特定用途スケジューラの sched_ext による置き換え可能性の検証

平田 諒[1]　乃村 能成[2]

概要：一般的な Linux 環境で利用されるスケジューラは，各タスクに対し公平に CPU を割り当てる．これは，デスクトップ PC などの一般用途においては，十分な性能を発揮する．一方，アプリケーションや実行環境を考慮したスケジューリングで，より高い性能や効率を実現できる．このような特定用途のスケジューラを実現する Linux の機能として，sched_ext がある．しかし，sched_ext は既存技術と仕組みが大きく異なり，実装難易度や性能特性について不明な点も多い．そこで本稿では，過去に提案された特定用途のスケジューラを sched_ext で再実装し，置き換えの可能性について考察する．具体的には，既存技術である ghOSt と比較した実装の手間，性能，拡張性の高さについて比較した結果を報告する．

キーワード：Linux，sched_ext，ghOSt

1. はじめに

Linux では，複数のタスクに対して CPU をどのように割り当てるかを決定するためにスケジューラが用いられる．一般的な Linux 環境では，Completely Fair Scheduler（CFS）[1] などの汎用スケジューラが利用されており，計算機上の各タスクに対して公平な CPU 割当を行う．一方で，アプリケーションや実行環境によっては，その特性に応じたスケジューリングを行うことで，より高い性能や効率を実現できる場合がある．例えば，Function-as-a-Service（FaaS）型のサーバレス環境では，短時間で終了する多数の関数が同時に実行される．このような環境においては，公平なスケジューリングは頻繁なコンテキストスイッチを誘発し，タスクの実行時間を悪化させる恐れがある．これに対し，短時間で終了するタスクを FIFO 方式で実行し，長時間実行されるタスクを CFS に移行することで，コンテキストスイッチ削減とタスクの応答時間削減を両立する手法が提案されている [2]．

このような，特定用途に特化したスケジューラを実装するための仕組みとして，ghOSt[3] や sched_ext[4] がある．これらは，ユーザが作成したプログラムをスケジューラとして利用することを可能にする機能であり，汎用スケジューラでは困難な，アプリケーションや実行環境に特化したスケジューリングを実装できる．このうち ghOSt は，特定バージョンのカーネルに独自パッチを適用する形で実現されており，開発もすでに終了している．これに対し sched_ext は，Linux カーネルのメインラインにマージされた機能であり，パッチ適用や専用カーネルの用意が不要である．こうした違いから，sched_ext は ghOSt と比較して導入の敷居が低く，今後の特定用途スケジューラ実装において，主流の技術になっていくことが見込まれる [5]．

ここで，sched_ext はプログラムをカーネルの一部として動作させる仕組みであり，スケジューリング判断をユーザランドで完結させる ghOSt とはアーキテクチャが異なる．この差が実装難易度や性能特性にどう現れるかについては，よく知られていない側面も多い．特に過去提案された特定用途のスケジューラを sched_ext に置き換えて実装した例はまだほとんど報告されておらず，そのようなスケジューラを sched_ext へ置き換えることが実際にどの程度可能なのかは明らかではない．

そこで本稿では，過去に提案された特定用途のスケジューラ [2] を取り上げ，これを sched_ext に置き換えた事例を取り上げる．そして，この置き換えを通じ，sched_ext に関して以下の 3 点を明らかにする．

(1) ghOSt と sched_ext の実装難易度はどの程度異なるか
(2) sched_ext に置き換えた場合でも，元のスケジューラに求められる性能を発揮できるか
(3) sched_ext で置き換えたスケジューラについて，さらなる拡張・変更を行うのは容易か

[1] 岡山大学大学院環境生命自然科学研究科
[2] 岡山大学学術研究院環境生命自然科学学域

ⓒ2026 Information Processing Society of Japan

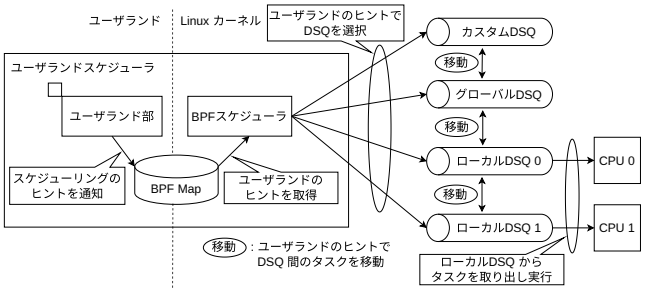

図 1: sched_ext を用いたスケジューラの全体像

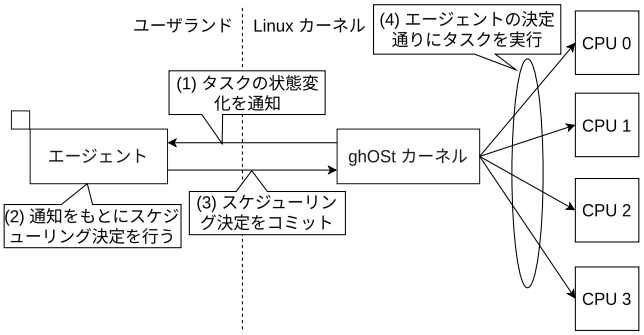

図 2: ghOSt を用いたスケジューラの全体像

2. 特定用途のスケジューラを実現する技術

2.1 sched_ext

sched_ext とは，2024 年に Linux カーネルのメインラインにマージされた，従来のカーネルスケジューラに代わりユーザが作成した BPF プログラムをスケジューラとして利用できる Linux の機能である．sched_ext を用いたスケジューラの全体像を**図 1** に示す．

sched_ext を用いてスケジューラを実装する際は，図 1 の要素のうち，ユーザランド部，BPF スケジューラを実装する．両者は BPF Map 経由で連携し，ユーザランド部がアプリケーションや環境固有の情報を BPF スケジューラに通知する．ユーザランド部は，アプリケーション自身がそれを兼ねる場合もある．BPF スケジューラは，キューへのタスク追加やキュー間のタスク移動といった，スケジューリング処理の各段階で呼び出されるハンドラを実装する．

これらのメカニズムにより，sched_ext を用いることで，特定用途のスケジューラを実装できる．

2.2 ghOSt

ghOSt とは，2021 年に Google により開発された，カーネル内部で行っていたスケジューリング決定をユーザランドのプログラムに委譲する仕組みである．ghOSt を用いたスケジューラの全体像を**図 2** に示す．

スケジューリングポリシを実装したプログラムは，通常のユーザランドプロセスとして動作し，エージェントと呼ばれる．ghOSt のカーネルは独自パッチを当てられた特殊なものであり，専用のスケジューリングクラスを提供する．

図 2 を用いて，ghOSt によるスケジューリングの流れを説明する．

(1) ghOSt カーネルは，タスクの状態変化 (生成・ブロック・wakeup など) をエージェントに通知する．
(2) エージェントは，ghOSt カーネルからの通知をもとにスケジューリング判断を行う．
(3) エージェントは，スケジューリング決定を ghOSt カーネルへとコミットする．
(4) ghOSt カーネルは，エージェントからのコミット内容に従い，指定された CPU でタスクを実行する．

(2),(3) について，エージェントがスケジューリング決定を行っている間にもタスクの状態は変化し得る．そのため，カーネル側のタスク状態とコミット内容が食い違っていると，そのコミットは失敗する．したがって，エージェントはカーネルと同期しながら動作する必要がある．

3. 研究のリサーチクエスチョン

1 章で述べた通り，sched_ext は今後，特定用途のスケジューラ実装において主流の技術になっていくと考えられる．2 章で述べた通り，sched_ext はプログラムをカーネルの一部として動作させる仕組みであり，スケジューリング判断をユーザランドで完結させる ghOSt とはアーキテクチャが異なる．しかし，この差がスケジューラの実装難易度や性能特性にどう現れるかについてはよく知られていない．特に過去考案された特定用途スケジューラを sched_ext に置き換えて実装した例はまだほとんど報告がなく，そのような置き換えを行った際に生じる困難・結果などは明らかでない．そこで本研究では，過去に考案された特定用途スケジューラを sched_ext に置き換えて実装し，以下 3 点を明らかにする．

RQ1: ghOSt と比較した際の実装難易度
sched_ext を用いてスケジューラを実装する際，ghOSt と比較して，実装の手間がどの程度異なるかは明らかでない．そこで，過去に ghOSt で実装されたスケジューラを sched_ext で置き換え，実装難易度を比較する（5.3 節）．

RQ2: 置き換えたスケジューラの性能
ghOSt と sched_ext はアーキテクチャが異なるため，両者の間で性能特性が異なる可能性がある．そのため，sched_ext に置き換えたスケジューラが，元のスケジューラに求められる性能を発揮できるかは明らかでない．そこで，ghOSt による既存スケジューラを sched_ext で再実装して追試評価を行い，元のスケジューラと同様の効果が得られるかを検証する（6 章）．

RQ3: 置き換えたスケジューラの拡張・変更しやすさ
特定用途のスケジューラは，対象ワークロードの特性に応じて何度も調整・変更されることが考えられる．

ⓒ2026 Information Processing Society of Japan

そのため，置き換えた後のスケジューラに対して変更を加えることが容易であるかは，実用上重要な観点である．そこで，sched_ext で置き換えたスケジューラに対し，さらなる拡張・変更を施すことの容易さを検証する（7 章）．また，拡張が意味のある変更であったことを確認するため，拡張後の性能も評価する．

以降では，これらの RQ を検証するためのケーススタディの設計（4 章），sched_ext によるスケジューラの置き換え（5 章），性能比較（6 章），スケジューラの拡張（7 章）について順に述べる．

4. sched_ext のケーススタディ設計

4.1 目的

3 章で述べた RQ を検証するためのケーススタディを設計する．ケーススタディとして，過去に ghOSt を用いて実装された特定用途スケジューラを取り上げる．つまり，スケジューリングの既存研究で提案されたスケジューラを sched_ext に置き換えるためのケーススタディと見立てる．

4.2 ケーススタディ(既存研究) の紹介

本研究では，以下の既存研究をケーススタディとする．

In Serverless, OS Scheduler Choice Costs Money: A Hybrid Scheduling Approach for Cheaper FaaS[2]

以下で内容を説明する．

概要

FaaS 型のサーバレス環境では，大規模なアプリケーションが短命でステートレスな関数に分割され，単一のマシン上で数千もの関数が同時に実行される．この高い同時実行性によりサーバは高稼働率を維持できる一方で，Linux 標準の CFS がタスクを頻繁に切替えるため，実行時間が長引き，ユーザのコストが増大する．この問題に対し既存研究では，短時間の関数を FIFO 方式でプリエンプションなしに実行し，長時間の関数を一定時間後に CFS へ引き渡すハイブリッドなスケジューリング手法を提案し，コストを大きく削減できることを示している．

サーバレスにおけるスケジューリングの問題

既存研究では，サーバレス環境は多数の短命な関数が同時に実行されるような環境であると説明している．具体的には，約 80% のタスクが 1 s 未満に終了するような環境である．このような環境では，公平性重視のスケジューラは頻繁なコンテキストスイッチを起こし，タスクの実行時間を不必要に増大させるという問題が指摘されている．また，タスクの実行時間に基づきミリ秒単位で課金が生じるため，ユーザにとって最大 10 倍のコスト増に繋がる．

提案手法

既存研究では，FIFO と CFS を組み合わせたスケジューリングを提案している．FIFO により，短時間で終了する関数をプリエンプションなしで実行し，コンテキストスイッチ回数を削減する．長時間関数については，FIFO による実行の終了後 CFS に引き渡し，FIFO のみの場合に生じる応答時間の長大化を防ぐ．

評価

既存研究では，ghOSt を用いてハイブリッドスケジューラを実装し，タスクの実行時間や応答時間，FaaS 利用コストについて，CFS と比較している．評価の結果，提案手法は CFS と比較して，タスクの実行時間，及び FaaS 利用コストを大幅に削減できることが示されている．評価の詳細については，追試の結果とあわせて 6 章で述べる．

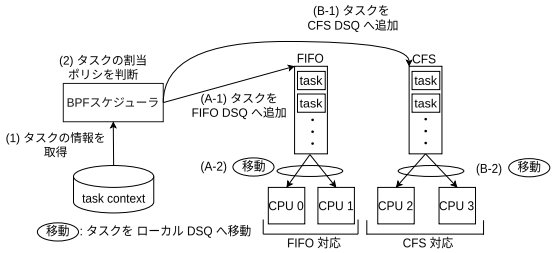

図 3: 既存研究のスケジューラを sched_ext で置き換えた際の構成及び処理流れ．あるタスクが runnable 状態になってから，CPU のローカル DSQ へ追加されるまでの流れを示している．CPU は全部で 4 つあり，CPU 0,1 は FIFO，CPU 2,3 は CFS に割り当てられた場合を想定．

5. sched_ext によるスケジューラの置き換え

5.1 概要

本節では，既存研究で提案されている FIFO と CFS のハイブリッドスケジューラを sched_ext で置き換えた際の概要について述べる．sched_ext で置き換えたスケジューラの構成を**図 3** に示し，以下で説明する．

BPF スケジューラ

DSQ へのタスク追加，DSQ 間のタスク移動，タスク実行開始/停止時のコンテキスト更新など，スケジューリング処理を担う機構．各スケジューリング処理は，5.2 節の 7 つのハンドラとして実装されている．

task context

各タスクのコンテキスト情報を保持した BPF Map．コンテキスト情報は，BPF スケジューラがスケジューリング処理に利用する．

FIFO DSQ

CPU 間で共有されている FIFO キュー．FIFO が割り当てられた CPU は，このキューからタスクを取り出し，自身のローカル DSQ へ移して実行する．

CFS DSQ

CPU 間で共有されている CFS キュー．新しくタスクが追加される際，各タスクに紐づく仮想実行時間 (それまで，タスクが CPU を掴んで実行した時間) に基

づいて，タスクをソートする．CFS が割り当てられた CPU は，このキューからタスクを取り出し，自身のローカル DSQ へ移して実行する．

CPU

タスクを実行する主体．各 CPU は専用のローカル DSQ を持ち，そこからタスクを取り出して実行する．

次に FIFO，CFS それぞれで，タスクが runnable 状態になってから CPU で実行されるまでの流れを説明する．

FIFO の場合

(1) BPF スケジューラは task context にアクセスし，runnable 状態になったタスクの情報を取得する．

(2) BPF スケジューラは，取得した情報から，タスクの割当ポリシが FIFO であると判断する．

(A-1)BPF スケジューラは，タスクを FIFO DSQ へ追加する．

(A-2)BPF スケジューラは，FIFO DSQ のタスクを FIFO 担当 CPU のローカル DSQ へ移す．

CFS の場合

(1) BPF スケジューラは task context にアクセスし，runnable 状態になったタスクの情報を取得する．

(2) BPF スケジューラは，取得した情報から，タスクの割当ポリシが CFS であると判断する．

(B-1)BPF スケジューラは，タスクを CFS DSQ へ追加する．

(B-2)BPF スケジューラは，CFS DSQ のタスクを CFS 担当 CPU のローカル DSQ へ移す．

タスクが CPU のローカル DSQ へ追加された後は，自動的に CPU がタスクを取り出し，実行する．また，図 3 には示していないが，CPU でタスクが実行開始されてからの処理流れは以下の通りである．

(1) CPU でタスクが実行開始されると，BPF スケジューラの `running()` ハンドラ (5.2 節) が呼び出される．

(2) CPU でタスクが実行停止すると，BPF スケジューラの `stopping()` ハンドラ (5.2 節) が呼び出される．

(3) タスクがまだ終了していない場合は，図 3 の (1) に戻る．終了した場合は，BPF スケジューラの `disable()` ハンドラ (5.2 節) が呼び出される．

5.2 詳細

図 3 の BPF スケジューラ実装にあたって，7 つのハンドラを実装した．以下に各ハンドラの処理内容を示す．

(1) `enqueue()`

処理内容: タスクの割当ポリシを判定し，該当 DSQ に追加する．図 3 の (1),(2),(A-1),(B-1) に対応．

呼出契機: タスクが runnable 状態となった際 (例えば，タスクが一度タイムスライスを使い切って CPU のランキューから外れた際)．

(2) `dispatch()`

処理内容: このハンドラ処理を実行した CPU のローカル DSQ へ，FIFO/CFS DSQ に入っているタスクを移動させる．図 3 の (A-2),(B-2) に対応．

呼出契機: CPU のローカル DSQ が空の際．

(3) `running()`

処理内容: ポリシに応じタスクの情報を更新する．FIFO のタスクについては何も更新しない．CFS のタスクについては，実行開始時点での CPU 累積実行時間の記録，CFS DSQ の仮想実行時間の更新などを行う．

呼出契機: タスクが実行開始する際．

(4) `stopping()`

処理内容: タスクを FIFO から CFS へ移行するかの判断を行い，その後各ポリシに応じてタスク情報を更新する．FIFO のタスクについては，まず CFS へ移行するか判断する．具体的には，実行開始してから一度も他タスクにプリエンプションされず，タイムスライスを使い切った場合に CFS へ移行する．その後タスク情報の更新を行う．CFS のタスクについては，タスクの CPU 累積実行時間を更新，CFS DSQ の仮想実行時間の更新などを行う．

呼出契機: タスクが実行停止する際 (タイムスライス切れによるプリエンプションや I/O 発行など)．

(5) `enable()`

処理内容: タスクの持つ情報を初期化する．

呼出契機: タスクが BPF スケジューラの管理下に入った際．

(6) `disable()`

処理内容: タスクの持つ情報のうち，評価に関わるものを BPF Map に書き出す．

呼出契機: タスクが実行終了し，BPF スケジューラの管理下から外れた際．

(7) `init()`

処理内容: BPF スケジューラを初期化し，FIFO DSQ，CFS DSQ を作成する．

呼出契機: BPF スケジューラのロード時．

5.3 ghOSt を用いた場合との実装難易度比較

本節では，sched_ext，ghOSt それぞれでハイブリッドスケジューラを実装した際の，両者の実装難易度を比較する．

比較方法

本稿では，難易度比較の評価軸として，コード行数を用いる．両者のコードを 1. ライフサイクル，2. データ構造，3. ロジックの 3 項目に基づいて分類し，各項目の行数をカウントする．

各項目について，1. はスケジューラのロード・起動・終了などを行う部分，2. はタスク・キューなど，スケジューリングで利用するデータの定義を行う部分，3. は実際のス

表 1: 各分類項目における，ghOSt，sched_ext それぞれのコード行数，および比率 (sched_ext / ghOSt)

No.	分類	sched_ext	ghOSt	比率
1	ライフサイクル	59 行	238 行	0.25
2	データ構造	97 行	282 行	0.34
3	ロジック	287 行	624 行	0.46

ケジューリング処理を担当する部分を示す．

比較に用いたコードは，それぞれ以下のリポジトリにて参照できる．

(1) sched_ext 版スケジューラ [6]

- 対象ファイル: `scx_replaced.bpf.c`, `scx_replaced.c`

(2) ghOSt 版スケジューラ [7]

- 対象ファイル: `hybrid_agent.cc`, `hybrid_scheduler.cc`, `hybrid_scheduler.h`, `cfs_scheduler.cc`, `cfs_scheduler.h`

比較結果

コード行数の比較結果を**表 1** に示す．なおコード行数は，コメント行や空行，デバッグ処理など，比較の必要がない箇所を除いてカウントした．

全項目において，sched_ext のコード行数が少ないことが分かった．これは，ghOSt がスケジューラをユーザランドプロセスとして実装する仕組みであるのに対し，sched_ext は作成したプログラムをカーネルの一部として動作させる仕組みであることに起因するといえる．例えば，データ構造定義について，sched_ext では，カーネル内の `task_struct` 構造体 [8] や DSQ をそのまま利用できる．一方 ghOSt では，これらを全て自前で定義する必要があるため，コード量が膨大になる．

スケジューリングロジックについても，ghOSt では，スケジューラがユーザランドで動作する都合上，スケジューリング以外にも考慮・実装すべきことが多くある (後述)．一方 sched_ext では，スケジューリングロジックのみをハンドラとして実装するのみで良い．

コード行数に差が生じる例として，各技術を用いた場合の，実行可能 CFS タスクの管理キューからタスクを 1 つ取り出し実行を依頼する処理の実装を **Listing1,2** に示す．なお，各コードはリポジトリ [6]，[7] から抜粋した．

Listing 1: (sched_ext) 実行可能 CFS タスクの管理キューからタスクを 1 つ取り出し実行を依頼する処理

```
1 void BPF_STRUCT_OPS(hybrid_dispatch, s32 cpu,
      struct task_struct *prev){
2   scx_bpf_dsq_move_to_local(GLOBAL_CFS_DSQ);
3 }
```

Listing 2: (ghOSt) 実行可能 CFS タスクの管理キューからタスクを 1 つ取り出し実行を依頼する処理

```
1  cs->run_queue.mu_.Lock(); //排他制御 (ghOSt ならでは)
2  CfsTask* next = cs->run_queue.PickNextTask(prev,
       cfs_allocator_.get(), cs);
3  cs->run_queue.mu_.Unlock(); //排他制御 (ghOSt ならでは)
4  cs->current = next;
5  if (next) {
6    req->Open({.target = next->gtid, //コミット処理 (ghOSt ならでは)
7               .target_barrier = next->seqnum,
8               .agent_barrier = agent_sw_last,
9               .commit_flags = COMMIT_AT_TXN_COMMIT})
       ;
10   while (next->status_word.on_cpu()) {
11     Pause(); //同期処理 (ghOSt ならでは)
12   }
13   uint64_t before_runtime = next->status_word.
       runtime();
14   if (req->Commit()) { //コミット処理 (ghOSt ならでは)
15     uint64_t runtime = next->status_word.runtime()
         - before_runtime;
16     next->vruntime += absl::Nanoseconds(static_cast
         <uint64_t>(static_cast<absl::uint128>(next->
         inverse_weight) * runtime >> 22));
17   } else { //コミット失敗時処理 (ghOSt ならでは)
18     cs->run_queue.PutPrevTask(next);
19   }
20 } else {
21   req->LocalYield(agent_sw_last, RTLA_ON_IDLE); //次のタスクがない場合の処理 (ghOSt ならでは)
22 }
```

Listing1, 2 より，同様の処理を実装する場合でも，ghOSt では 22 行必要なのに対し，sched_ext では 3 行で済むことが分かる．ghOSt では，スケジューリングに直接関わる処理以外にも，キュー操作の排他制御やカーネルとの同期処理，カーネルへのスケジューリング要求コミットを実装する必要がある．一方 sched_ext では，タスクを CPU の DSQ に入れておけば，自動でカーネルにより処理される．

このように sched_ext と ghOSt を比較した行数の差は，考慮・実装すべきことの少なさに起因している．このため，sched_ext は ghOSt より実装難易度が低いといえる．

6. 性能比較

6.1 概要

5 章のスケジューラについて，3 章の RQ2 を検証する．つまり，sched_ext で置き換えたスケジューラが，元のスケジューラに求められる性能を発揮できるかを明らかにする．

これらを検証するためには，既存研究と同一の観点から評価を行い，その結果を既存研究の評価結果と比較することが妥当と考えられる．既存研究と同様の傾向が得られれば，sched_ext に置き換えた場合でも元のスケジューラに求められる性能を発揮できていることが確認できる．そこで，以下 2 つの追試を行う．

追試 1: タスクの実行時間 (ET)，応答時間 (RT)，ターンアラウンド時間 (TAT) の比較

各タスクにおいて，以下 3 つの指標を計測する．

(1) 実行時間 (ET)
タスクの実行開始から終了までの時間．BPF スケジューラのハンドラのうち，`running()` ハンドラが初めて呼び出された時点から，`disable()` ハンドラが呼び出された時点までの時間を計測する．

(2) 応答時間 (RT)
タスクの実行準備が完了してから，初めて実行開始するまでの時間．BPF スケジューラのハンドラのうち，`enqueue()` ハンドラが初めて呼び出された時点から，`running()` ハンドラが初めて呼び出された時点までの時間を計測する．

(3) ターンアラウンド時間 (TAT)
タスクの実行準備完了から実行終了までの時間．ET と RT の和として計測する．

追試 2: FaaS 利用コストの比較
5 章のスケジューラを用いた場合の FaaS 利用コストを算出する．具体的な算出方法は 6.2 節で説明する．

6.2 詳細

比較対象

既存研究では，ハイブリッドスケジューラの比較対象として CFS を利用していたが，本稿では，sched_ext スケルトンを用いる．これは，CFS と同様のスケジューリングを行う sched_ext スケジューラである．追試 1 でのロギング処理を統一するため，この方式を用いる．

ワークロード

追試では，既存研究と同様のワークロードを用いる．ワークロードは，「一定時間のスリープと CPU バウンドタスクの起動を繰り返す」もので，実際のサーバレス環境における処理の到着パターンと処理時間分布をスリープと CPU バウンドタスク (フィボナッチ計算) で模倣している．スリープとタスク起動の間隔は，**Listing3** に示す形式のファイルにより決まる．

Listing 3: スリープとタスク起動の間隔を決めるファイル

```
1 t1, n1
2 t2, n2
3 ...
```

このファイルは，実際の Azure 環境における関数呼出のトレースデータ [9] をもとに作成される．上記ファイルを参照し，t_1 秒だけスリープ $\rightarrow \mathrm{fib}(n_1)$ を起動 $\rightarrow t_2$ 秒だけスリープ $\rightarrow \mathrm{fib}(n_2)$ を起動 $\rightarrow \cdots$ という処理を繰り返す．フィボナッチタスクの起動は非同期に行われるため，直前に起動されたフィボナッチタスクの終了を待たず，次々に新たなフィボナッチタスクを起動する．これにより，サーバレス環境特有のバースト性を模倣している．なお，既存研究では，12,442 回のタスク呼出を行うワークロードを利用しているが，今回は実験に用いた計算機に合わせてタスク呼出の回数を 10,000 回にダウンスケールした．また，FIFO のタイムスライスは，ワークロード内に含まれるタスクの p90 実行時間である 1,633 ms とした．

表 2: 実験環境

項目	内容
CPU	Ryzen 9 5950X
コア数	32
Memory	64GB
Kernel	Linux 7.0.0

追試 2: FaaS 利用コストの比較

FaaS 利用コストは，既存研究と同様，ワークロードを実行した際の 1 ms あたりの期待単価と，各タスクの総実行時間を掛け合わせることで算出する．

ワークロード中の各関数が利用するメモリサイズの割合 r_i と，各メモリサイズに対応する単価 p_i (USD/ms)[10] を用い，1 ms あたりの期待単価 $\bar{p}$ を求める (**式 1**)．

$$\bar{p} = \sum_i r_i \, p_i \tag{1}$$

この $\bar{p}$ と，各スケジューラにおけるタスクの実行時間の総和 T (ms) を掛け合わせ，コスト C を求める (**式 2**)．

$$C = T \cdot \bar{p} \tag{2}$$

環境

追試で用いた環境を**表 2** に示す．コアの割当について，FIFO/CFS それぞれに 15 コアずつ，ワークロードジェネレータなどに 2 コア割当てた．

6.3 結果

追試 1: タスクの ET，RT，TAT の比較

図 4 に既存研究の結果，**図 5** に追試の結果を示す．また，各図から分かることを以下に示す．

(1) sched_ext で置き換えたスケジューラでも，元のスケジューラに求められる性能を発揮できている
追試 1 で計測した 3 指標いずれも，既存研究と同様の傾向になっていることが分かる．まず ET について，既存研究，追試どちらも，ハイブリッドスケジューリングの方が良い結果を示した．このことより，sched_ext に置き換えたスケジューラでも，FIFO 実行によるコンテキストスイッチ削減を再現できたことが分かる．次に RT について，既存研究，追試どちらも，CFS の方が良い結果を示した．どちらの評価でも，CFS が公平性を維持するためにタスクを頻繁に切り替える特性が結果に現れている．最後に TAT について，既存研究，追試どちらも，ハイブリッドスケジューリングの方が良い結果を示した．どちらの評価でも，ハイブリッドスケジューラのタスクは，テール値が CFS より小さくなった．

ⓒ2026 Information Processing Society of Japan

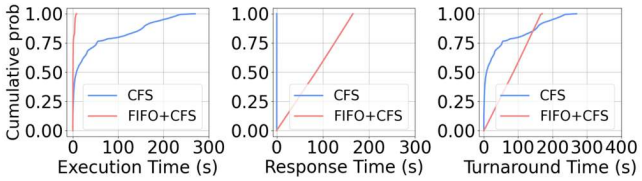

図 4: 既存研究の提案手法と CFS それぞれにおける，ET，RT，TAT の比較結果を示した累積分布．横軸は時間，縦軸は累積比率．赤線は提案手法，青線は CFS．

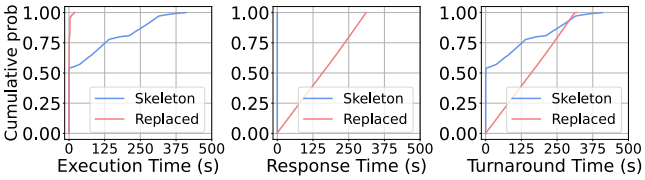

図 5: 置き換えたスケジューラとスケルトンそれぞれにおける，ET，RT，TAT の比較結果を示した累積分布．各軸は図 4 と同様．赤線は置き換えスケジューラ，青線はスケルトンで，それぞれ図 4 の提案手法，CFS に対応．

表 3: 既存研究，追試それぞれにおける FaaS 利用コスト

スケジューラ	コスト (既存研究)	コスト (追試)
CFS or Skeleton	4.51 USD	10.4 USD
CFS + FIFO or Replaced	0.11 USD	0.11 USD

追試 2: FaaS 利用コストの比較

FaaS コストの比較結果を**表 3** に示す．既存研究と同様，ハイブリッドスケジューリングにより，FaaS コストを大幅に削減できた．

追試 1,2 どちらも既存研究と同傾向の結果となり，sched_ext で置き換えたスケジューラは，元のスケジューラに求められる性能を発揮できていることが分かった．

7. スケジューラの拡張

7.1 概要

本章では，3 章の RQ3 を検証する．つまり，sched_ext で置き換えたスケジューラに対し，さらなる拡張・変更を施すことの容易さについて調査する．

6 章での追試評価を通じ，よりサーバレス環境に適したスケジューリングを行う着想を得たため，それを RQ3 の事例に見立てる．以下にスケジューラの拡張方針を示す．

拡張方針: タスク初回投入時のみランダムな CPU で短時間実行した後 FIFO/CFS スケジューリングへ移す

サーバレスワークロードは，非同期に起動されるフィボナッチタスクの群で構成されており，大半は短時間で終了する．しかし，置き換えたスケジューラは全てのタスクを最初に FIFO キューへ追加するため，ごく短時間で終了するタスクでも，先行する FIFO タスクのタイムスライス分 (本稿では 1,633 ms) 待たされる可能性がある．これを緩和するため，タスクの初回実行のみ短時間だけタスクを実行する．これにより，極端に短いタスクについては，FIFO のタイムスライス分の待ちを回避でき，RT のさらなる削減が期待される．初回の割込実行の CPU はランダムに決定する．短時間の割込であれば，すでにその CPU 上にある他タスクへの影響は小さく，逆に FIFO/CFS のどちらかに偏らせると，その CPU 上のタスクへの影響が大きいため，あえてランダムに分散させる．

7.2 スケジューラの拡張

7.1 節の方針に基づき，スケジューラを拡張した．具体的には，以下 3 つについて処理を拡張した．

(1) task context

拡張内容: 各タスクのコンテキスト情報として，一度でも実行されたかどうか判定するフラグを追加した．

(2) `enqueue()`

拡張内容: タスクが初回実行の場合，CPU をランダムに選択し，割込処理用タイムスライスをセットして DSQ の先頭に追加するようにした．

(3) `stopping()`

拡張内容: タスクが初回実行の場合，割込実行の終了後，(1) のフラグを更新するようにした．

Listing 4: スケジューラの拡張により追加したコード

```
1  // task context への拡張
2  u64 at_least_once_run;
3
4  // enqueue() への拡張
5  if (!tctx->at_least_once_run) {
6    __u32 random_cpu = bpf_get_prandom_u32() %
         NR_CPUS;
7    scx_bpf_dsq_insert(p, SCX_DSQ_LOCAL_ON |
         random_cpu, SCX_INTERRUPT_TS, SCX_ENQ_HEAD);
8    return;
9  }
10
11 // stopping() への拡張
12 if (!tctx->at_least_once_run) {
13   tctx->at_least_once_run          = 1;
14 }
```

拡張を行うにあたり追加したコードを **Listing4** に示す．

Listing4 に示す通り，拡張に伴い追加したコードは合計 9 行だった．本拡張では，ハイブリッドスケジューリングのロジック自体は変更しておらず，初回実行用の分岐を追加するのみで実現できた．これは，sched_ext では，タスクを DSQ に追加する際の追加位置，DSQ 種類，タイムスライスなどを引数で一度に指定できるためである．また，タスクに持たせる情報の追加も，BPF Map にメンバを追加するだけで良い．一方，ghOSt で同様の拡張を行う場合は，5.3 節で述べたコミット処理や同期処理を踏まえた実装が必要になり，より多くの行数を要すると思われる．

ⓒ2026 Information Processing Society of Japan

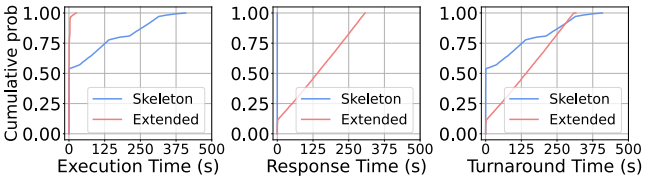

図 6: 拡張したスケジューラとスケルトンそれぞれにおける，ET，RT，TAT の比較結果を示した累積分布．各軸は図 4 と同様．赤線は拡張スケジューラ，青線はスケルトン．初回割込のタイムスライスは 20 ms とした．

7.3 拡張機構の評価

概要

拡張機構が有効に動作する場合，ごく短時間で終了するタスクの RT・TAT を改善できることが期待される．そこで，6 章の追試 1 を再度行う．

結果

拡張したスケジューラでの計測結果を**図 6** に示す．また，図から分かることを以下に示す．

(1) 一部タスクの RT，TAT をさらに改善できている
図 5 と比較して，図 6 の RT，TAT は，約 12.5% のタスクがほぼ 0 s となっている．これは，約 12.5% は 20 ms 以内に終了するタスクだからである．既存研究では，最初からタスクを FIFO で処理する都合上，そのタイムスライス分の待ちが最低でも生じるが，本拡張では，極端に実行時間が短いタスクは FIFO に回さず処理できる．7.1 節の拡張により，一部タスクの RT，TAT をさらに改善できることが分かった．

8. おわりに

本研究では，過去に ghOSt で実装された特定用途のスケジューラを sched_ext に置き換えることの可能性を検証し，3 章で示した 3 つの RQ について以下の知見を得た．

RQ1 については，ghOSt と sched_ext を比較した結果，sched_ext の方がコード行数の観点で実装難易度が低いことが分かった．その主な理由は，sched_ext がカーネル内のデータ構造を使い回せること，スケジューリングロジック以外の処理を考慮する必要がないことにあった．

RQ2 については，6 章の追試の結果，既存研究と同様の傾向が得られることを確認した．このことから，ghOSt を用いて実装された特定用途のスケジューラは，sched_ext に置き換えた場合でも，元のスケジューラに求められる性能を十分に発揮できることが分かった．

RQ3 については，初回の割込実行を行う機構を追加した場合における実装の手間を調査した．調査の結果，本拡張においては，sched_ext の機能を活用し，少ないコード行数で拡張を実現できた．また，拡張機構を評価し，一部のタスクの RT，TAT がさらに改善されることを確認した．

今後の課題として，sched_ext のアーキテクチャ及び実装に起因して性能が低下し得る状況の調査がある．本研究の追試では既存研究と同傾向の結果が得られたが，これは本研究のワークロードや環境において，ボトルネックが顕在化しなかっただけである可能性がある．実際，コア数やタスク到着頻度が増加した場合，BPF ハンドラの呼出頻度や DSQ へのアクセス回数が性能へ与える影響は，本研究では評価していない．そのため，性能低下が生じ得る状況を特定し，その要因と sched_ext のアーキテクチャ及び実装との因果関係を明らかにする必要がある．

9. 謝辞

研究において多くのご指導，ご意見をいただきました，富士通株式会社の関係者の皆様に深く感謝を申し上げます．

参考文献

[1] Linux Foundation: CFS Scheduler - The Linux Kernel documentation, Linux Foundation (online), available from ⟨https://docs.kernel.org/scheduler/sched-design-CFS.html⟩ (accessed 2026-10-30).

[2] Zhao, Y., Weng, W., van Nieuwpoort, R. and Uta, A.: In Serverless, OS Scheduler Choice Costs Money: A Hybrid Scheduling Approach for Cheaper FaaS, ACM Proceedings of Middleware '24, p. 172–184, DOI: 10.1145/3652892.3700757 (2024).

[3] Humphries, J. T., Natu, N., Chaugule, A., Weisse, O., Rhoden, B., Don, J., Rizzo, L., Rombakh, O., Turner, P. and Kozyrakis, C.: ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling, *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, New York, NY, USA, p. 588–604, DOI: 10.1145/3477132.3483542 (2021).

[4] Linux Foundation: Extensible Scheduler Class - The Linux Kernel documentation, Linux Foundation (online), available from ⟨https://docs.kernel.org/scheduler/sched-ext.html⟩ (accessed 2026-10-30).

[5] sched_ext: scx, GitHub (online), available from ⟨https://github.com/sched-ext/scx/blob/main/OVERVIEW.md⟩ (accessed 2026-10-30).

[6] nomlab: scx_replaced, GitHub (online), available from ⟨https://github.com/nomlab/scx_replaced/tree/c045bdc/scheds/c⟩ (accessed 2026-10-30).

[7] ZhaoNeil: hybrid-scheduler, GitHub (online), available from ⟨https://github.com/ZhaoNeil/hybrid-scheduler/tree/d13affa/schedulers/hybrid⟩ (accessed 2026-10-30).

[8] bootlin: sched.h - include/linux/sched.h - Linux source code v7.0 - Bootlin Elixir Cross Referencer, bootlin (online), available from ⟨https://elixir.bootlin.com/linux/v7.0/source/include/linux/sched.h#L820⟩ (accessed 2026-10-30).

[9] Azure: AzurePublicDataset, GitHub (online), available from ⟨https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsDataset2019.md⟩ (accessed 2026-10-30).

[10] AWS: AWS Lambda Pricing, AWS (online), available from ⟨https://aws.amazon.com/lambda/pricing/⟩ (accessed 2026-10-30).

ⓒ2026 Information Processing Society of Japan