシステムであると言える[*4]．

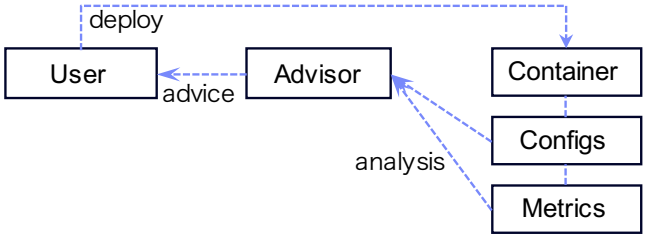
Kubernetes では，コンテナを Pod と呼ばれる単位で管理する．基本的には 1 アプリケーションコンテナにつき 1 Pod を用意するが，アプリケーションのヘルスチェックやメトリクスを取得するなどの機能を有するコンテナを同時に起動したい場合，Pod は network namespace を共有する単位でもあるため，これらを同一の Pod 内に置いたほうが実装上好ましい場合も多い．そのため，必要に応じて同じ Pod の中にこれらのコンテナをサイドカーコンテナとして用意することも出来るような設計になっている．

一方，様々なリソースおよびサービスを Kubernetes 上に定義し，Pod に紐づけて利用することができるが，Pod に対する設定をデプロイ時に調整する機能として，ConfigMap が用意されている．ConfigMap では，コンテナに対する設定ファイルを Kubernetes のリソースとして用意し，デプロイ時にコンテナにマウントすることで設定を反映することが出来る．本稿で提案するフレームワークにおいても，ConfigMap を使って設定を更新することを考えている．

3. フレームワークデザイン

本稿で提案するアドバイザーフレームワークが目指すゴールとしては，(1) コンテナおよびイメージに内在している性能に影響を与えるミスコンフィグを検知し，(2) ユーザのコンテナに対して適切なコンフィグをフィードバックし，(3) 性能を改善することである．このユーザへのフィードバックモデルを模式的に表したのが図 1 である．ミスコンフィグに加えて，コンテナアプリケーションから生成されるメトリクスを活用し，コンテナコンフィグへのフィードバックを行う．これら 3 つの機能を満たすため，解決すべき課題・要件を以下で論じ，フレームワークに必要なデザインを検討していく．

図 1 コンテナコンフィグチューニングのフィードバックモデル



3.1 ミスコンフィグの検出・解析

ミスコンフィグを正すためには，まず第一にどんなコンフィグがそのコンテナに存在しているかを検出・解析する必要がある．例えば MongoDB では，任意の場所に保存された mongod.conf を参照して，その動作設定を変えることが出来るが，動作中のコンテナもしくはイメージ内のディレクトリのどこかに存在する mongod.conf を探索し，さらにパースして設定されている情報を取得する仕組みが必要である．

また，アプリケーションコンテナは Kubernetes によってクラスタ上の任意のノードにデプロイされることを想定している．ユーザからアドバイスを要求されたタイミングでファイルの探索と解析を Just-In-Time で行う場合，ある程度のタイムラグが生じてしまう．そのため，フレームワークとしては，自動的にコンテナの開始を検知してアドバイス可能なコンテナ内のコンフィグを事前に探索および解析を行っていることが望ましい．

さらに，Liberty のように動的にコンフィグをロード出来るような仕組みを持ったアプリケーションでは，コンテナ起動後にコンフィグが設定・変更されることも考えられる．そのため，分析対象となるアプリケーションコンテナのコンフィグは，定期的/継続的に検出して変更点を追跡するような仕組みが必要である．

3.2 コンフィグチューニングの定義

次に，アプリケーションコンテナにおけるコンフィグチューニングの種類を定義する．性能に影響を与えるコンフィグは，そのコンフィグの性質やアプリケーションの状況により，

(1) コンテナ起動前に検出可能 (イメージ依存)
(2) デプロイ時に検出可能 (環境依存)

[*2] https://kubernetes.io/
[*3] https://www.cncf.io/
[*4] https://www.cncf.io/blog/2017/12/06/cloud-native-technologies-scaling-production-applications/

図 2　コンテナライフタイムにおけるコンフィグチューニング

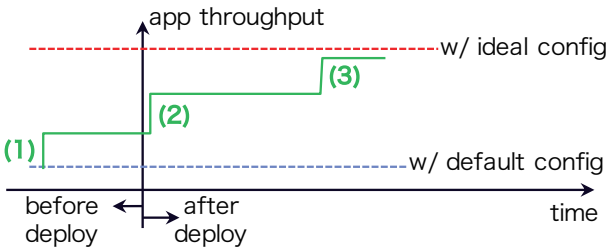
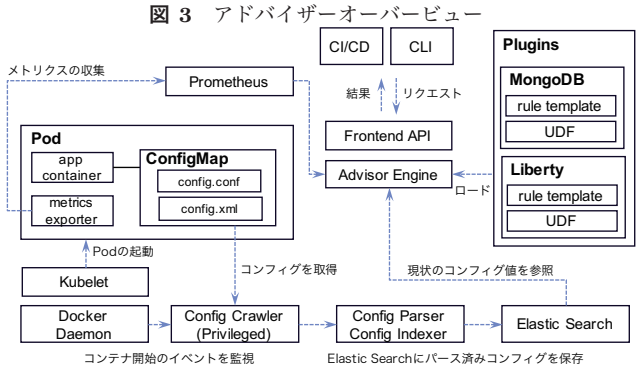


図 3　アドバイザーオーバービュー



(3) 一定時間後に検出可能 (ワークロード依存)

の 3 つに分類されると考えられる．図 2 は，コンテナライフタイム（デプロイ前からデプロイ後）における 3 つのフェーズにおいて，コンフィグチューニングによりアプリケーションの性能が向上する様子を表したものである．

JVM のヒープサイズを設定する例で考えてみると，-Xms の値が設定されていない場合，-Xmx の値と同じにするというアドバイスがコンテナ起動前に可能である．また，Pod に対してメモリクォータがかけられていてヒープの設定がそのクォータを考慮していなかった場合，-Xmx の値を CPU クォータの設定よりも小さな値 (例えば，90%) にセットするべきであるというアドバイスがデプロイ直後に可能である．さらに，コンテナが実行してしばらくした後にガベージコレクション (GC) の頻度や傾向を分析することで，-Xmn の値を増やすべきというアドバイスが可能となる．その結果として，GC の性能が改善され，アプリケーションのレスポンスタイムが改善されることに繋がる．これら 3 つのフェーズで発生するコンフィグに対するアクションをその時々に応じて対応出来るフレームワークが求められる．

3.3　コンフィグの調整

2.1 で言及した通り，ヒューリスティクスベースや性能モデルベース，機械学習やベイジアン最適化などを用いて最適なコンフィグの値を調整する方法に関しての研究が数多くなされている．これらの手法には一長一短があるが，アプリケーションのライフタイムに応じて変化するその時々で有効なコンフィグチューニング手法を適応的かつ必要に応じて選択しどちらもハイブリッドに実行出来るフレームワークにすることを目指している．単純なマッチングベースのヒューリスティクスでは，簡単なルールを記述するだけで実行でき，機械学習ベースでは，コンテナアプリケーションメトリクスを柔軟に分析出来るようにするべきと考える．また，システムごとにコンフィグ最適化のエキスパートナレッジおよびロジックを自由に入れ替えられる拡張性や出来るだけ低オーバーヘッドであることが機能として求められると考える．

4.　実装

3 節で挙げられた要件や課題をもとに，コンテナのコンフィグを調整するためのアドバイザーフレームワークのプロトタイプ実装を行った．図 3 は，本システムの全体像を示しており，フレームワークおよび利用するその他の基盤システム，ターゲットとなるコンテナを軸にして，データおよびオペレーションの流れを記載している．以下では，それぞれの要素についての説明を詳細に行っていく．

4.1　コンフィグクローラとパーサー

コンテナ内のコンフィグを取得するためのツールとして，Agentless System Crawler[*5] (以下，Crawler) を用いた．Crawler は，オープンソースで開発されている Python ベースのクローリングシステムであり，ユーザのコンテナ側への設定をすることなく，コンテナ内に存在するファイル等を取得し，Kafka や fluentd など様々なバックエンド向けに出力する機能を有している．プロトタイプ実装では，ホスト上の Docker デーモンのイベントを監視して，新たな Pod がデプロイされたこと動的に検知し，Liberty コンテナの場合は server.xml と jvm.options を，MongoDB コンテナでは mongod.conf を探索・取得して，後段に続くパーサーに Kafka 経由で渡している．

コンフィグをパースして分析するツールには，Augeas[*6] を用いた．Augeas 内部において，設定ファイルを構文解析して抽象構文木に変換する lens を用いて，アプリケーションのコンフィグを参照しやすい形に変形する．その後，各コンテナ・Pod に対応するコンフィグとして検索できるようにして ElasticSearch に保存する．

4.2　アプリケーションメトリクス

本稿で対象としているシステムソフトウェア多くは，そのシステムのモニタリング用途でメトリクスを Grafana 等に出力するエンドポイント (例えば，JMX) や，システ

*5　https://github.com/cloudviz/agentless-system-crawler
*6　http://augeas.net

ム内部の実行ログ (例えば，gc.log) をファイルに出力する機能を有している．これらの出力をパフォーマンス向上に向けたコンフィグチューニングに活かすため，本プロトタイプでは，これらのメトリクスを収集する Exporter を各アプリケーションコンテナのサイドカーコンテナとして同じ Pod 内に配置し，Prometheus に集約するようにした．Prometheus 内に貯められた Pod のメトリクス（時系列データ）をアドバイザーフレームワーク側で Pandas DataFrame に変換し，後述するプラグイン内で利用できるようにする．

4.3 アドバイザーフレームワーク

他のシステムとインタラクションを行うためのフロントエンド API と，ターゲットとなるコンテナのコンフィグを調整するためのアクションを行うエンジンの大きく分けて 2 つのコンポーネントで構成される．

フロントエンド API では，CI/CD やユーザとのインタラクションを通して対象となるコンテナの設定に対するアドバイスのリクエストおよび結果を送受信するための API を gRPC で実装した．

アドバイザーエンジンでは，フロントエンド API でのリクエストをもとに，後述する各システムに対応するプラグインのロードおよび実行，それらのプラグインで利用するデータ（現在のコンフィグデータ，クォータなどの実行環境データ，メトリクスデータ) を必要に応じて取得し，プラグインで利用出来るようにする．

4.4 プラグインとルールテンプレート

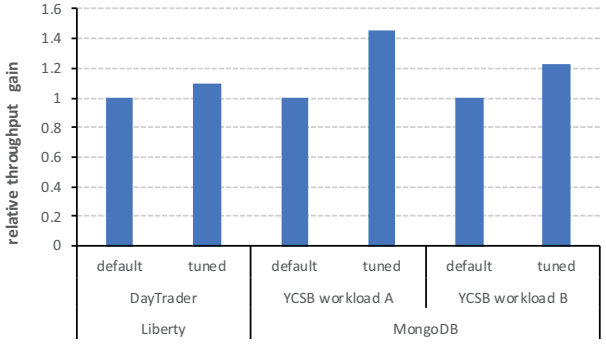
それぞれのシステム・ソフトウェアに対して，コンフィグを単純なルールベースでの調整およびメトリクスを元にした分析のそれぞれを，必要に応じて柔軟に実行できるよう，個々のシステム・ソフトウェアのコンフィグを調整するロジックをプラグインとして記述できるようにした．

プラグインでは，まず，アドバイスのベースになるテンプレートを JSON ファイルで記述する．図 4 は，Liberty プラグインで jvm.options で定義されるコンフィグのチューニングに用いるテンプレートの例である．cond(4 行目) および advice(7-8 行目) のところで，{{ }} (ブラケット) で囲まれた部分にアドバイスを行う条件およびアドバイスの値を計算する式を記述する．ブラケット内では current オブジェクトを介して現在の値を，advice オブジェクトを介して計算実行後の値を，それぞれ参照できるように実装している．cond エントリで定義された条件にマッチした場合のみ，アドバイスを計算するプロセスが動き，その結果が最終的にユーザに返される．ブラケットで囲まれた部分は，フレーム側で Python のテンプレートエンジンの Jinja2 で解釈され処理される．さらに，直近のメトリクスを分析して判断するといった単純なルールだけでは書き下せない処理を行いたい場合，プラグイン内に別途ユーザー定義関数 (UDF) を Python で定義して登録することで，ブラケット内での呼び出し (21 行目) が可能となるようにしている．

```
1  "jvm.options": [
2  {
3   "name":"mx",
4   "cond":"{{ current.mx > current.mem_requests }}",
5   "order":0,
6   "desc":"memory quota * 0.75",
7   "advice":"{{ current.mem_requests * 0.75
8               | round(1,'floor') | int }}"
9  },
10 {
11  "name":"ms",
12  "cond":"{{ current.ms != advice.mx }}",
13  "order":1,
14  "desc":"memory quota * 0.75",
15  "advice":"{{ advice.mx }}"
16 },
17 {
18  "name":"gcthreads",
19  "cond":"{{ current.gcthreads > current.cpu_cores }}",
20  "order":1,
21  "advice":"{{ my_udf(current.cores, current.usedheap) }}"
22 }
23 ]
```

図 4 コンフィグ/ライフタイムごとに定義するテンプレート例

図 5 メモリサイズに関するコンフィグを調整する前（デフォルト）と調整後でのスループットの比較 (左:Liberty，右:MongoDB)



5. コンフィグチューニング

本稿では，実装したシステムを用いてコンテナアプリケーションのコンフィグをアプリケーションのライフタイムに応じてチューニングするケーススタディとして，Liberty および MongoDB の 2 つアプリケーションにおけるコンフィグが性能に与える影響およびチューニングでの性能向上の余地に関して評価していく．

5.1 Liberty

Liberty は Java EE アプリケーションのためのランタイムであり，Tomcat に代表される Web アプリケーションフレームワークの一つである．本稿では，オンライントレーディングのワークロードを再現する DayTrader ベンチマークを利用し，JMeter を用いて Liberty に負荷をかけたとき

図 6 メモリクォータが設定された Liberty コンテナの DayTrader ベンチマークでのスループットと Global GC Interval の関係

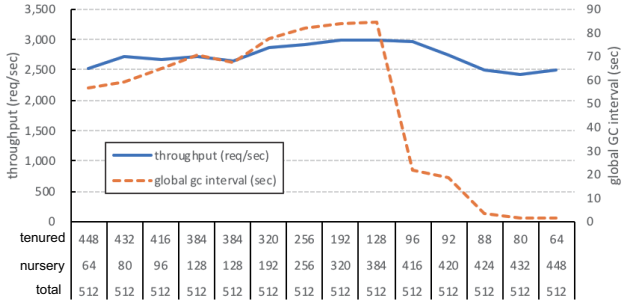
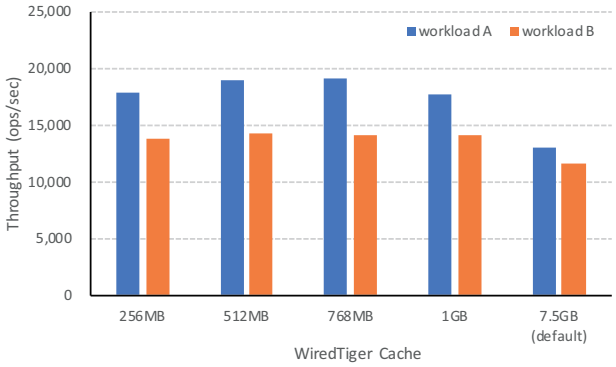


図 7 メモリクォータが設定された MongoDB における YCSB ベンチマークでのスループットと WiredTiger Cache の関係



の性能を評価した．Liberty コンテナは，POWER8 上に KVM で構築した VM(32 vCPU，メモリ 16GB，Ubuntu 16.04.02) にデプロイし，メモリクォータを 1GB および CPU クォータを 8 にそれぞれ設定した．また，Liberty が動作する JVM に対しては，512MB のヒープ (-Xmx512m) を設定した状態をデフォルトとしている．このとき，Nursery および Tenured の各ヒープ領域にはそれぞれ，128MB と 384MB が割り当てられる．

図 5(左) は，512MB のヒープの中で Nursery および Tenured ヒープの割当サイズをコンフィグチューニングの対象としたとき，デフォルト値とチューニング後で得られるスループットを比較したものであり，最大で 10%程度のスループット性能が向上している．Tuned の値は，パラメータの比率を実際に変えたときに得られた最も良い結果であり，Nursery をデフォルト値から増やすことでスループットが向上し，Nursery および Tenured それぞれ 384MB と 128MB のときに最大となった．

次に，これらのヒープの比率を様々変えたとき，アプリケーション実行時に得られるメトリクスのうちで，グローバル GC が発生する間隔との関係を示したものが図 6 である．Tenured ヒープサイズがある程度以下になった場合，グラーバル GC が発生する間隔が短くなり，アプリケーションが停止する時間が増えることで次第にスループット性能に影響を与えてしまうことが分かる．このことから，スループットを目標とする場合，Global GC Interval をメトリクスとして，Tenured ヒープを増やすようなヒューリスティクスを用いたアドバイスが可能であると言える．

5.2 MongoDB

MongoDB は，スキーマレスの NoSQL データベースである．本稿では，YCSB ベンチマークを用いて MongoDB に対して負荷をかけ，ワークロード特性に応じて，MongoDB におけるインメモリキャッシュ機構である WiredTiger のキャッシュサイズをコンフィグした場合のスループットへの影響を評価する．MongoDB コンテナは，Xeon E5-2683 v3 上に Xen で構築した VM(8 vCPU，メモリ 16GB，Ubuntu 16.04.04) にデプロイし，メモリクォータを 1GB に設定した．また，YCSB の設定として，データサイズ 1KB・レコード数 1M で Uniform なデータを生成し，オペレーション数 1M，クライアントスレッド数 8 で実行した．

図 5 (右) では，workload A (read/update=50/50) と workload B (read/update=95/5) において，WiredTiger Cache のデフォルト値と調整後のスループットを相対的に比較した結果である．MongoDB では，WiredTiger Cache がインメモリキャッシュとして使われているが，そのデフォルト値はシステムメモリに対する割合で計算され，コンテナに対するメモリクォータの値は考慮されない．今回実験に用いたノードの実メモリは 16GB あるため，デフォルトでは 7.5GB の WiredTiger Cache が設定されている状態として動作してしまう．しかしながら，メモリクォータによる制約から実際には 7.5GB のキャッシュは設定できないため，スラッシングが発生し，結果としてスループットが低くなっていまう．メモリクォータが設定されている場合にデフォルト値が性能低下を引き起こす例であり，環境依存のアドバイスとして提示可能である．

さらに，ワークロード特性に対して WiredTiger Cache の値がどの程度性能に影響を与えるかを調べる．図 7 は，1GB のメモリクォータ内で WiredTiger Cache のサイズを変更したときの YCSB ベンチマークの結果を示したものである．Read が支配的な Workload A では 768MB に設定したときが最も性能が良く，Read と Update の比率が 50% ずつの Workload B では，512MB のときが最も性能が良い結果であった．しかしながら，ベストのときの性能差が他の値を設定した場合と比べて大きくないことが分かる．そのため，アプリケーションのメトリクスを活用したアドバイスを行う場合，ワークロードのオペレーション数だけをメトリクスとするだけでなく，キャシュに関するメトリクスなど他のメトリクスも考慮すべきであるが，これらを利用したヒューリスティクスベースのアドバイスは今後の課題の一つである．

6. 関連研究

PerfConf [21] は，本稿で目指すゴールと同様に，パフォーマンスに影響を及ぼすコンフィグを調整するためのフレームワークを提案している．PerfConf では，ユーザプログラムに対して API を提供し，その API を通じてメトリクスに対する性能ゴールの設定を行い，対象となるコンフィグを動的に調整するが，対象となるシステム・ソフトウェアそのものを書き換えて API を呼ぶように変更する必要がある．

BestConfig [22] は，コンフィグしたい値のパラメータ空間を探索して性能最適化するためのフレームワークを提案している．一般的に，パラメータ空間を探索するためには多数のサンプリングが必要になるが，パラメータ空間を限定して効果的なサンプリング手法を組み合わせることで，同様のシステムに比べて少ないサンプル数で最適なコンフィグを導出している．

CherryPick [11] や Scout [12]，BOAT [13] などベイズ最適化を用いてコンフィグを調整し最適な性能を導出する研究がいくつかなされている．モデル化およびパラメータのサーチを繰り返し実行されるジョブの結果をもとに更新できるため，コンテナ化したアプリケーションのコンフィグ最適化に適した手法であると考える．本稿で提案するシステムにおいても，より最適な性能を達成するためにヒューリスティクスベースでの最適化後にベイズ最適化の要素を取り入れてより最適なコンフィグが出来るようにすることを考えている．

7. おわりに

本稿では，コンテナ型アプリケーションに内在するコンフィグおよびメトリクスを自動的に収集・解析し，ヒューリスティクスベースおよび機械学習ベースの両方のアプローチから，様々なアプリケーションのコンフィグをコンテナのライフタイムに応じて最適なコンフィグをユーザにフィードバックするためのフレームワークを提案し，Kuberenetes 上に実装したプロトタイプの説明を行った．また，Liberty および MongoDB の各コンテナに対してデフォルトコンフィグが性能に及ぼす影響を調査し，アドバイスのベースとなるメトリクスについて検討を行った．今後の課題としては，コンテナ起動前後の初期段階におけるヒューリスティクスの改善や，長期にわたって取得可能なメトリクスを活かしたコンフィグチューニング手法について考えていく予定である．

参考文献

[1] Xu, T., Jin, L., Fan, X., Zhou, Y., Pasupathy, S. and Talwadker, R.: Hey, You Have Given Me Too Many Knobs!: Understanding and Dealing with Over-designed Configuration in System Software, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, New York, NY, USA, ACM, pp. 307–319 (online), DOI: 10.1145/2786805.2786852 (2015).

[2] Rabkin, A. and Katz, R. H.: How Hadoop Clusters Break, *IEEE Software*, Vol. 30, No. 4, pp. 88–94 (online), DOI: 10.1109/MS.2012.73 (2013).

[3] Barroso, L. A. and Hoelzle, U.: *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*, Morgan and Claypool Publishers, 1st edition (2009).

[4] Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L. N. and Pasupathy, S.: An Empirical Study on Configuration Errors in Commercial and Open Source Systems, *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, New York, NY, USA, ACM, pp. 159–172 (online), DOI: 10.1145/2043556.2043572 (2011).

[5] Xu, T., Zhang, J., Huang, P., Zheng, J., Sheng, T., Yuan, D., Zhou, Y. and Pasupathy, S.: Do Not Blame Users for Misconfigurations, *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, New York, NY, USA, ACM, pp. 244–259 (online), DOI: 10.1145/2517349.2522727 (2013).

[6] Xu, T., Jin, X., Huang, P., Zhou, Y., Lu, S., Jin, L. and Pasupathy, S.: Early Detection of Configuration Errors to Reduce Failure Damage, *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA, USENIX Association, pp. 619–634 (online), available from ⟨https://www.usenix.org/conference/osdi16/technical-sessions/presentation/xu⟩ (2016).

[7] Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F. B. and Babu, S.: Starfish: A Self-tuning System for Big Data Analytics, *In CIDR*, pp. 261–272 (2011).

[8] Steinbach, C. and King, S.: Dr. Elephant for Monitoring and Tuning Apache Spark Jobs on Hadoop, Spark Summit 2017.

[9] Van Aken, D., Pavlo, A., Gordon, G. J. and Zhang, B.: Automatic Database Management System Tuning Through Large-scale Machine Learning, *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, New York, NY, USA, ACM, pp. 1009–1024 (online), DOI: 10.1145/3035918.3064029 (2017).

[10] Venkataraman, S., Yang, Z., Franklin, M., Recht, B. and Stoica, I.: Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics, *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, USENIX Association, pp. 363–378 (online), available from ⟨https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/venkataraman⟩ (2016).

[11] Alipourfard, O., Liu, H. H., Chen, J., Venkataraman, S., Yu, M. and Zhang, M.: CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics, *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA, USENIX Association, pp. 469–482 (online), available from ⟨https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard⟩ (2017).

[12] Hsu, C., Nair, V., Menzies, T. and Freeh, V. W.: Scout: An Experienced Guide to Find the Best Cloud Configuration, *CoRR*, Vol. abs/1803.01296 (online), available

from ⟨http://arxiv.org/abs/1803.01296⟩ (2018).
[13] Dalibard, V., Schaarschmidt, M. and Yoneki, E.: BOAT: Building Auto-Tuners with Structured Bayesian Optimization, *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, Republic and Canton of Geneva, Switzerland, International World Wide Web Conferences Steering Committee, pp. 479–488 (online), DOI: 10.1145/3038912.3052662 (2017).
[14] Kurtzer, G. M., Sochat, V. and Bauer, M. W.: Singularity: Scientific containers for mobility of compute, *PLOS ONE*, Vol. 12, No. 5, pp. 1–20 (online), DOI: 10.1371/journal.pone.0177459 (2017).
[15] cri-o: http://cri-o.io/.
[16] frakti: https://github.com/kubernetes/frakti.
[17] IBM Cloud: Managing image security with Vulnerability Advisor, https://console.bluemix.net/docs/services/va/va_index.html.
[18] Google Cloud: Google Container Registry Vulnerability Scanning, https://cloud.google.com/container-registry/docs/vulnerability-scanning.
[19] CoreOS: clair, Vulnerability Static Analysis for Containers, https://github.com/coreos/clair.
[20] Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E. and Wilkes, J.: Large-scale cluster management at Google with Borg, *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France (2015).
[21] Wang, S., Li, C., Hoffmann, H., Lu, S., Sentosa, W. and Kistijantoro, A. I.: Understanding and Auto-Adjusting Performance-Sensitive Configurations, *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, New York, NY, USA, ACM, pp. 154–168 (online), DOI: 10.1145/3173162.3173206 (2018).
[22] Zhu, Y., Liu, J., Guo, M., Bao, Y., Ma, W., Liu, Z., Song, K. and Yang, Y.: BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning, *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, New York, NY, USA, ACM, pp. 338–350 (online), DOI: 10.1145/3127479.3128605 (2017).