Tokyo Institute of Technology
Department of Computer Science

修士論文

SSD の並列性を引き出す I/O スケジューラに関する研究

指導教員: 吉瀬 謙二 准教授

平成 28 年 1 月

提出者

大学院情報理工学研究科 計算工学専攻

14M38043　奥村開里

概要

近年，Solid State Drive(SSD) が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の幅広い範囲で使われ始めている．SSD は Hard Disk Drive(HDD) と比べ，機械的な部品を持たない為に耐衝撃性が高いことや，低レイテンシ，I/O の並列処理による高い帯域幅を持つといった利点がある．一方で，記憶素子あたりの寿命が短いことや，容量あたりの価格が高いといった欠点がある．また，記憶素子の性能の劣化を抑える為の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従来の HDD には存在しない機構を備えている．この様な SSD は HDD の置き換えを目的として開発された為，HDD 用に書かれたカーネルのコードで動作させることができる．現在 Linux Kernel に組み込まれている Deadline スケジューラや，CFQ スケジューラなどの I/O スケジューラなどに関しても，SSD のこれらの特徴を考慮せず，HDD を想定したコードとなっている．

本論文では，研究が行われている SSD 向けの I/O スケジューラのポリシーに加え，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．提案する AC スケジューラを Linux Kernel のダイナミックカーネルモジュールとして実装し，SSD に対する様々な I/O リクエストパターンにおいて，キャッシュを用いた時，用いなかった時双方で，SSD の帯域幅とレイテンシを評価した．その結果，キャッシュ用いた時，用いなかった時，共に概ね性能の向上が見られた．しかし，キャッシュを用いた場合には，キャッシュを用いなかった場合と比較し，相対的に性能が劣化した．これは，キャッシュが AC スケジューラに適さないように実装されていることが原因だと思われる．一方で，キャッシュを用いなかった場合には，キャッシュを用いた場合と比べ，相対的に性能が向上した．特に，一般的な Web サーバで発行される Read リクエストと Write リクエストの比になっている I/O アクセスパターンにおいて，提案した AC 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，大幅な帯域幅の向上，レイテンシの低減共に達成した．

目次

図目次

表目次

List of Listings

第 1 章

序論

1.1 はじめに

1.1.1 背景

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．しかし SSD には，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ある一方，欠点も存在する．

記憶素子として使用されている NAND チップには，書き込み回数制限が存在する [1]．その問題を回避するために，SSD には寿命を延ばす目的のウェアレベリングという機構が搭載されている．また，削除操作をブロック単位でしか行えないという SSD の欠点から，ガーベジコレクションという SSD 特有の操作が存在する．その他，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．

上記のように HDD と SSD の間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 用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，Linux Kernel には，性能を犠牲にし，互換性を重視したコードが残っており，未だに SSD の性能を十分には引き出せずにいる．

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. SSD の性能を引き出すためには SSD の特徴を活かした OS 側のサポートも必要である．

このため本論文では，上記の特徴や，関連研究からのアイディアを活かし，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として組み込むことが可能であり，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．

本論文では, 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と，Read リクエストと Write リクエストの相互干渉による性能低下を防ぐ事によって性能の向上を見込むスケジューラである．この 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について評価する．

1.2　本論文の構成

本論文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．その後に続いて，謝辞，研究業績，付録を掲載している．付録には実験および実装で使用したエミュレーション環境を作成するにあたって，使用したツール，手順を掲載している．

第 2 章

背景

この章においては，提案手法において考慮した SSD の性質，カーネルにおいて一般的な I/O スケジューラが果たす役割，発行される 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．

SSD には，I/O を管理するフラッシュコントローラー，記憶素子の階層として，チャンネル，パッケージ，ダイ，プレーンなどが存在する．フラッシュコントローラーについては，2.1.1 節において述べる．以下，それぞれの記憶階層について述べる．

- チャンネル:リクエストを複数パッケージに伝達する役割を果たす．フラッシュコントローラーに複数接続されている (図 2.1).
- パッケージ:複数のダイを保持している．チャンネルに複数接続されている (図 2.1).
- ダイ:複数のプレーンを保持している．パッケージに複数接続されている (図 2.2).
- プレーン:実際にデータを保持する．データは論理的に削除の最小単位であるブロックと，読み込みの最小単位であるページによって管理される．ブロックは複数ページで構成されている為，読み込み操作のほうが粒度が小さい (図 2.2).

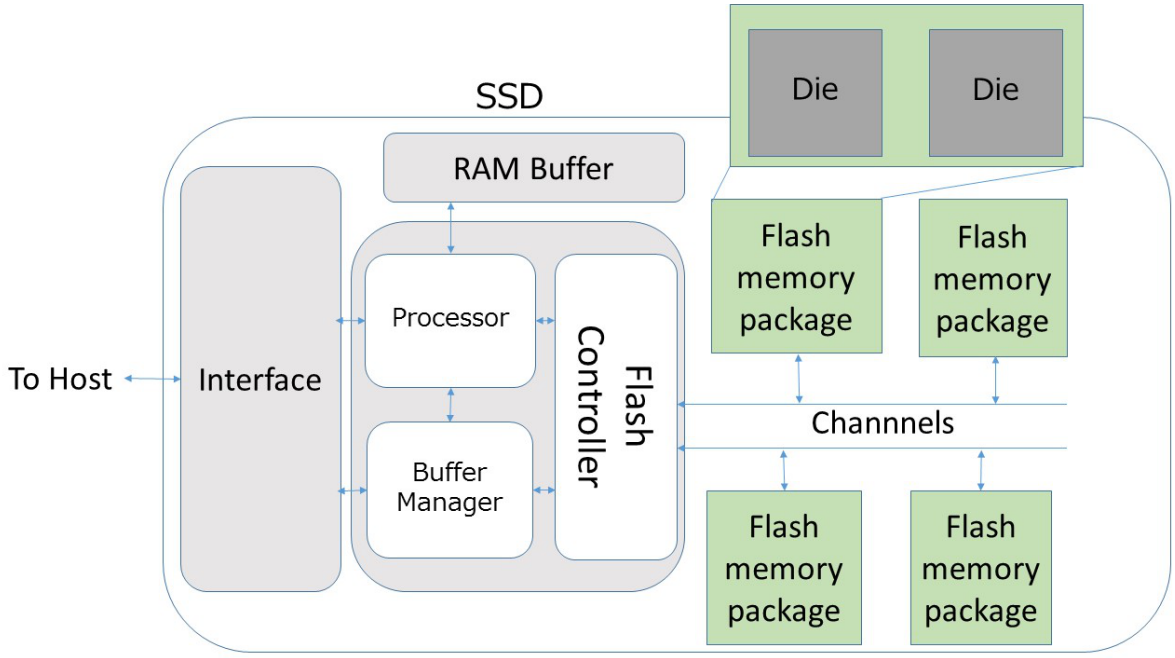

図 2.1　SSD の構造図

2.1.1　フラッシュコントローラー

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，ガーベジコレクションという独自の操作がフラッシュコントローラによって実現されている．SSD 内部の並列性，それぞれについて詳細に述べる．

2.1.2　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 2.1）.
- 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

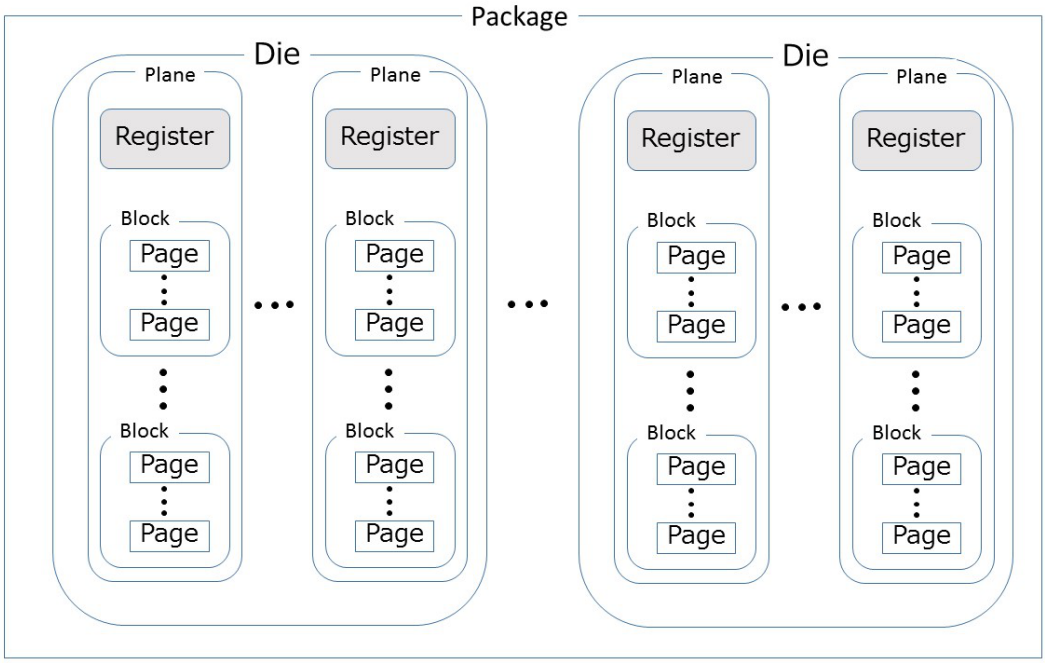

図 2.2　パッケージの構造図

ス可能なこと（図 2.1）.

- 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 2.1）.
- 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．(図 2.2)

これらの SSD 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 I/O リクエストと言える．

2.1.3　SSD の I/O 特性

SSD において無視することができない特徴として，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，ブロック単位でしか処理できないといった事が原因で，ライトアンプリフィケーションという状態 (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

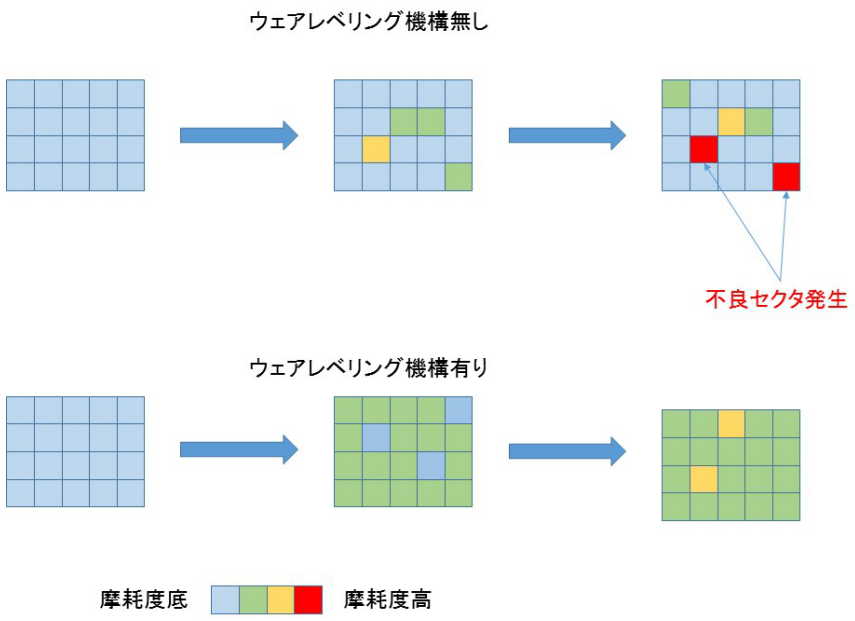

図 2.3　ウェアレベリング図

くない状態) が発生しうる事などの特徴がある [2, 1].

また，Read と Write 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 (Read リクエストと Write リクエストの相互干渉) が起こるので，できるだけ Read と Write を別々に行う事が好ましい．例えば Read のみを先に処理する readahead ストラテジーを，I/O を大量に吐くアプリケーションに取り入れると，性能が向上する [3, 4].

2.1.3.1　SSD におけるウェアレベリング

SSD においては，記憶素子として NAND チップが使用されているが，この NAND チップには書き込み回数制限が存在する．

ウェアレベリングという SSD の機構は，この書き込み回数制限が存在する記憶素にに対して，書き変えが一つの素子に集中することを防ぎ，各記憶素子にできるだけ均等に分散して書き換えが起こるように制御する機構のことである [5]. ウェアレベリング図を図 2.3 に示した. 書き換えをそれぞれの素子に分散させることにより，SSD 全体としての書き換え限度回数を向上させることができる．

2.1.3.2 SSD におけるガーベジコレクション

SSD においては，HDD などの記憶媒体との最も大きな違いとして，読み書きはページ単位で行えるが，書換，消去に関してはブロック単位でしか操作を行えないという特徴がある．この SSD の特徴に起因する機構としてガーベジコレクションが存在する．

このガーベジコレクションの手順を図 2.4, 図 2.5 図 2.6 に示した．図 2.4, 図 2.5 図 2.6 のように，SSD においてデータの存在するページに対して書き込み（上書き）を行う場合，下記の様な手順でガーベジコレクションと呼ばれる操作を行う必要がある [6].

1. 更新対象のページに使用不可能フラグを立て，キャッシュ内にて，更新された後のページを作成する．(図 2.4)
2. 空きブロックに対して，更新された後のページ，もともとのページ情報などをコピーする．(図 2.5)
3. アドレステーブルを新規に書き込まれたブロックを参照するように更新し，元のブロックを削除する．(図 2.6)

このガーベジコレクションを効率良く行うと，SSD の性能は向上する．

2.1.4 ライトアンプリフィケーション

上記２つのウェアレベリング及び，ガーベジコレクションという機構において，問題となるのが，本来の書き込みを行いたいデータ量よりも多くのデータ量を SSD 側では処理しなくてはならないといったことである．この本来書き込みに必要なデータ量よりも書き込みデータ量が増えてしまう現象をライトアンプリフィケーションという [7]．ライトアンプリフィケーションは SSD においては書き込み回数制限が存在するので，性能劣化を早めたり，単位時間あたりのデータ処理量の増大による帯域幅の低下などを引き起こす．よって，ライトアンプリフィケーションの低減は結果的に SSD の性能向上を見込む事ができる．

2.1.5 SSD 内部のデータ移動

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, 8, 9]．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．つまり SSD 内においては論理アドレスと物理アドレスの変換が行われるが，チップレベルにおいては，データの格納される場所は同一である．

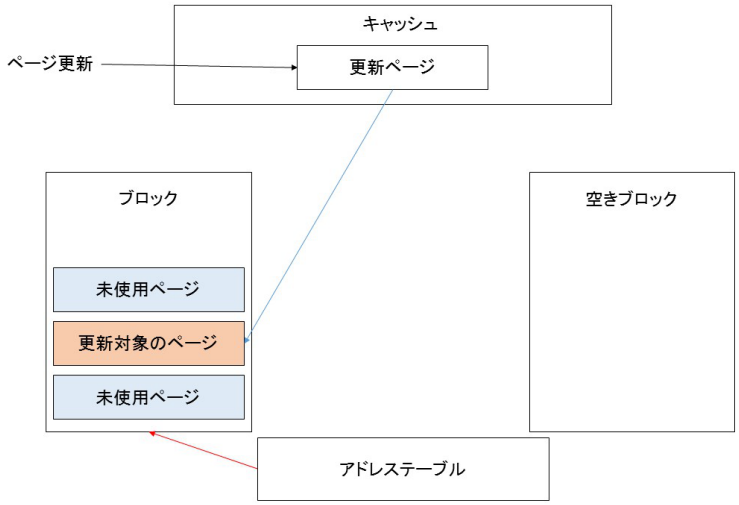

図 2.4　ガーベジコレクション手順 1

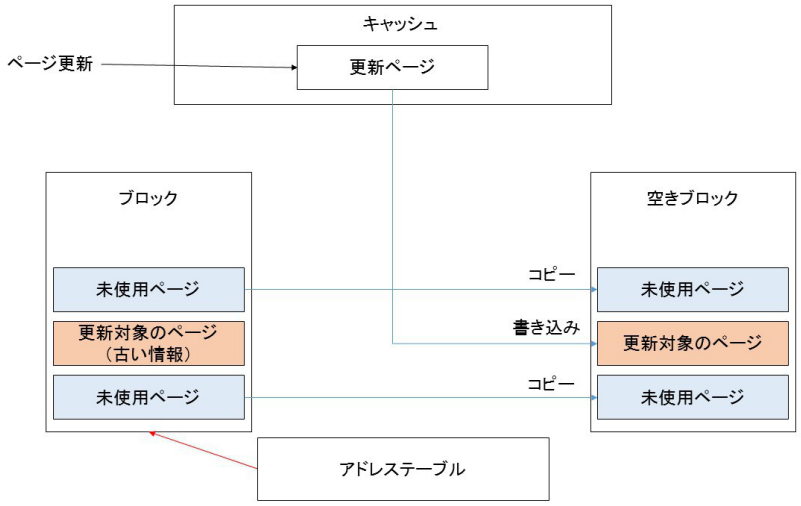

図 2.5　ガーベジコレクション手順 2

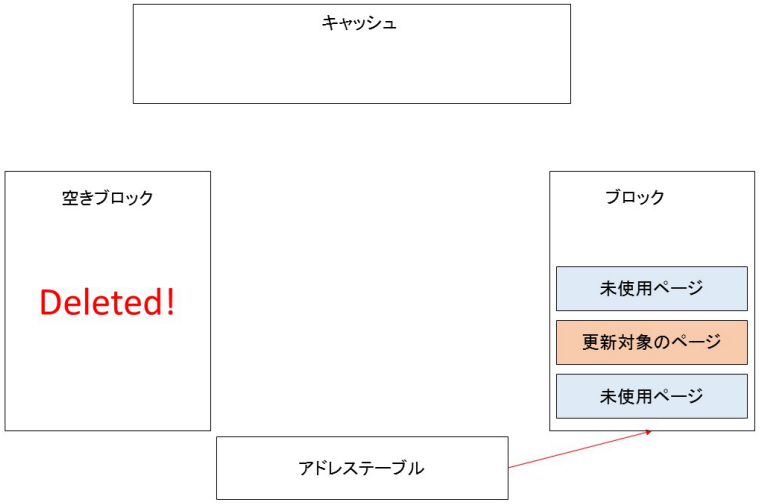

図 2.6　ガーベジコレクション手順 3

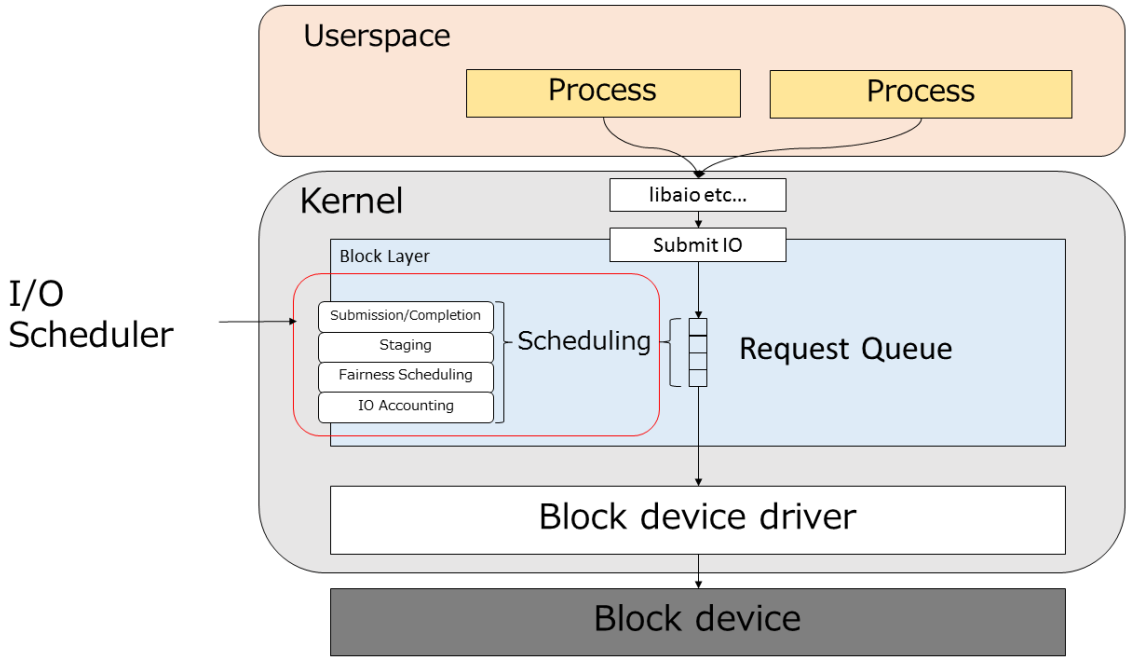

図 2.7　I/O スケジューラの概念図

2.2　I/O Scheduler

I/O スケジューラとは，図 2.7 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 とは異なりそもそもシーク時間が存在しないため，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．

2.2.1　I/O リクエスト

まず I/O スケジューラにおいて扱う I/O リクエストについて説明する．I/O リクエストは図 2.8 の様に，ユーザー空間のプロセスから発行され，カーネル空間の I/O エンジンを経由し，デバイスドライバに渡され，実際にデバイス（SSD) に発行される．

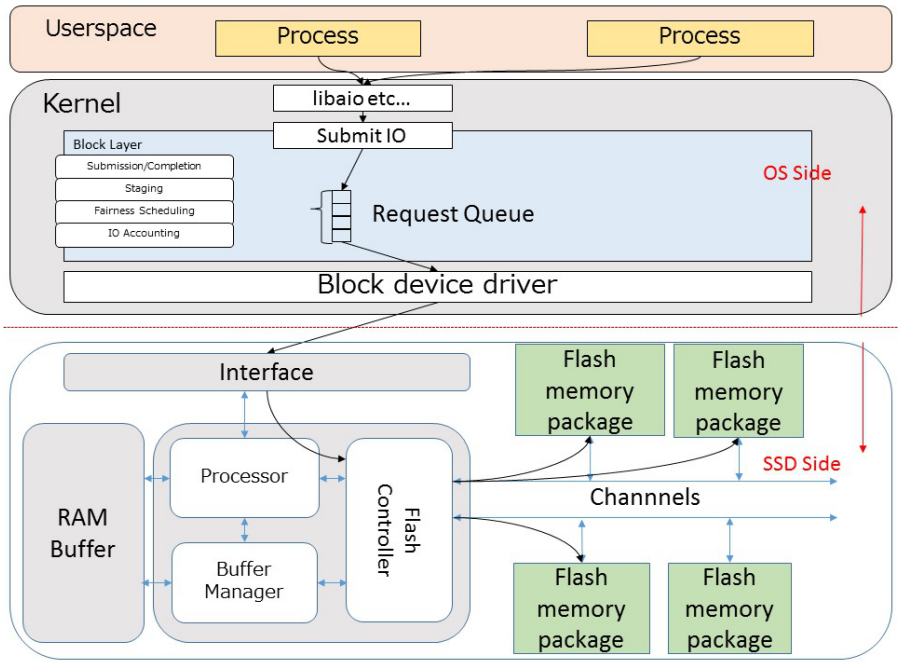

図 2.8　I/O リクエスト全体図

SSD に発行された後，フラッシュコントローラーによってリクエストが処理され，該当物理アドレスに書き込み，読み込み，削除，上書きなどの操作が行われる．

2.2.1.1　Read と Write

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それぞれ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 SSD においては，Read と比較して Write は非常に遅いために，同一のポリシーは適さない．

2.2.1.2　同期と非同期

また更に Read と Write という大別以外に，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 [10]．同期 I/O と非同期 I/O についての簡単な図を図 2.9 に示した．同期 I/O の場合は，

1. プロセスから I/O に関するシステムコールが発効．カーネルに処理が譲渡．プロセ

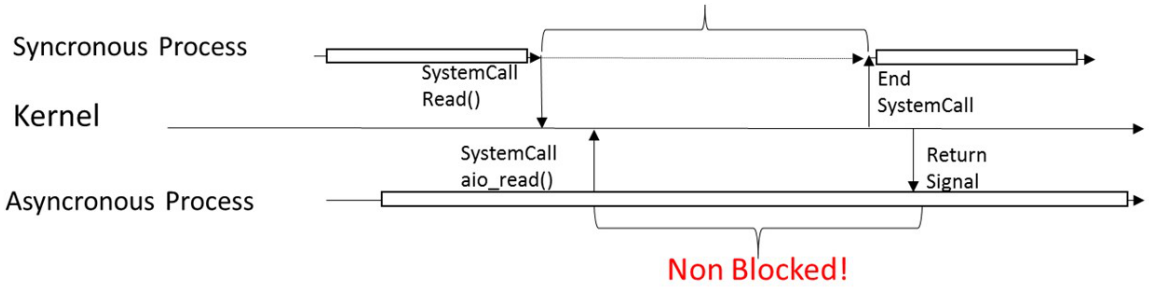

図 2.9　同期リクエスト，非同期リクエスト図

スは中断．（ブロッキング状態）

2. カーネルがシステムコールを処理する．
3. カーネルがシステムコールの処理を完了．
4. プロセスに処理が返却され，システムコールの結果を用いて処理を再開する．

といった処理が必要であるが，非同期 I/O の場合には，

1. プロセスから I/O に関するシステムコールが発行される．
2. プロセスは，カーネルがシステムコール処理中に並行して，本来の処理を継続．
3. システムコールが終了すると，終了した旨のシグナルが返却されるので，そのシグナルに対応したコールバック処理を行うことによって処理を完了する．

といったように，非同期 I/O においてはプロセスがブロックされることがないので，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．

後述する CFQ スケジューラにおいては，同期 I/O と非同期 I/O で異なるポリシーを適用することにより，プロセスのブロッキングをできるだけ防ごうと試みる機構が搭載されている．

2.2.2　現行の I/O 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つ．それぞれのプロセスに割り振られた優先度に応じてディスパッチに使用するタイムスライスを与え，タイムスライス制限時間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 [11]，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 において性能の低下が抑えられている．

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方針のポリシーしか実装されておらず，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 においては上記の様に SSD 向けの修正がなされていてもなお，アルゴリズムの複雑さによって，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 [12].

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たり，Read と Write の干渉を引き起こしたりといった事が多々あるので， 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．

第 3 章

関連研究

3.1 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 スケジューラを述べる．

3.1.1 IRBW スケジューラ

IRBW スケジューラ [13] 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．

3.1.2 STB スケジューラ

STB スケジューラ [14] は前述の IRBW スケジューラ同様 read リクエストと write リクエスト間の干渉を減らすことによって性能の向上を目指すスケジューラである．

前述の IRBW スケジューラはリクエストを READ リクエストと WRITE リクエストの２種類に分けて，それぞれのリクエストに対して異なる処理を行っていた．

IRBW スケジューラとの差異として，STB スケジューラはそのポリシーに加えて，さらに同期リクエスト，非同期リクエストという区別を加え，同期 WRITE，同期 READ，非同期 READ，非同期 WRITE の４種類のリクエストに対し異なるポリシーを適用するといっ

た点や，それぞれのリクエストに対して，デッドラインを設定し，レイテンシを抑えるためのデッドライン機構が追加されたことなどがあげられる．

3.1.3 Block-preferencial スケジューラ

IRBW スケジューラと同年に提案された SSD 向けの I/O スケジューラとして，block-preferencial スケジューラ [15] がある．このスケジューラにおいては，同じブロックに属する可能性があるデータを扱うリクエストをなるべくまとめてディスパッチすることによって，Write のクロスペナルティ（部分的なデータのアップデートがガーベジコレクションの性能に悪影響を与えることとされている.）を抑えることによる，Write リクエストの性能向上及びガーベジコレクションの性能向上を見込んだポリシーが搭載されている．

3.1.4 SIO スケジューラ

SIO スケジューラ [16] はできるだけレイテンシを小さくすることによる性能向上を目指したスケジューラである．

SIO スケジューラは基本的には FIFO(First In First Out) でリクエストを処理するが，それぞれのリクエストにデッドラインを設定し，デッドラインを満了しているリクエストを優先的にディスパッチするポリシーを備えている．計算が単純な為に I/O と比べて CPU が非力なマシン（組み込み環境や，ポータブルなデバイスなど）に搭載される事が多く，計算量の削減により，レイテンシを抑えることができ，デッドラインにより最大レイテンシも抑えることができる．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．

またそれだけではなく，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．NOOP スケジューラと Deadline 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る．

3.1.5 FIOS

FIOS(flash I/O scheduler) は次に処理される I/O の予測を行うことによるプロセス毎の公平性の向上と，SSD の並列性を利用し，全体のスループットの向上の両方の達成を目指すスケジューラである．この公平性は，論文内では，一つずつアプリケーションを動かした時のスループット合計と，並列にアプリケーションを動かした時のスループット合計で論文内で定義されている．

3.2　その他の SSD 性能向上手法

ここまでは，OS の I/O スケジューラに関する研究を見たが，もちろん I/O スケジューラ以外にも SSD の性能を向上させるために改善の余地が残されており，様々なアプローチで研究がなされている．

3.2.1　FTL(File Translation Layer) Design

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 [17] が行われている．

この研究は，SSD 自体に手を加える必要があるため，ポータビリティの低さなどが問題としてあげられる．

3.2.2　PAQ Scheduler

PAQ(Physical Address Queueing) スケジューラ [18] は SSD 側に実装されたスケジューラである．物理アドレス変換後に，なるべく同じチャンネルを使用するリクエストを検知し，競合が起きやすいようなリクエストを離してディスパッチすることにより，性能向上を目指すスケジューラである．しかし上記の研究同様 SSD 自体に手を加える必要と，計算資源に乏しい SSD 側で実装されていることにより，計算量の増大がレイテンシに悪影響を与えているという問題がある．

3.2.3　ライトアンプリフィケーションの低減

SSD の性能を向上させる目的でライトアンプリフィケーションの低減を目指した研究も行われている．[19] この研究では新たに Write Buffer を実装し，実際に I/O リクエストが発効されるまでの間に使用することによりライトアンプリフィケーションの低減を行っている．

3.2.4　blk-mq

blk-mq[20] はストレージが秒間あたりに処理する IO 数の増大についていくために，Linux でのそもそもの I/O リクエスト処理を行う Block-Layer の設計を変えようという研究である．複数のハードウェアキューがサポートされ，複数の CPU コアに I/O が分散される様になった．複数の CPU コアを利用できるようになり，より良い ReadWrite 性能と，

レイテンシの削減が実現された．しかし，使用する場合には，カーネルの再コンパイルの必要があり，可搬性には問題がある．また，それぞれのハードウェアキューには，複数のソフトウェアキューが割り当てられているが，それぞれが FIFO なために，改善の余地があると考えられる．

3.3　まとめ

ここまでに紹介したホスト側に実装されたどの I/O スケジューラにおいても，Read と Write の競合，SSD 内部の並列性を抽出できないようなリクエスト処理パターンといった，本論では I/O のコンフリクトと呼ぶ SSD に向いていないリクエスト全般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．

そこで本論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 スケジューラを次章において提案する．

第 4 章

提案手法

4.1 提案手法概要

本論文で提案する 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 4.1 に示す.

AC スケジューラにおいては，I/O リクエストがディスパッチされた順番や，I/O リクエスト種別，I/O リクエストの数などを管理するために，4.1 で示されるようなデータ管理マトリックスと本論文で呼ぶデータ構造を用いる.

AC スケジューラは Read リクエストに関しては，このデータ管理マトリックスを用いて，I/O コンフリクトとよばれる SSD 内の資源の競合を抑え，SSD の並列性を出来るだけ抽出できるような順番で I/O リクエストをディスパッチする．また Write に関しては，Block-preferencial スケジューラ [15] で述べられている Block Crossing Penalty の低減を目指すため，同じブロックに属するリクエストをなるべく同時にディスパッチする機構を備えている.

また，Read または Write の一方向だけが一定数以上連続してディスパッチされた時には，Read と Write の優先度を途中で切り替えることにより，一方向のリクエストばかりがディスパッチされてしまう現象を防ぐ機構も備えている．この現象は starvation と呼ばれる．starvation は図 4.2 の様に，優先度のリクエストを吐き続けるアプリケーションが存在する場合に発生しやすい.

また，レイテンシの増大を防ぐために，starvation を防ぐ機構だけではなく，それぞれの I/O リクエスト一つ一つに対して，デッドライン（クロックを用いて設定している）をスケジューラ内で設定し，デッドラインを過ぎてしまっているリクエストを優先的にディスパッチするデッドライン機構も備えている.

さらには，Read と Write が同時に発行されていた場合には，Read と Write の性能を低

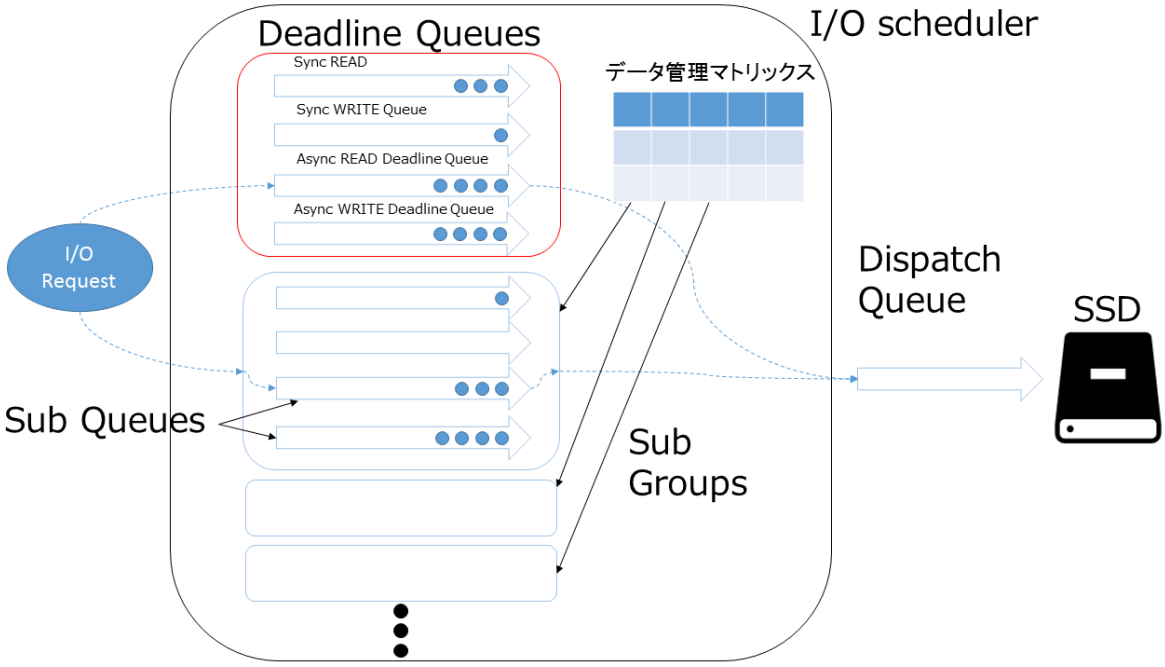

図 4.1　AC スケジューラーの概念図

めてしまうので (Read and Write interference)，できるだけ Read と Write を分けて発行するような仕組みを搭載している．

また，AC スケジューラにおいては，デッドラインの設定，リクエスト種別毎の優先度の変更などはユーザ空間から行うことができる．AC スケジューラは，sysfs[21] を用いて，モジュールのソースを変更することなく，詳細に設定することが可能である．sysfs は，デバイスやドライバの情報をユーザ空間にエクスポートする仮想ファイルシステムである．

I/O コンフリクトの低減による SSD の並列性の抽出方法，Read and Write Interference, Block Crossing Penalty，データ管理マトリックスのデータ構造，I/O リクエストをディスパッチする順番，I/O スケジューラへの I/O リクエストのインサートの仕方などの詳細は次節以降追って説明する．

4.2　I/O コンフリクト

まず AC スケジューラの根幹と関連する I/O コンフリクトについて説明する．

SSD 内において同じ資源の競合が起こり，結果として全体の性能低下を招く事を本論文では，I/O のコンフリクトと呼んでいる．

一例として，図 4.3 のように Channel1 と Channel2 を利用できている上記 (Read ABCD)

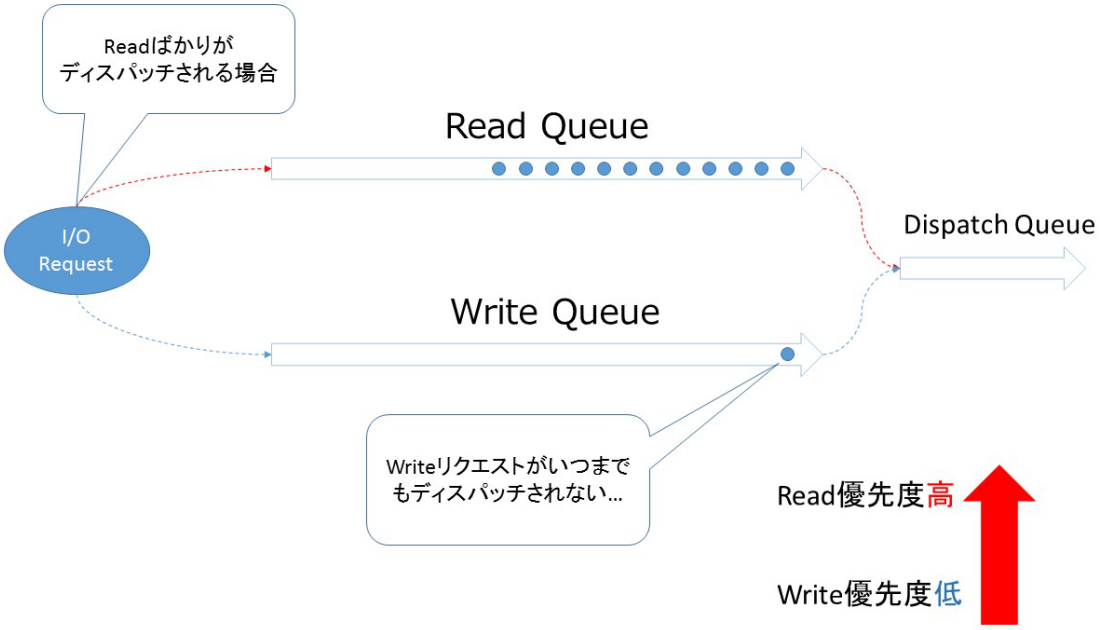

図 4.2　starvation 概略図

のリクエストと Channel1 しか利用できていない下記 (Read ABEF) のリクエストでは，上記のリクエストは SSD の並列性を効率よく利用できているので，下記のリクエストより性能が向上する．

よって，スケジューラからディスパッチされるリクエストが，なるべく同じチャンネルを利用しない順番，つまり，同じチャンネルを利用するリクエストは離してディスパッチするほうが良いというのが AC スケジューラの基本方針である．

4.3　Read と Write の相互作用

SSD において，Read リクエストと Write リクエストが同時に発行されると Read と Write 両方の性能が低下する [22]. SSD においては Read 操作は Write 操作よりも高速という非対称性がある．このような性質から，Write リクエストの処理を後回しにし，Read リクエストを先に処理すると，ワーストケースの場合のレイテンシが抑えられるという結果 [22] が出されている. AC スケジューラにおいては，基本的に Read over Write なポリシーを備えている．

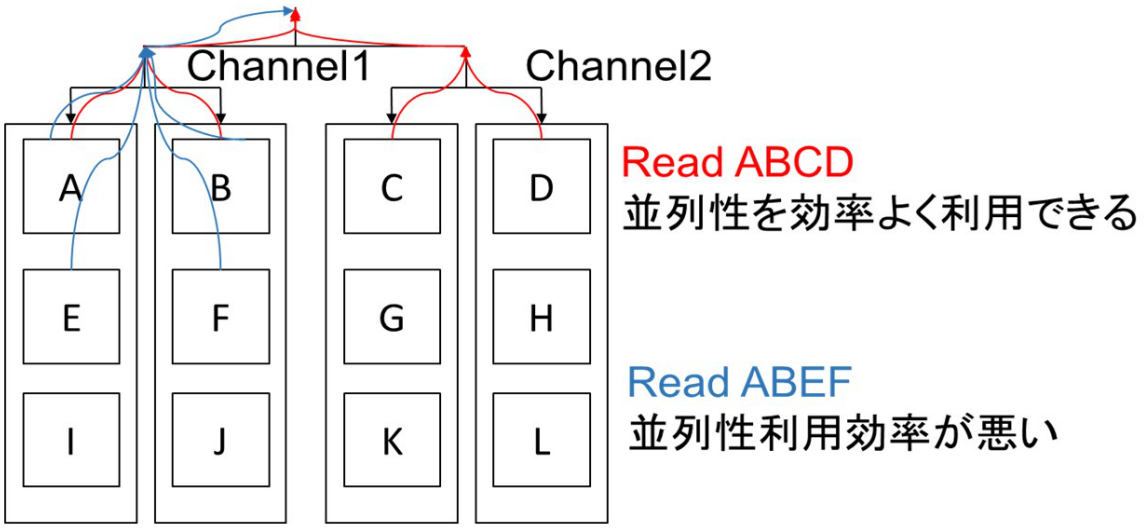

図 4.3　チャンネルレベル並列性

4.4　Block Crossing Penalty

SSD に発行される Write リクエストについて考える．背景において述べた様に，上書きを行う際には，ガーベジコレクションという操作が SSD 内にて行われる．このガーベジコレクションという操作は，Block 内全てに書き込む Write リクエストを処理する場合だと，前に書き込まれているブロックの情報が効率良く操作されるので，Write リクエスト全体としての効率は向上する．一方で，ブロックが部分的に変更されるリクエストの場合には，valid なデータと前に書き込まれていたデータの整合が発生するので，ガーベジコレクションにオーバーヘッドが発生してしまう．

Block-Preferencial スケジューラ [15] や FIOS スケジューラ [22] と同様に，AC スケジューラにおいても同じチャンネルに属するリクエストはなるべく同時にディスパッチし，その結果として，ブロックをまたがないリクエストになるように目指したポリシーが搭載されている．

4.5　データ管理方法

4.5.1　サブグループ及びサブキュー

本論文においては，提案する 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，SSD の論理ブロックを SSD のチャンネル数分に分割して管理する．

AC スケジューラは，I/O リクエストを希望の順番，つまり SSD の並列性をうまく抽出できるような順番でディスパッチするために，図 4.1 の様に，リクエストを SSD 内に存在するチャンネル数分のサブグループと呼ぶ単位に分けて管理する．

まず I/O リクエストは，操作対象となる論理アドレスから，該当するサブグループが割り出される．そして，そのサブグループ内には同期 Write，非同期 Write，同期 Read，非同期 Read の４つのサブキューが存在し，I/O リクエストの種別によってインサートされるキューが決定され，実際にインサートされる．また今回使用したキューの実装については，付録 6.4 に示した．

4.5.2　デッドラインキュー

次にデッドラインを管理するデッドラインキューについて述べる．全てのサブグループ内の全てのサブキューに存在するリクエストに対して，デッドラインを満了しているかをチェックする事によってデッドライン機構はもちろん実現可能である．しかし，できるだけ計算量を抑え，レイテンシを下げる必要のある I/O スケジューラにおいては，その計算は望むところではない．

その為に AC スケジューラにおいては，図 4.4 のようにデッドラインキューを保持している．デッドラインキューは該当するアドレスに関わらず，リクエストが時系列順にインサートされる．リクエスト種別ごとに優先度，つまりデッドラインが異なるので，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存在する．この様なデッドラインキューを準備することにより，それぞれのデッドラインキューの先頭がデッドラインに最も近いことは保証されているので４つのデッドラインキューの先頭についてのみデッドラインを満了しているかどうかを調べる事により，計算量を抑えたままデッドライン機構が実現できる．

4.5.3　データ管理マトリックス

AC スケジューラはキューだけではなく，本論文ではデータ管理マトリックスと呼ぶデータ構造を用いて，ディスパッチする順番を決定している．

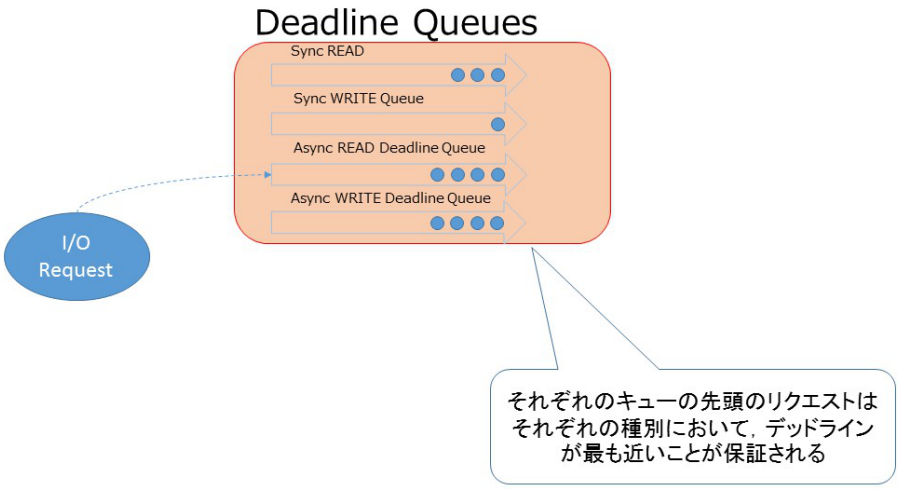

図 4.4　デッドラインキュー

表 4.1　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	PEND	END	PEND	PEND	PEND	PEND	PEND

表 4.2　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	PROC	PEND	PEND	PEND	PEND	PEND

AC スケジューラは，表 4.1 及び，表 4.2 のように Read リクエストを管理する Read データ管理マトリックスと，Write リクエストを管理する Write データ管理マトリックスの 2 つを使用する．

4.5.3.1　Read データ管理マトリックスとサブグループの状態

Read データ管理マトリックスはサブグループ内のサブキューに存在する Read リクエストの数と，Read リクエストに関するサブグループの状態を管理する．Read に関するそれぞれのサブグループの状態は下記の３つがある．

- ZERO:該当サブグループに Read 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 Read リクエストが存在するが，処理されていないことを示す．ディスパッチ対象としての優先度が高い．
- 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．ディスパッチ対象としての優先度が低いことを示す．優先度が低いのは，SSD の並列性をうまく利用するために，同じチャンネルを用いるリクエストをできるだけ離してディスパッチする為である．

4.5.3.2　Write データ管理マトリックスとサブグループの状態

Write データ管理マトリックスはサブグループ内のサブキューに存在する Write リクエストの数と，Write リクエストに関するサブグループの状態を管理する．Write に関するそれぞれのサブグループの状態は下記の３つがある．

- ZERO:該当サブグループに Write 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 Write リクエストが存在するが，処理されていないことを示す．ディスパッチ対象としての優先度が低い．
- 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．Read リクエストと Write リクエストの干渉を防ぐ為に，同じチャンネルを使用する Write リクエストを出来るだけまとめてディスパッチする為である．

4.6　インサートポリシー

AC スケジューラが I/O リクエストのインサートを行うときのアルゴリズムについて述べる．図 4.1 で示すようにまず，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．

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

Algorithm 1 Insert Algorithm

1: Calculate Proper Sub-Group Index
2: $DeadlineQueue \leftarrow Request$
3: $SubGroupQueue \leftarrow Request$
4: $RequestCounter \leftarrow RequestCounter + 1$
5: $FLAG \leftarrow PEND$

表 4.3　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

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 (jiffies6.5.1) 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 6) 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 4.3 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 1)

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．

4.6.1　インサート時のデータ管理マトリックスのアップデート

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が，その詳細な手順を示す．

4.6.1.1　インサート時の Write リクエストデータ管理マトリックスのアップデート

Write リクエストがスケジューラのキューにインサートされた時の AC スケジューラの挙動を示す．

1. Write リクエストのアドレスの SubGroupIndex に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する．
2. サブグループの状態を表すフラグを PEND フラグに設定する．

4.6.1.2　インサート時の Read リクエストデータ管理マトリックスのアップデート

Read リクエストの時もインサート時にはほぼ同様である．

1. Read リクエストのアドレスの SubGroupIndex に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する．
2. サブグループの状態を表すフラグを PEND フラグに設定する．

4.7　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，Algorithm3，Algorithm4 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．

4.7.1　ディスパッチポリシーにおけるデッドライン機構の実現

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 6). 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

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:     Dispatch ASYNC READ
14: end if
```

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 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る．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関連研究結果に基づいたものである [13, 14](要件 5) . 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．

4.7.2　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 Algorithm4 及び，Algorithm3 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 2).

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．

4.7.2.1　マトリックスによる Read リクエスト選択と，Read データ管理マトリックスのアップデート

Algorithm3 に Read データ管理マトリックスを用いてのディスパッチ対象となる Read リクエストの選択方法についての擬似コードを掲載した．この擬似コードの内容を詳細に述べる．

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．よって，複数のサブグループに I/O リクエストが入っていた場合には，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，分散してディスパッチさせようと試みる. リクエストが存在しないサブグループには ZERO フラグが立っていることから判断可能である．つまり，サブグループ３，サブグループ５，サブグループ６が Read リクエストを保持していた場合，$3 \rightarrow 5 \rightarrow 6$ といった順番でリクエストのディスパッチを行う．もちろんサブグループ 2 しかリクエストを保持していない場合には，サブグループ２の Read リクエストを連続してディスパッチする．

また Read と Write は同時にディスパッチされ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，Write リクエストに関して，ディスパッチ優先度が高い PROC フラグが立っているサブグループはディスパッチ対象としての優先度を下げる．例えば，Read データ管理マトリックスを参照して，サブグループ 3, サブグループ 4, サブグループ 5 に Read リクエストが入っていても，同時に Write データ管理マトリックスを参照し，サブグルー

Algorithm 3 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
```

プ 4 のフラグが PROC，つまり，優先度が高かった場合には，サブグループ 3，サブグループ 5 の 2 つに分散してリクエストを発行する．

次にディスパッチ対象として Read リクエストをディスパッチした後に，Read データ管理マトリックスのアップデートを行う．

1. 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．
2. デクリメント後のリクエスト数が 0 になった場合には，ZERO フラグを設定し，それ以外の時には END フラグを設定する．
3. 全てのフラグが ZERO になった場合には終了，ZERO と END だけになった場合には，END フラグを PEND フラグに戻す．

4.7.2.2　マトリックスによる Write リクエスト選択と，Write データ管理マトリックスのアップデート

Algorithm4 に Write データ管理マトリックスを用いてのディスパッチ対象となる Write リクエストの選択方法についての擬似コードを掲載した．この擬似コードの内容を詳細に述べる．

Algorithm 4 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 $0 \leq i \leq MaxSub - GroupIndex$ **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: $Sub - Group'sFLAG \leftarrow ZERO$
9: **else**
10: $Sub - Group'sFLAG \leftarrow PROC$
11: **end if**
12: **end if**
13: **end for**

ディスパッチ対象として Write が選択された場合には，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る．ここで，PROC フラグが立っているサブグループは，前回ディスパッチ対象として採択され，ディスパッチ対象としての優先度が高い．そのため，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．

例を挙げると，サブグループ 3, サブグループ 5, サブグループ 7 に Write リクエストが入っている時には，まずサブグループ 3 をディスパッチ対象として決定 (PROC フラグを立てる) し，サブグループ 3 のリクエストが空になるまでサブグループ 3 をディスパッチ対象として決定．空になった時（ZERO フラグになった時）には，Write リクエストが入っている別のサブグループ (今回の場合はサブグループ 5，またはサブグループ 7) をディスパッチ対象として選択し，また同じ操作を繰り返す．

次にディスパッチ対象として Write リクエストを選択し，ディスパッチした後の Write データ管理マトリックスのアップデートについて述べる．

1. 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.
2. デクリメント後のリクエスト数が 0 になった場合には，ZERO フラグを設定し，それ以外の時には PROC フラグを設定する.
3. 全てのフラグが ZERO になった場合に終了.

第 5 章

評価及び議論

5.1 評価

5.1.1 評価環境

表 5.1 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 として実装し，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 [23].

5.1.2 fio ベンチマーク

fio はディスク I/O を計測するベンチマークツールである．

- 連続 Read，連続 Write，ランダム Read，ランダム Write など一般的なベンチマーク

表 5.1 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

で計測されるアクセスパターンだけではなく，Read と Write が一定の比で混合されているようなアクセスパターンも計測可能な為，実際の運用に近いリクエストパターンを生成可能.

- ベンチマークに使用するファイルサイズの設定，使用する IO エンジンの変更，同時に走らせるスレッドの数，一度に平行して非同期読み書きを行う最大数の指定など細かいパラメータの指定が可能.
- 広く SSD の性能測定の為に使用されている.

といった理由から今回このベンチマークツールを採択した.

5.1.3　評価パラメータ

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)

また SSD の性能を正確に測る為に，RDB など，アプリケーション自体にキャッシュが搭載されている場合などに使用される O_DIRECT フラグを I/O リクエストに立てて，キャッシュを経由しない場合のベンチマーク結果を測定した.

また，キャッシュを ON にした場合それぞれのベンチマーク結果についても同様のリクエストパターンを用いて測定した.

5.2　結果

5.2.1　キャッシュを経由しなかった場合

図 5.1 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 5.2 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.

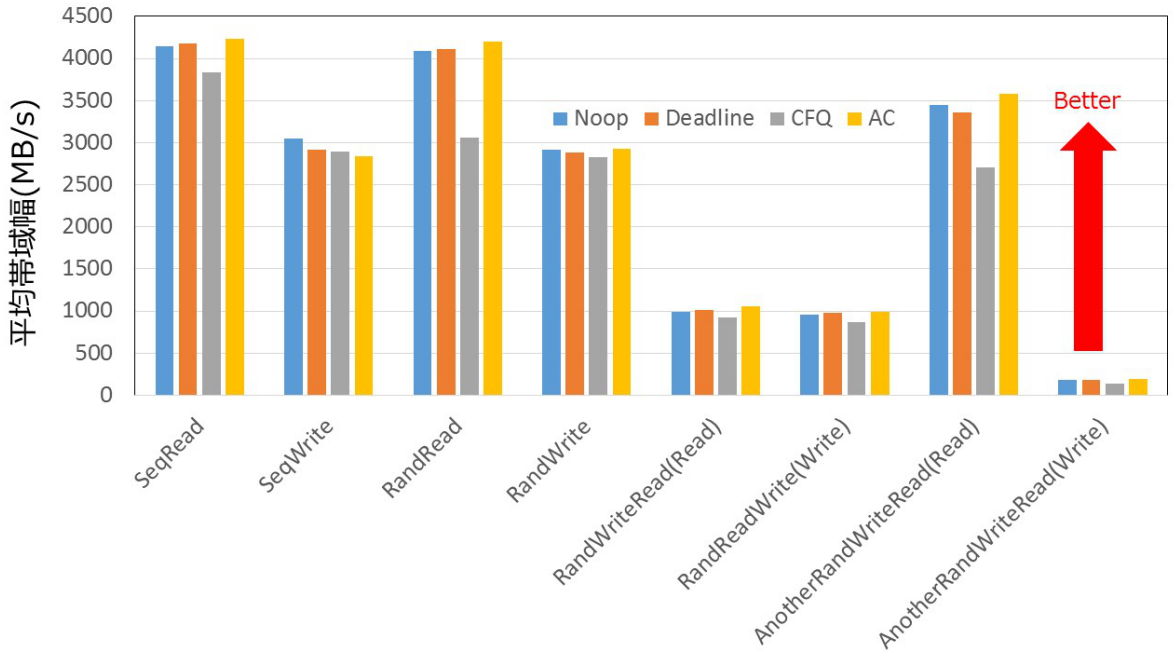

図 5.1　様々なダイレクト I/O リクエストパターンにおいて各 I/O スケジューラにが達成する平均帯域幅

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 5.2 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．

5.2.2　キャッシュを経由した場合

また同様にキャッシュを有効にした場合の結果を，帯域幅について図 5.3 に，レイテンシについて図 5.4 に示した．キャッシュを有効にした場合，他のスケジューラと比べて，相対的に性能が低下してしまっていることがわかる．

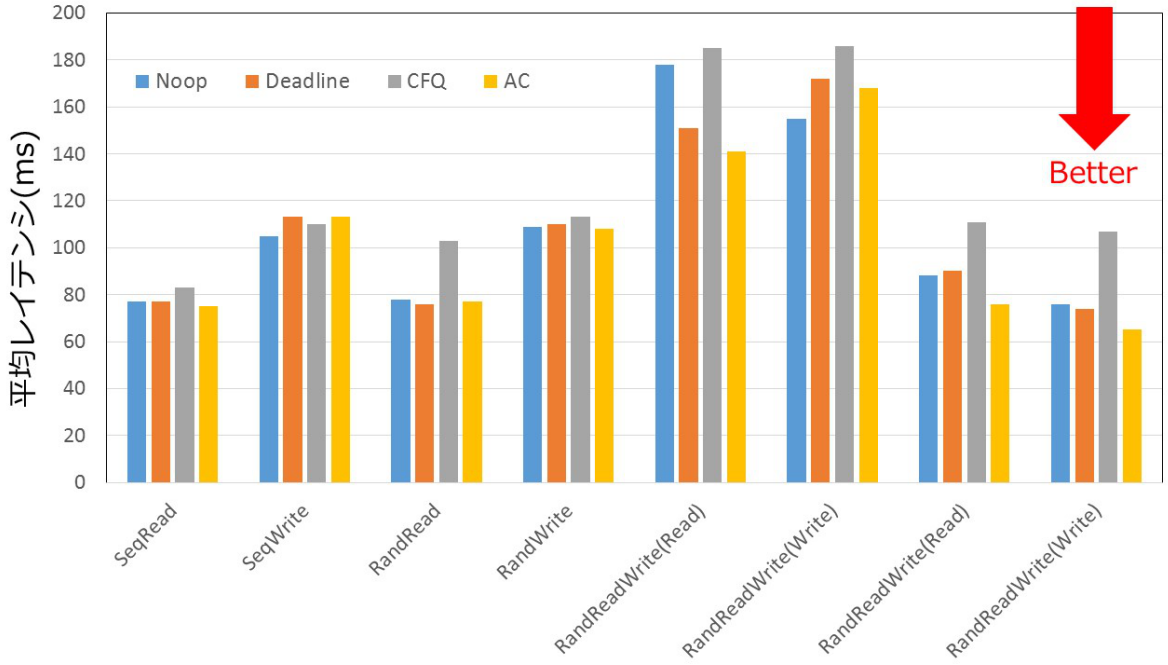

図 5.2　様々なダイレクト I/O リクエストパターンにおいて各 I/O スケジューラが達成する平均レイテンシ

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．設定した要件は以下の通りであった．

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．

まず，設定要件 4 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．

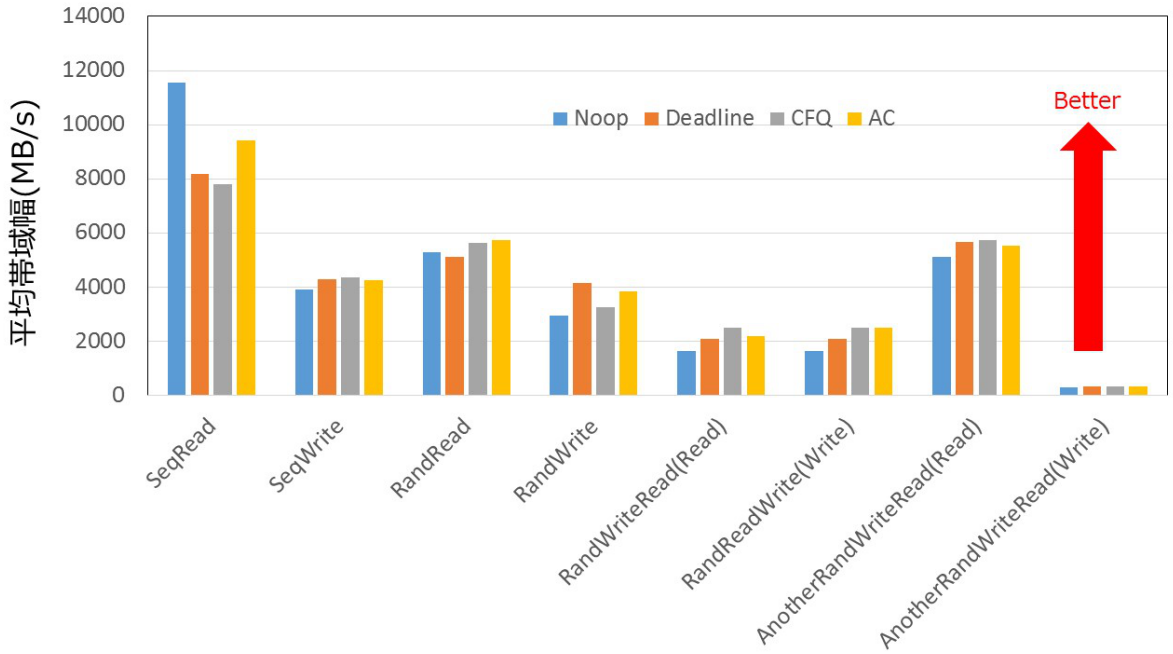

図 5.3　キャッシュ有効時に様々な I/O リクエストパターンにおいて各 I/O スケジューラにが達成する平均帯域幅

次に設定要件 2 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 3 と設定要件 5 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る．この結果，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 5.2 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 3 が達成されていることがわかる．また図 5.1 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 5 を満たしていることが分かる．

次にキャッシュを有効にした場合の結果，図 5.3，図 5.4 についてだが，AC スケジューラの性能が相対的に低下してしまっている．この原因として，AC スケジューラが発行する I/O リクエストがキャッシュにぶつかりにくい様な発効の順番になっている事が原因だ

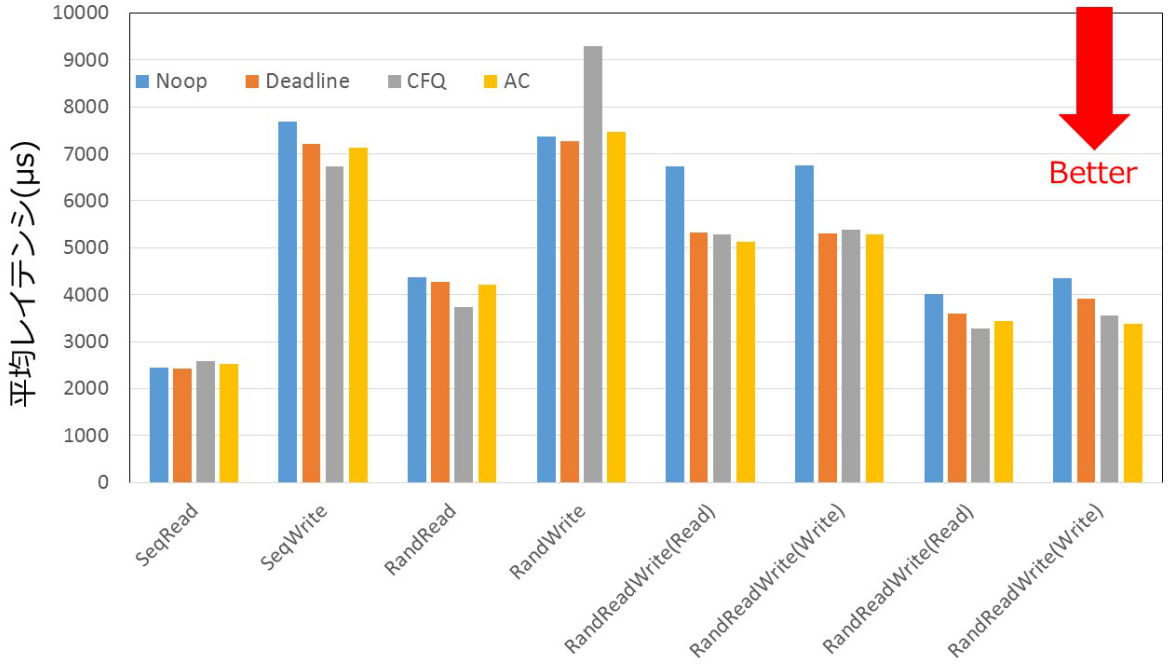

図 5.4　キャッシュ有効時に様々な I/O リクエストパターンにおいて各 I/O スケジューラが達成する平均レイテンシ

と考えられる．

またデッドライン保障機構については，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 6 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 1 が有効であることを示せる評価は本論文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キャッシュを無効にし，SSD そのものの性能についてのベンチマークの場合，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 アクセスの 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 [24].

また，キャッシュを有効にした場合の結果から，キャッシュの実装についても同時に考える必要や，AC に適したキャッシュの提案などが考えられる．また，それだけではなく，RDB や KVS など独自のキャッシュを持つアプリケーションについても独自キャッシュやリクエストの発効手法について本論文の手法を適用することによって，SSD の性能を大幅に引き出すことができると考えられる．なので，I/O スケジューラだけではなく，更に包括的な SSD のための I/O の枠組みが必要であることがわかる．

第 6 章

結論

6.1 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 スケジューラにおいては，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，データ管理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，キャッシュを使用しない場合，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．しかし，連続 Write の時には，性能が低下してしまっているので，連続 Write の性能を向上させる工夫が今後必要である．

キャッシュを利用した場合には，キャッシュを使用しない場合と比べると，相対的に既存のスケジューラに負ける事が増えてしまった．これは，提案したスケジューラで発行するリクエストの順番が今のキャッシュの実装に適さない事が原因だと考えられる．前節から，今回の研究は RDB，KVS など，キャッシュを自分自身に内蔵しているアプリケーションの性能向上の一助とも成り得ると考えられる．

6.2　今後の予定

また，提案手法は，Dynamic Kernel Module として実装されているので，今後様々なシステムにおいて，導入が容易であり，PCIe インタフェース向けの SSD においても同様の手法の適用，及び改善の余地が見られる．

謝辞

本研究を進める，また本論文を執筆するにあたり，終始熱心な御指導をして頂いた吉瀬謙二准教授に感謝致します.

また，日常の議論において，今後の方針，多くの知識を頂いた吉瀬研修室の皆様に感謝いたします.

また，学費，家賃などのサポートだけでなく，相談などを何も言わずに行ってくれた家族に深い感謝を致します.

参考文献

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference*, pp. 57–70, 2008.

[2] Jaehong Kim, Sangwon Seo, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. Parameter-aware i/o management for solid state disks (ssds). *IEEE Transactions on Computers*, Vol. 61, No. 5, pp. 636–649, May 2012.

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

[4] Feng Chen, David A Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 37, pp. 181–192. ACM, 2009.

[5] Li-Pin Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, pp. 1126–1130. ACM, 2007.

[6] Werner Bux and Ilias Iliadis. Performance of greedy garbage collection in flash-based solid-state drives. *Performance Evaluation*, Vol. 67, No. 11, pp. 1172–1186, 2010.

[7] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, p. 10. ACM, 2009.

[8] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Chao Ren. 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.

[9] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy

and data granularity. In *Proceedings of the international conference on Supercomputing*, pp. 96–107. ACM, 2011.

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

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

[12] Lei Han, Yi Lin, and Weiwei Jia. A new i/o scheduler designed for ssd through exploring performance characteristics. In *Computing, Communications and IT Applications Conference (ComComAp), 2014 IEEE*, pp. 161–166, Oct 2014.

[13] Jaeho Kim, Yongseok Oh, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H Noh. Disk schedulers for solid state drivers. In *Proceedings of the seventh ACM international conference on Embedded software*, pp. 295–304. ACM, 2009.

[14] Seungyup Kang, Hyunchan Park, and Chuck Yoo. Performance enhancement of i/o scheduler for solid state devices. In *Consumer Electronics (ICCE), 2011 IEEE International Conference on*, pp. 31–32. IEEE, 2011.

[15] MARCUS PAUL DUNN. *A new I/O scheduler for solid state devices*. PhD thesis, Texas A&M University, 2009.

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

[17] Ji-Yong Shin, Zeng-Lin Xia, Ning-Yi Xu, Rui Gao, Xiong-Fei Cai, Seungryoul Maeng, and Feng-Hsiung Hsu. Ftl design exploration in reconfigurable high-performance ssd for server applications. In *Proceedings of the 23rd international conference on Supercomputing*, pp. 338–349. ACM, 2009.

[18] Myoungsoo Jung, Ellis H Wilson III, and Mahmut Kandemir. Physically addressed queueing (paq): improving parallelism in solid state disks. In *ACM SIGARCH Computer Architecture News*, Vol. 40, pp. 404–415. IEEE Computer Society, 2012.

[19] Feng Chen, David A Koufaty, and Xiaodong Zhang. Hystor: making the best use of solid state drives in high performance storage systems. In *Proceedings of the international conference on Supercomputing*, pp. 22–32. ACM, 2011.

[20] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: Introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pp. 22:1–22:10, New York, NY, USA, 2013. ACM.

[21] Patrick Mochel. The sysfs filesystem. In *Linux Symposium*, p. 313, 2005.

[22] Stan Park and Kai Shen. Fios: a fair, efficient flash i/o scheduler. In *FAST*, p. 13, 2012.

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

[24] Mingyang Wang and Yiming Hu. An i/o scheduler based on fine-grained access patterns to improve ssd performance and lifespan. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pp. 1511–1516, New York, NY, USA, 2014. ACM.

[25] qemu. `http://wiki.qemu.org/Main_Page`.

[26] Buildroot. `https://buildroot.org/`.

[27] Linux kernel source tree. https://github.com/torvalds/linux.

[28] kdb. `http://elinux.org/KDB`.

[29] gdb. `https://www.gnu.org/software/gdb/`.

[30] blktrace. `http://www.cse.unsw.edu.au/~aaronc/iosched/doc/blktrace.html`.

[31] The gnu operating system and the free software movement. `http://www.gnu.org/`.

[32] Linux kernel design patterns - part2. https://lwn.net/Articles/336255/.

[33] Red-black Trees (rbtree) in Linux. https://www.kernel.org/doc/Documentation/rbtree.txt.

[34] cgroups. `https://www.kernel.org/doc/Documentation/cgroup-v1/`.

研究業績

1. 奥村 開里, 小林 諒平 吉瀬 謙二: SSD の並列性を引き出す I/O スケジューラ, 研究報告システムソフトウェアとオペレーティング・システム, 御茶ノ水女子大学 (2015 年 11 月 24 日発表).

付録

6.3 実装

今回本論文において提案した AC スケジューラは Linux Kernel v4.x 上に実装した．v3.x 系においても数行の変更により，動作することを確認した．

具体的な実装方法として，エミュレーション環境において動作させる事が目的の段階の時には，カーネルに組み込んでビルドし，動作を確認した．

次に実際に実機において動作を確認，及びデバッグを行う際には，Dynamic Kernel Module としてスケジューラを実装した．

これらのエミュレーション及び実機の環境において詳細を記す．

6.3.1 エミュレーション環境

デバッグ段階においては、Kernel Panic や Kernel Oops が発生し、syslogd まで動作が停止することが考えられ、デバッグが難航すると考えられるので、x86_64 のサンドボックス環境を用意した．

6.3.1.1 QEMU

実際のエミュレーションにはオープンソースのプロセッサエミュレータである QEMU[25] を用いてエミュレーションを行った．qemu-system-x86_64 コマンドを用いることによって x86_64 環境を想定したエミュレーションを行った．

6.3.1.2 ファイルシステム

ファイルシステムの構築として Buildroot[26] を使用した．ビルドオプションはまず make qemu_x86_64_defconfig として，エミュレーションに用いる QEMU 用の設定ファイルを作成し，さらに，使用するファイルシステムとして ext2 と ext4 を使用するように変更を行った．

その後ビルドを行い，エミュレーションに必要なファイルシステムを構築した．

6.3.1.3 Kernel

シミュレーションにおいて使用した Linux カーネルは Github[27] からクローンしたものを使用した．ビルドオプションは，make defconfig としてデフォルト設定で作成したものを使用した．

6.3.2 実機

実機での評価を行う場合，コンパイルの手間，リブートの煩雑さなどの理由から Dynamic Kernel Module として実装を行った．具体的には評価対象のモジュールを評価マシンでコンパイルを行い，insmod することにより，カーネルに挿入した．実機は v4.x 系が必要だったために，開発中ではあるが，ubuntu の v4.x 系のカーネルを搭載しているものを使用した．

6.3.3 デバッグ

デバッグ段階においては Kernel Oops や Kernel Panic が発生し，カーネルがハングアップしてしまう箇所があった．そこでデバッグを行う際には，kgdb，kdb[28]，gdb[29],blktrace[30] などのツールを用いた．

6.3.3.1 print debug

カーネル内部では標準 C ライブラリが存在しないので，/var/log/messages バッファに printk を用いてメッセージを出力したり，QEMU の仮想シリアルポートをホストの標準入出力や/dev/tty0 にリダイレクトさせることによってプリントデバッグを行っていた．

しかし，メモリ上のダンプ情報がリブートによって破棄されるのでこの手法は煩雑であることが多々あった．しかし，コンソールの初期化が始まるまでの前のブートプロセスにおいても printk が動作していたので，アーキテクチャ毎の設定を行うブートプロセス以前でのデバッグには非常に有効だと思われる．

6.3.3.2 GDB,KGDB

gdb は GNU プロジェクト [31] のデバッガである．gdb は、プログラムの実行の変更や追跡といったデバッグに有用な機能を提供する。プログラム内部の変数の値を修正、監視したりすることや、プログラムの通常の動作とは別に関数を呼び出すことが可能である．

また，kgdb は，Linux カーネルをソースレベルでデバッグする際に使用される．kgdb は gdb の遠隔モードを利用している．遠隔モードにおいてはデバッグ対象のプログラムと gdb を異なるマシンで動作させ，GDB プロトコルをシリアルや TCP/IP 経由で理解する遠隔スタブと呼ばれるものを用いて通信することができる。この遠隔モードにより，通常のアプリケーションプログラムのようにカーネルを動作させ，gdb によってデバッグを行う事ができる．具体的には，カーネルのコードにブレークポイントを設定することができたり、ステップ動作、変数を参照可能となる。また，ハードウェアの debug register が使えるアーキテクチャでは、ウォッチポイント (データブレークポイント)

を設定することが可能となる．よって，ウォッチポイントも設定可能な今回の x86_64 環境においては，非常に有用であった．

ただしデバッグを行う際には, CONFIG_DEBUG_KERNEL,CONFIG_DEBUG_INFO,CONFIG_KGDB を有効にし，デバッグシンボル情報をビルド時に作成するようにした．またカーネルがメモリ上で再配置され，デバッグシンボルとの不整合の発生を防ぐために，CONFIG_RELOCATABLE を無効にした．

そして，Kernel Oops や Kernel Panic が発生した箇所で，逆アセンブル，元のコードとの対応を行いデバッグを行った．

6.3.3.3 blktrace

blktrace はアプリケーションが発行した I/O リクエストがドライバに渡されるまでをトレースするツールである．Kernel v2.6 以降は公式にサポートされているので，パッチをカーネルに当てることもなく使用する事ができる．

blktrace を用いることによって，I/O リクエストが長時間 PENDING 状態にあったり，いつまでもキューにインサートされてないなどの問題を早期に発見することができた．

6.4 使用したデータ構造

Linux Kernel 内においては標準 C ライブラリを使用することができないので，自前で用意したデータ構造や，カーネル内で提供されているデータ構造を使用する必要がある．基本的には，可読性を高めたり，最適化がすでに行われているといった理由から，カーネルのソース lib ディレクトリで宣言されているデータ構造を用いる事が推奨されている．

今回実装した AC スケジューラについても，Linux コミュニティのポリシーにしたがって Linux Kernel 内で提供されている List や，赤黒木 (Red-Black Tree) を用いた．

6.4.1 List

AC スケジューラの実装を行う際，デッドラインキューや，サブキューなどを実装する際には，Linux カーネルが提供する list_head 構造体 (listing6.1) を用いた．

Listing. 6.1 struct list_head

```
struct list_head {
  struct list_head *next, *prev;
};
```

コンパイル時計算 (list_entry マクロ：実態は container_of マクロ) を用いて，この list_head 構造体が埋め込まれている (親関係にある) 構造体に対するポインタを計算する事ができる．よってこの list_head 構造体によって，List が実現できる．コンパイル時計算を行う container_of マクロを listing6.2 に示した．

Listing. 6.2　コンパイル時計算を行う container_of マクロ

```
#define container_of(ptr, type, member) ({          \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
    (type *)( (char *)__mptr - offsetof(type,member) );})
```

一見理解しづらいが，Linux Kernel において多用されている Embbeded Anchor パターン [32] であり，どんな構造体にも汎用的に使えるという利点があるために，広く利用されている．

6.4.2 Red-Black Tree

AC スケジューラの実装段階においては，サブグループ内のサブキューをカーネルが提供している赤黒木 [33] を用いて実装し，論理アドレスをキーにしたソートを行ってからディスパッチするという案や，サブグループを用意せずに，赤黒木を用いた巨大なソートされたツリーを用意し，SSD の並列性を抽出するようにディスパッチするという案もあった．（最終的には，可読性の悪さと計算量を比較し，サブグループを用いた）これは計算量を抑えたまま $(\log n)$ ソートされたリストを保持したいと言った理由があったからである．

結論の章で述べたように連続 write の性能を向上させる関連研究 [24], を用いる際には，write リクエストに関して，論理アドレスごとにソートされていると望ましいので，赤黒木を用いての実装を考えている．

6.5 実装に使用した Linux Kernel API

6.5.1 jiffies

デッドラインを実装する際には，カーネルが起動してからの tick(1/HZ) 単位をカウントしているタイマーである jiffies を使用した．jiffies は tick 秒ごとに，１ずつインクリメントされ，カーネル内部で時間を図るときに広く用いられている．kernel/timer.c で提供されている．

6.5.2 I/O scheduler API

I/O スケジューラを実装する際には，ブロックレイヤーにて呼ばれる関数を登録していくといった形で実装する．（関数ポインタをコールバックとして登録していく．）実装する関数として以下に示すものがある．

- elevator_merge_req_fn: I/O リクエストがマージを行えるか問い合わせる関数.
- elevator_merged_fn: ２つの I/O リクエストがマージされた時に呼び出される関数．片方のリクエストはマージされるとスケジューラの管轄外になる．
- elevator_allow_merge_fn: ブロックレイヤーが安全にマージを行えると判断した時に必ず呼び出される関数．スケジューラの内部の処理的によってはマージを行われると困るといった場合などにマージを行わないようにしたりできる．
- elevator_dispatch_fn: ディスパッチキューに入れる処理を行う関数．スケジューラはディス

パッチされた後のリクエストには操作を行う事ができない．

- elevator_add_req_fn: リクエストをスケジューラにインサートするために呼ばれる関数.
- elevator_former_req_fn: ブロックレイヤがマージを行えるかどうかを判断するのに使用する関数．I/O リクエストの操作対象のアドレスを基準として前のリクエストを返却する．
- elevator_latter_req_fn: ブロックレイヤがマージを行えるかどうかを判断するのに使用する関数．I/O リクエストの操作対象のアドレスを基準として後のリクエストを返却する．
- elevator_completed_req_fn: リクエストが完了した時に呼ばれる関数.
- elevator_may_queue_fn: キューのリミットを超えていたとしてもスケジューラがエンキューを許可するときに true を返却させる関数．挙動がおかしくなる危険が高いので，慎重に使用することが求められている．
- elevator_set_req_fn: リクエストに対して，特有の記憶領域を割り当てる時に必ず使われなければならない関数.
- elevator_put_req_fn: リクエストに対して，特有の記憶領域を除去する時に必ず使われなければならない関数.
- elevator_activate_req_fn: デバイスドライバがリクエストを始めて処理するときに呼ばれる関数．I/O スケジューラはこの関数をいつ実際のリクエスト処理が始まるかを決定する為に使用できる．
- elevator_deactivate_req_fn: デバイスドライバがリクエストをリキューし，処理を遅らせた時に呼ばれる．
- elevator_init_fn: I/O スケジューラ特有のデータを割り当てるときに使われる．
- elevator_exit_fn: I/O スケジューラ特有のデータを除去するときに使われる．

以上の様に細かく設定されているので，細かい実装が可能である．また，cgroups[34] を用いた API なども提供されているので，cgroups の API を用いた実装も可能である．