Vol.2015-OS-135 No.14
Vol.2015-EMB-39 No.14
2015/11/24

SSDの並列性を引き出すI/Oスケジューラ

奥村 開里[1,a)]　小林 諒平[1,b)]　吉瀬 謙二[1,c)]

概要：近年，Solid State Drive(SSD) は個人用のパソコンのみならず，クラウドストレージ，データセンターなどといった幅広い範囲で使われ始めている．SSD は性能向上のために，複数チャンネル，またチャンネル毎に存在する複数のチップによって I/O の並列処理を行い性能を向上させているが，それらを考慮した SSD 用のスケジューラは OS 側に組み込まれていない．そのため本稿では，SSD の並列性を抽出することにより，レイテンシの低減，及びスループットの向上を目的とする Alleviate Conflict(AC) スケジューラを提案する．Linux に提案するスケジューラを実装し，SSD に対する様々な I/O リクエストパターンを用いて，SSD の帯域幅とレイテンシを評価した．その結果，Web サーバに近い I/O アクセスパターンにおいては，提案した I/O スケジューラは，Linux カーネルで標準的に使用されている Noop スケジューラ，Deadline スケジューラ，CFQ スケジューラそれぞれと比較し，Noop スケジューラからは帯域幅 4%の向上，レイテンシは 15%の低減，Deadline スケジューラからは帯域幅 7%の向上，レイテンシは 7%の低減，CFQ スケジューラからは帯域幅 34%の向上，レイテンシは 40%の低減を達成した．

キーワード：SSD，I/O スケジューラ，Linux，Kernel， OS

1. はじめに

近年 Solid State Drive(SSD) は，個人用の PC のみならず，クラウドストレージやデータセンターなどといった企業向けのシステムでも採用が進んでいる．SSD は Hard Disk Drive(HDD) と比べ機械的な部品を持たず耐衝撃性が高いことや，レイテンシが低い，高い帯域幅を持つなどといった利点が存在する．しかし利点ばかりではなく，記憶素子として使われている NAND フラッシュメモリーは HDD と比べて寿命が短い [1]，HDD などのストレージと比較して容量あたりの価格が高額といった欠点も存在する．また，SSD では，書き込み回数を平均化させるウェアレベリングといった機構や，ガーベジコレクションといった HDD には存在しない機構が組み込まれている．

上記のように HDD と SSD には無視できない大きな違いがある．当初，SSD は HDD の置き換えという目的で開発されたため，HDD 様に書かれたカーネルのコードで SSD を動作させることができた．しかし，SSD の性能を引き出すためには SSD の特徴を活かした OS 側のサポートが必要であるが，未だに Linux Kernel には HDD 用に書かれたコードが多く残っている．その為に HDD から SSD へストレージを入れ替えても，多数のリクエストを処理する Web サーバのような I/O がボトルネックとなる環境においては，その恩恵を十分に享受できない．この問題に対処するために，これまではソフトウェア側のアプローチよりも，RAID の構築などのハードウェア側のアプローチが採用されてきたが，このようなアプローチは時間的・経済的にコストが大きい．

このため我々は，OS 側からの SSD へのサポートの強化による I/O 性能の向上を目指す．そのためのアプローチとして，I/O リクエストのソート・マージによるスループットの向上やレイテンシの低減といった役割を担う Linux カーネルにおける I/O スケジューラに着目する．I/O スケジューラは Linux Kernel においてモジュール化がなされており，すでに稼働しているシステムにおいて再起動を必要とせずに組み込むことが可能である．したがって，時間的にも経済的にも負担とならない解決案となりうる．また，現行のカーネルにおいては，No-operation(Noop) スケジューラ，Deadline スケジューラ，Completely Fair Queueing(CFQ) スケジューラと 3 つのスケジューラが標準では組み込まれているが，どれも SSD 向けに設計されたスケジューラではなく，大幅な性能の改善が見込める．

本稿では，SSD の性能を引き出すための I/O スケジューラである Alleviate Conflict(AC) スケジューラを提案する．AC スケジューラは SSD 内の並列性を抽出する事によって性能の向上を見込むスケジューラである．AC スケジューラを Linux カーネル上に実装し，様々なアクセスパターンで帯域幅とレイテンシを評価した．その結果，既存の I/O スケジューラと比較して，AC スケジューラはほぼ全てのアクセスパターンにおいてレイテンシ，スループットを改善することが分かった．

本稿の構成は以下の通りである．第 2 章においては，今回の提案手法において着目する SSD の特徴と，現行の問題点を述べる．第 3 章においては，参考にした研究について述べる．第 4 章においては，提案手法の詳細を述べる．第 5 章においては，実験を行った環境及び，実験結果，考

[1] 東京工業大学 大学院情報理工学研究科
[a)] okumura@arch.cs.titech.ac.jp
[b)] kobayashi@arch.cs.titech.ac.jp
[c)] kise@cs.titech.ac.jp

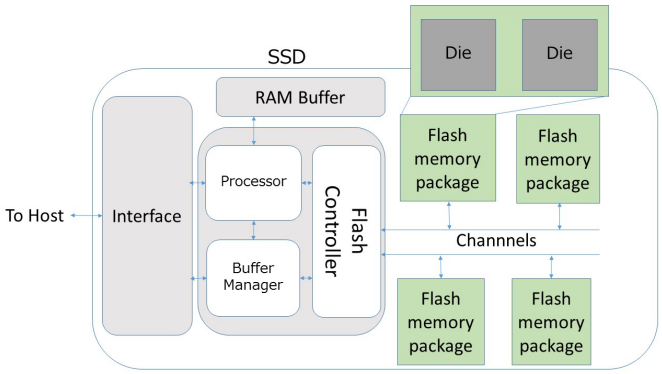

図 1 SSD の構造図

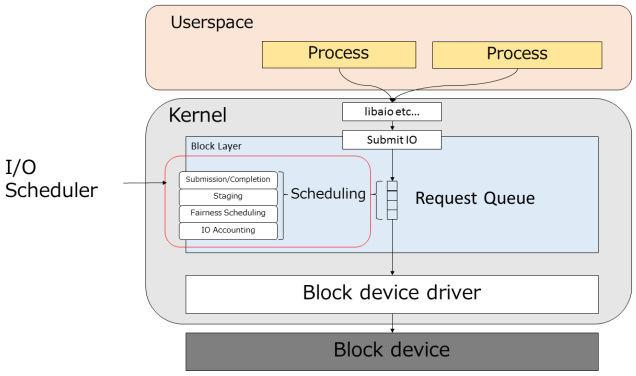

図 2 I/O スケジューラの概念図

察と今後の方針について述べる．第 6 章においては，今回の提案手法についての総括を述べる．

2. SSD と I/O スケジューラ

提案手法において考慮した SSD の性質，一般的な I/O スケジューラが果たす役割，I/O リクエストの種類，現行のスケジューラの問題点について述べる．

2.1 Solid State Drive

SSD では記憶素子として NAND フラッシュメモリーを用いている．しかしそれぞれの記憶素子自体の速度はそこまで高くなく (32MB/s-40MB/s)[1]，並列処理する機構を組み込むことによって性能を向上させている．

2.1.1 SSD 内部の並列性

SSD には以下の様な並列性が存在する [2], [3].

- チャンネルレベルでの並列性:フラッシュコントローラーは複数のチャンネルとデータの送受信を行う．その際に，それぞれのチャンネルが独立に，また同時にアクセス可能なこと（図 1）.
- パッケージレベルでの並列性:同チャンネルに属するパッケージ内で独立してアクセス可能なこと（図 1）.
- ダイレベルでの並列性:パッケージに含まれている複数ダイ（2～4）それぞれに同時にアクセス可能なこと（図 1）.
- プレーンレベルでの並列性:ダイ内に存在する複数プレーン（2～4）に同操作（Read，Write，Erase）ならば同時に行えること．

これらの並列性を効率良く利用できる I/O リクエストが SSD に対して適していると言える．

また，SSD の書き込み操作と読み込み操作は，ページ単位で処理することができるが，削除操作及び上書き操作においては，ブロック単位でしか処理が行えないという欠点がある．よって SSD においては，読み込みコストと比べて，書き込みコストが高い．具体的には，Read とくらべて Write が著しく遅いといった特徴や，ライトアンプリフィケーションと言った状態 (実際の write よりも多くのデータが書き込まれるという好ましくない状態) が発生しうる事などの特徴がある [1], [2].

また，書き込みと読み込みはプレーンレベルにおいて同時に行うことができず，リクエスト間で資源の競合が起こるので，できるだけ読み込みの操作と書き込みの操作を別々に行う事が好ましい．例えば read のみをまず先に行う readahead ストラテジーなどを行えば性能が向上する [3], [4].

また，上記の SSD 内部の並列性を抽出するために，フラッシュコントローラが大きな役割を果たしている．しかし，フラッシュコントローラの役割は SSD 内部の並列性の抽出のみではない．SSD の記憶素子には書き込み回数の制限が存在するので，それぞれの書き込みを平均化させ，SSD の寿命を延ばす為のウェアレベリングという機構や，使われていないセクタを予め使用可能にしておくガーベジコレクションと言った機構がフラッシュコントローラによって実現されている．

また，最新のウェアレベリング機構および，ガーベジコレクション機構においては，同一チップ内においてのみデータの移動を行うという特徴がある [1], [5], [6]．つまり，チップレベルにおいては，ウェアレベリングもガーベジコレクションもデータの位置に全く影響を与えない事を意味する．

2.2 I/O Scheduler

I/O スケジューラとは，図 2 のように，ユーザ空間のプロセスから発行された I/O リクエストが，カーネル空間のリクエストキューに挿入され（インサート），ブロックデバイスに発行（ディスパッチ）されるまでの間に，I/O リクエストのソートやマージなどの操作を行うカーネル空間で動作するプログラムである．HDD が対象の場合には，マージやソートによって，シーク時間，シーク距離を低減し，結果としてスループットの向上，レイテンシの低減，プロセス毎の公平性の向上などといった役割を担っている．しかし SSD において性能を改善させるためには，HDD の様にシーク時間の低減という性能改善の為のアプローチは行えず，別のアプローチが必要となる．

2.3 I/O リクエスト種別

まず一般的に I/O スケジューラにおいては，Read リクエストと Write リクエストという大別が行われ，この 2 種類のリクエストに対して異なるポリシーが適用される．

異なるポリシーを適用する理由は，通常，書き込みが遅延することによって，プロセスは休止しないが，読み込みが遅延すると殆どの場合プロセスが休止してしまう．つまり，Read と Write 双方向の処理が対等では無いからである．

また更に，同期 I/O リクエスト，非同期 I/O リクエスト

という大別も存在する [7]．同期 I/O においては，対象のファイルディスクリプタが準備完了状態では無い場合に，システムコールの返答待ち状態（ブロッキング状態）に陥る．非同期 I/O においては，処理が完了するまでバックグラウンドで待機し，終了したタイミングでシグナルを返したりコールバックが行われる．したがって，通知があるまでアプリケーションは別の処理を行えるといった違いがある．

2.4 現行のスケジューラと問題点

現在 Linux Kernel v4.x 系においては I/O スケジューラは以下の３つが用意されている．

- NOOP:カーネルから発行されたリクエストを単純な I/O リクエストのマージを行う以外には，そのままの順番で発行するスケジューラ．
- Deadline:それぞれの I/O リクエストに対してデッドラインを設定し，デッドラインを過ぎているリクエストがない場合には，出来る限りリクエストのソートを行う．デッドラインを過ぎているリクエストが存在する場合には再優先でデッドラインを過ぎているリクエストをディスパッチしようとする．上記のポリシーの特性から，一定のレイテンシを保つことができるという特性がある．
- CFQ:Linux 標準のスケジューラであり，プロセス毎の公平性と性能の両立を目指したスケジューラ．同期 I/O リクエストに対しては，それぞれのプロセス毎にキューをもち，それぞれに割り振られた優先度に応じてディスパッチするタイムスライスを与えて，タイムスライス内でディスパッチさせることにより，公平性を達成する．また，非同期 I/O リクエストに対しては，システム全体で同じキューを保持し，ディスパッチを行う．以上のように同期 I/O リクエストと非同期 I/O リクエストを分けて扱う事によって全体の性能を向上させるスケジューラ．また，最近 SSD 向けのパッチが当てられ [8]，SSD が接続されていると検知された場合には，タイムスライスではなく，IOPS(I/O per second) をディスパッチ単位とするような変更が行われている．これは，SSD を使用した環境では，時間あたりに多数の I/O リクエストがディスパッチされるため，タイムスライスの計測が難しく，適切なスケジューリングが行われないためである．この変更によって，SSD において性能が大幅に向上した．

しかし上記の Deadline スケジューラ及び，CFQ スケジューラは HDD ドライブのシーク時間，及びシーク距離をなるべく少なくするためのポリシーしか実装されておらず，全く SSD の性能を引き出すために考えられたスケジューラではない．そればかりか，CFQ においては，リクエストのソートやマージにかかる計算時間のために，Noop，Deadline よりもレイテンシが大きくなるばかりか，スループットまでも NOOP スケジューラに及ばないというのが現状である [9]．

また，一番計算量の小さい NOOP スケジューラにおいても，同一チャンネル内に存在するリクエストを同時にディスパッチしようと試みる事が多々あるので， SSD 内部の並列性を抽出できておらず，レイテンシの増大や，スループットの減少といった問題を抱えている．

3. 関連研究

3.1 SSD 内部の並列性の抽出

I/O スケジューラに限らず，SSD 内部の並列性を抽出しようとする試みは様々に行われてきた．例えば SSD 内のアドレス変換などを行う File Transfer Layer 機構 (FTL) の修正によって，SSD 内部の並列性をさらに抽出しようと試みる研究や [10]，新たに Write Buffer を実装し，ライトアンプリフィケーションを低減する目的の研究 [11] が行われてきた．スケジューラに関しては，ホスト側ではなく SSD 側に PAQ(Physical Address Queueing) と呼ばれるスケジューラを実装し，同じチャンネルを使用するリクエストを離してディスパッチすることにより，チャンネルレベルでの並列性を抽出する研究などがある [12]．しかし，この研究はホスト側ではなく SSD 側に実装するため，ハードウェアに手を加える・SSD の限られた計算資源しか使えないといったデメリットが存在する．

3.2 SSD のための I/O スケジューラ

先に述べた SSD の性質を利用し，SSD のスループット，レイテンシといった性能を向上させる目的のスケジューラや，SSD の寿命を延ばすためのスケジューラが提案されてきた．この節においては，SSD のためのスケジューリングポリシーを搭載した I/O スケジューラと，それらに改善の余地がある点について見ていく．

IRBW スケジューラ [13] や STB スケジューラ [14] は read と write の干渉を減らすことによる性能向上を見込んだスケジューラである．具体的には Write リクエストをできるだけまとめ，Read リクエストを先にディスパッチすることによって，遅い Write リクエストによって，Read 性能の低下が引き起こされる事を防いだスケジューラである．

また，同年に提案された block-preferencial スケジューラ [15] においては，ブロックの局所性を用いる事により，Write リクエストの性能向上を狙ったポリシーが搭載されている．このポリシーにおいては，同じブロックに属するデータが同時にディスパッチされるので，ガーベジコレクションの性能を上げる事ができる．

また，Android カーネルなどに搭載されている SIO スケジューラや SIOPLUS スケジューラ [16] は，Deadline スケジューラと NOOP スケジューラの中間に位置するようなスケジューラであり，計算の単純化によるレイテンシの低減と，デッドライン機構が搭載されたスケジューラである．また，同期的 I/O リクエストと，非同期 I/O リクエストに対して異なるディスパッチポリシーが搭載されており，同期リクエストによる他プロセスのブロッキングが抑えられている．それだけではなく，Read と Write の優先度の違いによって生じる一方向ばかりのリクエストのみがディスパッチされる状態（これを starvation という）を防ぐ機構も搭載されている．

しかしここまでのホスト側に実装されたどのスケジューラにおいても，I/O のコンフリクトを防ぐ機構は搭載されておらず，その部分においては SSD コントローラーの能力に任すといった方針になっている．

そこで本稿では上記の関連研究からの知見を元に，Write リクエストだけ，Read リクエストだけと言った特定環境

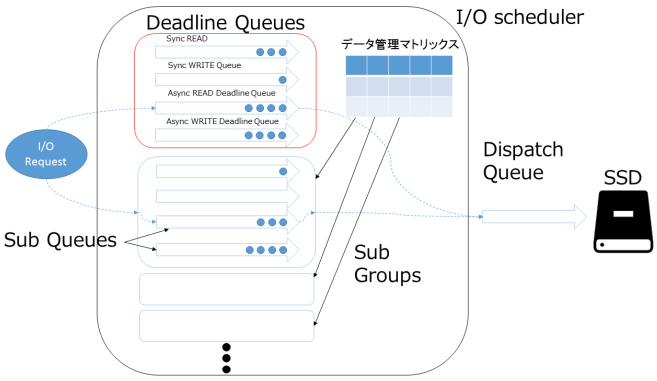

図 3 AC スケジューラーの概念図

下で性能の向上を目指すスケジューラではなく，Read と Write が混在した実際の環境に近いリクエストパターンで性能を出すことを目指す．

その要件として，以下の様な要件を設定する．

(1) 同期 I/O，非同期 I/O に対して，異なる優先度を割り当てる事が可能な事
(2) SSD 内部の並列性を抽出し，性能を向上させるために，I/O のコンフリクトを抑える事
(3) 優先度が高いリクエストのみのディスパッチによる starvation を防ぐ事
(4) ホスト側に実装することにより，豊富な計算資源を利用し，性能の向上を目指す事
(5) read と write の干渉を抑える事
(6) 最大レイテンシを抑えるために，デッドライン保障機構を搭載する事

以上の要件を達成した I/O スケジューラを提案する．

4. 提案手法

本稿で提案する Alleviate Conflict I/O スケジューラ (AC スケジューラ) の全体の概念図を図 3 に示す．AC スケジューラは，マトリックスと呼ばれる直近にディスパッチされた I/O 情報を管理するデータ構造を用いて，I/O のコンフリクトが少なくなるようにディスパッチするスケジューラである．詳細なマトリックスによるデータ管理の方法や，ディスパッチポリシーにおいては次節以降で説明する．AC スケジューラは図 3 で示されるようなデータ管理マトリックスをリクエストのインサート時とディスパッチ時に更新し，そのマトリックスの情報を用いて，どのサブグループに属するリクエストをディスパッチするのが適切なのかを随時判断し，ディスパッチしていく．また一定数ディスパッチされると，優先度を途中で切り替えることにより，優先度が高いリクエストのみがディスパッチされることを防ぐ機構や，レイテンシを保障するために，全てのリクエストを時系列で管理する機構も備えている．

4.1 データ管理方法

AC スケジューラにおいては I/O コンフリクトを抑え，性能を向上させるために，SSD の論理ブロックを SSD のチャンネル数分に分けて管理する．

具体的なデータ管理方法としては，図 3 の様に，AC スケジューラは，担当するリクエストを等間隔に論理アドレス毎に区切られたサブグループ内それぞれに同期 Write，非同期 Write，同期 Read，非同期 Read の 4 つのサブキューを持つ．またデッドライン管理のためのデッドラインキューを同期 Write，非同期 Write，同期 Read，非同期 Read の 4 つのリクエスト種別毎に保持する．

またそれぞれのサブグループ内のサブキューに存在する I/O リクエストの数と，それぞれのサブグループの状態を管理する 2 つのマトリックスを表 3 及び，表 2 のように．Write と Read それぞれのリクエスト種別について合計 2 つ保持している（それぞれを read マトリックス．write マトリックスと呼ぶ)．

マトリックスによって管理するそれぞれのサブグループの状態には，

- ZERO:該当サブグループに I/O リクエストが存在しないことを示す．
- PEND(pending):該当サブグループに I/O リクエストが存在するが，処理されていないことを示す．
- PROC(proceeding):ディスパッチ対象として前回採択されたサブグループであることを示す．Write リクエストを管理するマトリックスにおいてのみ使用する．ディスパッチ対象としての優先度が高いことを示す．
- END:ディスパッチ対象として前回採択されたサブグループであることを示す．Read リクエストを管理するマトリックスにおいてのみ使用する．ディスパッチ対象としての優先度が低いことを示す．

の以上 4 状態が存在する．

4.2 インサートポリシー

図 3 で示すように I/O リクエストを 2 種類のキューに挿入する．インサートポリシーの擬似コードを Algorithm1 に示す．

Algorithm 1 Insert Algorithm

```
1: Calculate Proper Sub-Group Index
2: DeadlineQueue ← Request
3: SubGroupQueue ← Request
4: RequestCounter ← RequestCounter + 1
5: FLAG ← PEND
```

1 種類目のキューは全体の I/O リクエストを管理するデッドラインキューに挿入する．デッドラインキューは同期 Write，非同期 Write，同期 Read，非同期 Read それぞれに対して一つずつ，計 4 つ存在する．リクエストをインサートする際に，カーネルが起動してからの tick 単位をカウントしているカウンタ (jiffies) を使用し，それぞれの I/O リクエストにデッドラインを設定する．(要件 5.3) このデッドラインは，同期 Write，非同期 Write，同期 Read，非同期 Read それぞれの種類のリクエストに対して表 1 の様に異なるデッドライン値を設定している．(HZ はタイマー割り込みの間隔を示す変数であり，Linux に於いては 10ms を示す．) (要件 5.3)

2 種類目のキューはそれぞれのチャンネルに相当する部分を管理するサブグループ内のリクエスト種別毎のサブキューである．挿入する際には，I/O リクエストが開始される論理アドレスから，どのサブグループに挿入されるかを計算する．

実際の計算式は，該当サブキューのインデックス，リ

表 1 Deadline 値

リクエスト種別	Deadline
同期 Write	2 * HZ
非同期 Write	16 * HZ
同期 Read	HZ / 2
非同期 Read	4 * HZ

クエスト開始位置を StartSector，SSD のキャパシティを Capacity，チャンネル数を Channel とすると，

$$SectorSize = Capacity/Channel$$
$$SubGroupIndex = StartSector/SectorSize$$

とした時の値 SubGroupIndex を用いている．

また，I/O リクエストをスケジューラに挿入する際には，データを管理するマトリックスのアップデートを行う．該当するサブグループのリクエスト数の値をインクリメントし，Read リクエスト，Write リクエストの種別に関わらず PEND フラグを設定する．

4.3 ディスパッチポリシー

I/O スケジューラの要点となる，ディスパッチポリシーについて説明する．リクエストをディスパッチする際には，マトリックスのフラグを参照しながらディスパッチ対象を選択する．ディスパッチポリシーの擬似コードを Algorithm2，Algorithm4，Algorithm3 に示す．以下，具体的なディスパッチポリシーについて説明していく．

Algorithm 2 Dispatch Algorithm

```
1: if batch_limit ≤ number of dispatched request continuously then
2:     search expired request in order of request property
3:     if exist expired request then
4:         dispatch expired request
5:         exit
6:     end if
7: end if
8: if write starvation may occur then
9:     Dispatch SYNC WRITE
10:    Dispatch ASYNC WRITE
11: else
12:    Dispatch SYNC READ
13:    Dsiaptch ASYNC READ
14: end if
```

4.3.1 デッドライン機構

デッドライン機構の擬似コードを Algorithm2 に示す．一定数のリクエストがディスパッチされるたびに，４つの Deadline キューのそれぞれの先頭のリクエストが，デッドラインを過ぎていないかをチェックする．(要件 5.3) これは低レイテンシを保障するためである．

チェックする順に関しては，一定数以上連続して Read をディスパッチしていない (starvation が発生していないと想定される) 場合には，同期 Read，非同期 Read，同期 Write，非同期 Write の順でデッドラインを過ぎていないかを調べる．

また，一定数以上連続して Read をディスパッチしていた場合 (starvation が発生している可能性がある場合) には，同期 Write，非同期 Write，同期 Read，非同期 Read の順でデッドラインを過ぎていないかを調べる．(要件 5.3)

そして，デッドラインを過ぎているリクエストが存在した場合には，該当のリクエストをディスパッチ対象に選択する．

このデッドラインキューの調査順については，同期リクエストは，他プロセスをブロックする要因になり，ディスパッチが遅れると，全体のパフォーマンスの低下原因になるために，先にディスパッチしようと試みている．

また, このスケジューラは Read Over Write なスケジューラ，つまり Read の優先度が高いスケジューラであり，これは，Write リクエストはなるべくまとめてディスパッチしたほうが，ページ単位で上書きできず，ブロック単位毎にしか上書きできない SSD において，性能を向上させる事ができるという研究結果に基づいたものである [13], [14]. (要件 5.3)

デッドラインを超えているリクエストが存在しない場合には，デッドライン機構によってディスパッチ対象の I/O リクエストは採択せず，マトリックスを使用してのリクエスト選択段階に入る．

4.3.2 マトリックスによるリクエスト選択

次にマトリックスを用いたディスパッチ対象の選択方法について述べる．ディスパッチ対象の選択方法についての擬似コードを Algorithm3 及び，Algorithm4 に示した．デッドラインを超えているリクエストが存在しない場合には，SSD 内部の並列性を抽出し，全体のスループットの向上及び，レイテンシの低減目的でチャンネルレベルの I/O コンフリクトが発生しないようにディスパッチする．(要件 5.3)

まずディスパッチするリクエストの種類に関して，starvation を防ぐために，一定数以上連続して Read をディスパッチしていた場合には，Write を，一定数以上連続してディスパッチしていなかった場合には，Read をリクエスト方向として選択する．そしてディスパッチ方向の同期リクエスト，非同期リクエストの順でディスパッチ対象とするリクエストを選択する．

そしてマトリックスから，ディスパッチ対象のサブグループを選択する段階に入る．

ディスパッチ対象として Read が選択されていた場合，チャンネルごとの I/O コンフリクトは，短い時間内に同一チャンネルに属するリクエストが SSD に発行された場合に発生するので，それを回避するために同一チャンネルに属するリクエストを連続して選択せず，リクエストが存在するチャンネルごとに，分散させようと試みる．また Read と Write は同時にディスパッチすると資源の競合が発生し，性能が低下するので，直近に Write リクエストが発行されたサブキューを避けるようにディスパッチする．具体的には，Read マトリックスにおいて PEND フラグが立っており，かつ，Write マトリックスについて PROC フラグが立っていないサブキューをディスパッチ対象として選択する．

ディスパッチ対象として Write が選択された場合には，Write マトリックスにおいて PROC フラグが立っているサブキューを選択する．これは，なるべくライトアンプリフィケーションを低減し，Write のコストを抑えるために，

同一チャンネルに属する Write をまとめて発行しようと試みていることを意味する．

Algorithm 3 Dispatch SYNC WRITE or ASYNC WRITE Algorithm

```
1: if WRITE-SYNC(ASYNC)-Queue is empty then
2:     return
3: end if
4: for all i such that 0 ≤ i ≤ MaxSub − GroupIndex do
5:     if i'th Sub-Group Flag is PROC then
6:         Dispatch i'th Sub-Group head request
7:         if Sub-Groups 's request num is 0 then
8:             Sub − Group'sFLAG ← ZERO
9:         else
10:            Sub − Group'sFLAG ← PROC
11:        end if
12:    end if
13: end for
```

4.3.3 ディスパッチ時のマトリックスのアップデート

ディスパッチ対象として選択されたリクエスト種別に関わらず，ディスパッチされたサブグループのリクエスト数をデクリメントする．ディスパッチ対象として，Read が選択されていた場合には，Read マトリックスのデクリメント後のリクエスト数が 0 以外の時は END フラグを，0 の時には ZERO フラグを設定する．そして，すべてのサブキューにおいて，PEND リクエストが存在しなくなった場合には，すべての END フラグを PEND フラグへと変更する．

次にディスパッチ対象として，Write が選択された場合には，デクリメント後のリクエスト数が 0 位外の時は Write マトリックスの該当サブグループに PROC フラグを，0 の時には ZERO フラグを設定する．

Algorithm 4 Dispatch SYNC READ or ASYNC READ Algorithm

```
1: if READ-SYNC(ASYNC)-Queue is empty then
2:     return
3: end if
4: for all i such that 0 ≤ i ≤ MaxSub − GroupIndex do
5:     if i'th READ Sub-Group Flag is PEND then
6:         if i'th Write Sub-Group Flag is PROC then
7:             search other group
8:         else
9:             Dispatch i'th Sub-Group head request
10:            if Sub-Groups 's request num is 0 then
11:                Sub − Group'sFLAG ← ZERO
12:            else
13:                Sub − Group'sFLAG ← END
14:            end if
15:        end if
16:    end if
17: end for
```

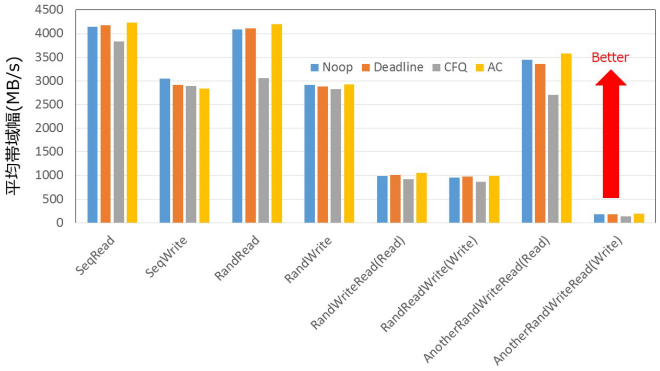

図 4 様々な I/O リクエストパターンにおいて各 I/O スケジューラにが達成する平均帯域幅

5. 評価及び議論

5.1 評価環境

表 4 の評価環境を準備し，評価を行った．提案した AC スケジューラは，Dynamic Kernel Module としてカーネルに組み込んだ（要件 4）．

評価には様々なパラメータを細かく設定できる fio ベンチマークを用いた [17]．

提案手法との比較対象として，NOOP スケジューラ，Deadline スケジューラ，CFQ スケジューラを選択した．

ベンチマークのための I/O リクエストパターンは以下の 6 つのパターンを用いた．

- シーケンシャル Read
- シーケンシャル Write
- ランダム Read
- ランダム Write
- ランダム Read&Write(50%:50%)
- ランダム Read&Write(90%:10%)(Web サーバーなどの挙動に近いアクセスパターン)

5.2 結果

図 4 に NOOP スケジューラ，Deadline スケジューラ，CFQ スケジューラ，提案した AC スケジューラの 4 つで計測した平均帯域幅の比較を示した．また図 5 に NOOP スケジューラ，Deadline スケジューラ，CFQ スケジューラ，提案した AC スケジューラの 4 つで計測した平均レイテンシの比較を示した．

図から見て取れるように，シーケンシャル Write での実験結果においては，帯域幅，レイテンシともに従来のスケジューラほどの性能は出すことはできなかった事がわかる．しかしシーケンシャル Write 以外の様々なリクエストパターンにおいて，帯域幅，レイテンシともに性能の向上を達成することができた事がわかる．特に Web サーバのようなリクエストパターン (図 5 における AnotherRandWriteRead) においては Noop スケジューラからは帯域幅 4%の向上，レイテンシは 15%の低減，Deadline スケジューラからは帯域幅 7%の向上，レイテンシは 7%の低減，CFQ スケジューラからは帯域幅 34%の向上，レイテンシは 40%の低減を達成することができた．

表 2　Write リクエスト管理マトリックス

Index	sub queue 0	sub queue1	sub queue2	sub queue3	sub queue4	sub queue5	sub queue6	sub queue7
request num	0	1	2	3	4	5	6	7
flag	ZERO	PEND	PEND	PEND	PEND	PEND	PEND	PEND

表 3　Read リクエスト管理マトリックス

Index	sub queue 0	sub queue1	sub queue2	sub queue3	sub queue4	sub queue5	sub queue6	sub queue7
request num	0	1	2	3	4	5	6	7
flag	zero	END	PROC	PEND	END	PROC	END	END

表 4　評価環境

Kernel	4.1.0-040100-generic(uname-r)
CPU	i7-3770K CPU @ 3.50GHz
SSD Device Model	Crucial_CT256MX100SSD1
接続インタフェース	SATA3
セル種別	Multi Level Cell
Capacity	256.1GB
SSD logical sector size	512 byte
SSD physical sector size	4096 byte

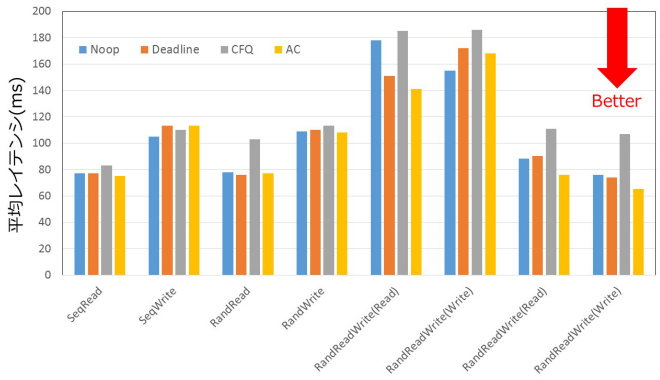

図 5　様々な I/O リクエストパターンにおいて各 I/O スケジューラが達成する平均レイテンシ

5.3　議論

この節では，我々の提案する AC スケジューラが設定要件を満たしているかを議論し，また AC スケジューラの改善点についても議論する．まず，設定要件に関しては，提案したスケジューラをホスト側に実装したことからすでに満たしている．

次に設定要件に関して議論する．AC スケジューラは，Noop スケジューラや Deadline スケジューラと比較して，スケジューラ自体の計算量が増加する．しかしそれにも関わらず，シーケンシャル Write 以外で性能が向上していることから，I/O コンフリクトの削減が達成されていると思われる．

次に設定要件と設定要件に関しては．ランダム Read が 90%，ランダム Write が 10%の I/O リクエストパターンにおける評価結果から議論する．Web サーバに近いこのリクエストパターンにおいては，starvation を防ぐ機構が組み込まれていない場合，優先度が高く設定されている Read リクエストが頻繁にインサートされ，稀にインサートされる Write リクエストが全くディスパッチされない starvation 状態が発生し，Write リクエストのレイテンシが著しく増大する．しかし図 5 から Write のレイテンシが抑えられていることから設定要件が達成されていることがわかる．また図 4 から，そのリクエストパターンにおいて帯域幅が向上していることもわかる．これは，Read と Write の競合を防いだ事によるものなので，設定要件を満たしていることが分かる．

デッドライン保障機構は，AC スケジューラからカーネルバッファに出力したデバッグ情報から適切に動作していることを確認した．また既存のスケジューラと比較して低レイテンシを達成できていることから，設定要件を満たしている．

このように AC スケジューラは 6 つ中 5 つの設定要件を満たしていることが確認できた．最後に設定要件が有効であることを示せる評価は本稿では行えていないので，これは今後の課題である．

上記で述べたように AC スケジューラはほぼすべてのアクセスパターンにおいて，既存の I/O スケジューラと比較して，帯域幅とレイテンシを改善したが，シーケンシャル Write の時は性能の向上が見られなかった．Write に関して，AC スケジューラは，帯域幅を向上させるために同じサブグループに属するリクエストをまとめるアルゴリズムが実装されている．しかしシーケンシャル Write においては，同じサブグループに属するリクエストが始めから連続して発行されるので，すでに AC スケジューラのディスパッチポリシーが適用された後のリクエストパターンに相当する．これはシーケンシャル Write においてはそのアルゴリズムは無駄であることを意味しており，そのアルゴリズムのための計算量が性能低下の要因となっている．そのため，シーケンシャル Write なアクセスパターンだと判断した場合には，AC スケジューラによるスケジューリングを一時的に OFF にする方法や，Write アクエスの Alignment を行うことによって性能の向上を図る研究で提案されている手法を write に関して適用してみることが改善案としてあげられる [18].

6.　まとめ

SSD 内部の並列性を効率よく抽出する目的で SSD のための I/O スケジューラ，AC スケジューラを提案した．AC スケジューラに於いては，SSD 内の I/O リクエストのコンフリクトを抑え，SSD の使用効率を向上させるために，マトリックスによって管理されたサブグループを用いてのディスパッチポリシーによる，I/O コンフリクトの低減を行った．

この提案手法により，様々なアクセスパターンにおいて，帯域幅の向上，レイテンシの低減といった性能の向上を達成する事ができた．Web サーバのようなアクセスパターンによる実験を行った時には，Linux カーネルで標準的に使用されている Noop スケジューラ，Deadline スケジューラ，CFQ スケジューラそれぞれと比較し，Noop ス

ケジューラからは帯域幅 4%の向上，レイテンシは 15%の低減，Deadline スケジューラからは帯域幅 7%の向上，レイテンシは 7%の低減，CFQ スケジューラからは帯域幅 34%の向上，レイテンシは 40%の低減を達成することができた．また，提案手法は，Dynamic Kernel Module として実装されているので，様々なシステムにおいて，導入が容易である．

参考文献

[1] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M. S. and Panigrahy, R.: Design Tradeoffs for SSD Performance., *USENIX Annual Technical Conference*, pp. 57–70 (2008).

[2] Kim, J., Seo, S., Jung, D., Kim, J.-S. and Huh, J.: Parameter-Aware I/O Management for Solid State Disks (SSDs), *IEEE Transactions on Computers*, Vol. 61, No. 5, pp. 636–649 (online), DOI: 10.1109/TC.2011.76 (2012).

[3] Chen, F., Lee, R. and Zhang, X.: Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing, *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, IEEE, pp. 266–277 (2011).

[4] Chen, F., Koufaty, D. A. and Zhang, X.: Understanding intrinsic characteristics and system implications of flash memory based solid state drives, *ACM SIGMETRICS Performance Evaluation Review*, Vol. 37, No. 1, ACM, pp. 181–192 (2009).

[5] Hu, Y., Jiang, H., Feng, D., Tian, L., Luo, H. and Ren, C.: Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance, *Computers, IEEE Transactions on*, Vol. 62, No. 6, pp. 1141–1155 (2013).

[6] Hu, Y., Jiang, H., Feng, D., Tian, L., Luo, H. and Zhang, S.: Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity, *Proceedings of the international conference on Supercomputing*, ACM, pp. 96–107 (2011).

[7] Boost application performance using asynchronous I/O: http://www.ibm.com/developerworks/library/l-async/.

[8] [PATCH] block: Make CFQ default to IOPS mode on SSDs: http://lkml.iu.edu/hypermail/linux/kernel/1506.0/04483.html.

[9] Han, L., Lin, Y. and Jia, W.: A new I/O scheduler designed for SSD through exploring performance characteristics, *Computing, Communications and IT Applications Conference (ComComAp), 2014 IEEE*, pp. 161–166 (online), DOI: 10.1109/ComComAp.2014.7017189 (2014).

[10] Shin, J.-Y., Xia, Z.-L., Xu, N.-Y., Gao, R., Cai, X.-F., Maeng, S. and Hsu, F.-H.: FTL design exploration in reconfigurable high-performance SSD for server applications, *Proceedings of the 23rd international conference on Supercomputing*, ACM, pp. 338–349 (2009).

[11] Chen, F., Koufaty, D. A. and Zhang, X.: Hystor: making the best use of solid state drives in high performance storage systems, *Proceedings of the international conference on Supercomputing*, ACM, pp. 22–32 (2011).

[12] Jung, M., Wilson III, E. H. and Kandemir, M.: Physically addressed queueing (PAQ): improving parallelism in solid state disks, *ACM SIGARCH Computer Architecture News*, Vol. 40, No. 3, IEEE Computer Society, pp. 404–415 (2012).

[13] Kim, J., Oh, Y., Kim, E., Choi, J., Lee, D. and Noh, S. H.: Disk schedulers for solid state drivers, *Proceedings of the seventh ACM international conference on Embedded software*, ACM, pp. 295–304 (2009).

[14] Kang, S., Park, H. and Yoo, C.: Performance enhancement of I/O scheduler for solid state devices, *Consumer Electronics (ICCE), 2011 IEEE International Conference on*, IEEE, pp. 31–32 (2011).

[15] DUNN, M. P.: A new I/O scheduler for solid state devices, PhD Thesis, Texas A&M University (2009).

[16] cyanogenmod: http://www.cyanogenmod.org/.

[17] fio - freecode: http://freecode.com/projects/fio.

[18] Wang, M. and Hu, Y.: An I/O Scheduler Based on Fine-grained Access Patterns to Improve SSD Performance and Lifespan, *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, New York, NY, USA, ACM, pp. 1511–1516 (online), DOI: 10.1145/2554850.2554971 (2014).