Vol.2022-ARC-248 No.31
Vol.2022-SLDM-198 No.31
Vol.2022-EMB-59 No.31
2022/3/10

BFQ スケジューラにおける ストレージ I/O の帯域保証による応答性向上

長谷川博紀[1]　松原豊[1]　加藤寿和[2]　山本整[2]　水口武尚[2]　高田広章[1]

概要：　近年，車載情報機器のような組込み機器においても，機器メーカが開発する基本機能に加え，第三者が開発したアプリを導入可能とすることが検討されている．複数のアプリを組込み機器に統合する場合，追加されたアプリによる基本機能への悪影響を防止したいという要求がある．BFQ スケジューラは，マルチキューによるスループット向上と，weight を用いた帯域保証による高い応答性の実現を目的として開発された．しかし，100MB の巨大な direct I/O が発行された時に分配率（weight で指定した各グループからの I/O の比率）が崩れた．本論文では，まず，基本機能に相当する動画再生プロセスと，第三者アプリに相当する I/O 負荷生成プロセスを同時に実行した際に，動画再生で発生する遅延の原因を明らかにする．さらに，2 つのプロセスからの I/O を，分配率に影響が出ない大きさに分割することで，巨大な I/O に対しても weight に基づく分配率を実現する．提案手法を適用することで，weight を 2：1 に設定した場合の分配率が 1.11：1 から 1.93：1 まで改善され，その結果，動画再生プロセスの遅延も解消することを確認した．

1. はじめに

近年，車載情報機器のような組込み機器においても，機器メーカが開発する基本機能に加え，第三者が開発したアプリケーションを導入可能とすることが検討されている．複数のアプリケーションを組込み機器に統合する場合，追加されたアプリケーションによる基本機能への悪影響を防止したいという要求がある．複数アプリケーションが共有するストレージデバイスを対象にした，アプリケーションの独立性とリアルタイム性の両立を実現するために，Linux の BFQ（Budget Fair Queueing）スケジューラ [1] に注目した．BFQ スケジューラは，マルチキューによるスループット向上と，cgroup io.weight を用いた帯域保証による高い応答性の実現を目的として開発された．cgroup io.weight は，グループ毎に weight を与え，その weight の比率に従って，帯域幅を分配する機能である．

まず，どのような I/O パターンが基本機能に悪影響を及ぼすのかを調べるために，基本機能に相当する動画再生プロセスと，第三者アプリケーションに相当する I/O 負荷生成プロセスを同時に実行し，動画再生で遅延が発生するかどうかを確認した．次に，基本機能となるアプリに高い weight を与えることで，動画再生に優先的に帯域を割り当てることによる改善を試みたが，動画再生での遅延を抑制できなかった．動画再生での遅延要因と weight に従った帯域分配ができなかった要因の分析結果と，それらの解決方法を提案する．

本論文の貢献は以下の通りである．

- データサイズが大きい（本論文では 100MB 以上のデータサイズを指す）Direct I/O が発行された時に分配率（weight で指定した各グループからの I/O の比率）が崩れ，基本機能となる動画再生で遅延が発生することを明らかにした．
- 遅延要因となるアルゴリズムと分配率が崩れる要因となるアルゴリズムを見つけるために，各レイヤーでの I/O 量や BFQ アルゴリズムを調査した．
- BFQ において，2 つのプロセスからの I/O を分配率に影響が出ない大きさに分割することで，データサイズが大きい I/O アクセス（以降，I/O と表記する）に対しても weight に基づく分配率を実現した．
- I/O を分割後，動画再生プロセスと I/O 負荷生成プロセスの 2 つのプロセスを同時実行し，動画再生での遅延が改善されたかを評価した．

性能評価には，組込み機器での利用を想定し，様々な組込み機器で搭載実績のある eMMC を搭載した機器を使った．分割によるオーバーヘッドは，全プロセスのスループットの合計量で評価した．その結果，平均スループットの低下

[1] 名古屋大学 大学院情報学研究科
[2] 三菱電機株式会社 情報技術総合研究所

Vol.2022-ARC-248 No.31
Vol.2022-SLDM-198 No.31
Vol.2022-EMB-59 No.31
2022/3/10

は 1%程度であり，誤差と言えるほどであった．

2. Linux I/O システム

2.1 BFQ スケジューラ

図 1 に，Linux での I/O 処理構造を示す．ブロックレイヤーはファイルシステムとデバイスドライバの間に存在している．I/O スケジューラは，I/O の並び替えや結合によって，応答性やスループットを向上している．Linux 5.10.83 のマルチキュー対応 I/O スケジューラは，NONE, mq-Deadline, Kyber, BFQ の 4 種類である．BFQ は，高い応答性を目的に開発されており，ソフトリアルタイムなアプリケーションに最適なスケジューラである．今回は，カーナビ等の組込み機器での利用を想定しているため，高い応答性が得られる BFQ を利用する．BFQ では，リクエストキュー毎に budget を割り当てることで，I/O 量の管理をしている．budget とは，プロセスが一度にストレージデバイスに送信できる I/O サイズである．I/O 処理の一連の流れは以下のようである．

(1) 次に処理するリクエストキューの選択
(2) 割り当てられた budget 分の I/O 量だけ処理
(3) budget の使用状況に基づいて，次の budget を調整
(4) (1) に戻る

デバイスへの経路は 1 本なので，複数のリクエストキューから処理するリクエストキューを 1 つ選択する必要がある．

blktrace を使うことで，ブロックレイヤー内のトレースポイントを通過した回数や時間や，トレースポイントを通過した I/O のサイズや番地，発行しているプロセス名などを知ることができる．図 1 に，計測可能なトレースポイントの一部を示す．Queue のポイントではブロックレイヤーへ入ってきた時，Dispatch と Complete で実際にストレージに処理を要求，処理が完了した時の I/O の情報を知ることができる．他のトレースポイントは，I/O の分割や結合などがある．ブロックレイヤーは，デバイスドライバの直前であるため，この部分での I/O の流れを知ることで実際にストレージデバイスに向けて発行されている I/O を知ることができ，ストレージデバイスへの負荷の計測に適している [2].

2.1.1 budget の動的調整

BFQ は，高い応答性以外にも，スループットをできる限り高めるためのアルゴリズムがある．その 1 つが budget の動的調整である．表 1 にその更新値を示す．リクエストキューに処理が割り当てられた際に，そのキューに設定された budget を使い切ったのか，budget を使い切る前にリクエストキューが空になったのか，処理時間がタイムアウトとなったのかで処理が分岐する．min_budget は budget に設定できる最小値，max_budget は budget に設定できる最大値である．I/O 量が多いプロセスに対して budget を増やすことで，そのプロセスは一度に I/O を処理することができるためスループットが向上する．

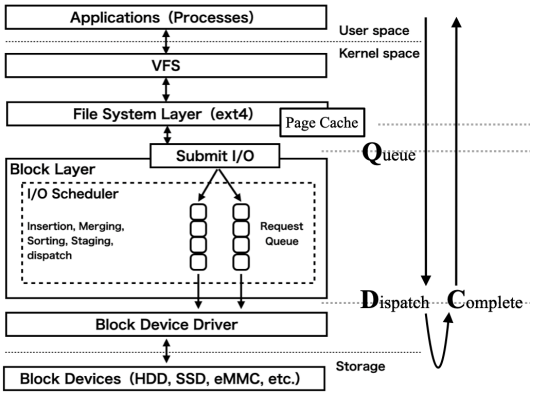

図 1 Linux での I/O 処理構造

表 1 キュー切り替え時の budget の更新値 [3]

		キューに残っている I/O	
		あり	なし
タイムアウト	あり	if budget > 5*min_budget then -= 4*min_budget; else min_budget;	min(budget*2, max_budget)
	なし	max(前回の使用量, min_budget)	min(budget*4, max_budget)

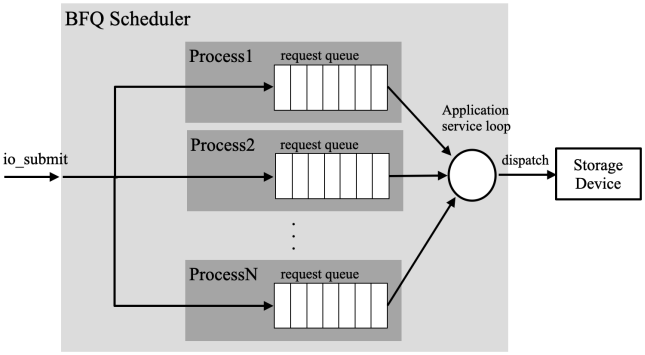

図 2 BFQ スケジューラの構成

2.1.2 キューの選択アルゴリズム

BFQ は weight によって budget を増減させるのではなく，処理の割り当て回数を多くする．これは，処理時間を長く割り当ててしまうと他のプロセスが少量の I/O を処理することもできず，応答性が低くなるためである．BFQ では，処理するリクエストキューの選択に service（実際に使用された I/O 量）を使った [service/weight] の指標を使っている [3]．この指標を本論文ではタイムスタンプと呼ぶ．BFQ は図 2 に示すようにマルチキュー用のスケジューラであるため，処理するリクエストキューを 1 つ選択する必要がある．ストレージデバイスへの dispatch 直前で weight とタイムスタンプを基準に，処理するリクエストキューを選択する．

処理順の計算には全てに共通の vtime, リクエストキュー毎の start, finish という変数を使用している．budget を 3000 セクタ固定で，常に使い切るとすると，図 3 に示す

Vol.2022-ARC-248 No.31
Vol.2022-SLDM-198 No.31
Vol.2022-EMB-59 No.31
2022/3/10

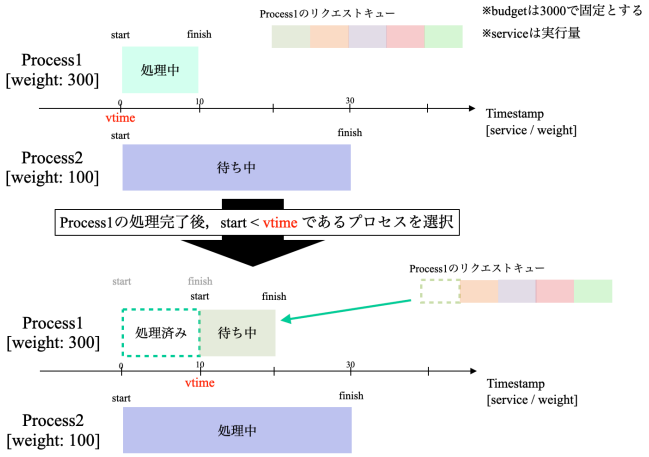

図 3 処理するキューの選択

ように，まずプロセス 1 とプロセス 2 がそれぞれ start と finish を設定し，vtime は start の値になる．次にプロセス 1 が選択され，budget 分だけ I/O を処理する．次の I/O のサイズが残りの budget 未満である限り，リクエストキュー内の I/O を連続で処理できる．その時，service/weight = 3000/300 = 10 だけタイムスタンプを増加させる．budget を使い切った時，vtime はプロセス 1 の finish の値になり，プロセス 1 の start と finish が次の I/O で更新される．その後, 次に処理するプロセスを start <vtime を満たす start を持つプロセスから選択する．そうして次に選択されるプロセスはプロセス 2 となり，budget を消費する．プロセス 2 は weight が 100 であるため，タイムスタンプの増加量は service/weight = 3000/100 = 30 となり，プロセス 1 の 3 倍増加する．これによって，次とその次はプロセス 1 が選択されることになり，プロセス 1 が 9000 セクタ処理する間にプロセス 2 は 3000 セクタしか処理できないため，weight 通りの I/O 量の分配になる．

また，読み込みよりも書き込みの方が，同じ I/O サイズでもデバイスの占有時間が長い．そのため，同等に扱うと読み込みのスループットの低下に繋がる [4]．これに対応するために，BFQ では非同期 I/O を同期 I/O の 3 倍のサイズで扱うことで，書き込みによって読み込み処理が遅くならないようにしている．3 倍という数値は実測から導かれた数字である．

2.1.3 設定可能なパラメータ

BFQ には任意に設定可能なパラメータが用意されている [5] ．以下に主なパラメータを示す．

- slice_idle (default: 8s)
  処理中のリクエストキューが空になった時に，次の I/O 到着を待つ時間 [s]
- slice_idle_us (default: 8000us)
  上記の [us] 版
- fifo_expire_async (default: 250s)
  非同期 I/O に対する timeout の時間 [s]
- fifo_expire_sync (default: 125s)
  同期 I/O に対する timeout の時間 [s]
- max_budget (default: 0(未設定))
  budget の最大値 [sector]
- strict_guarantees (default: 0(off))
  on の場合は次の動作
  – 処理中のリクエストキューが空になった時に，必ず次の I/O 到着を待つ時間を設ける
  – ストレージデバイスに対して，同時に 1 つの I/O しか送らない

slice_idle を短くすると待ち時間が短くなるため，スループットの向上は見込めるが，同期 I/O を使うプロセスの場合，処理が終わったと勘違いして処理権限が次のプロセスに移ってしまい，逆にスループットや応答性が落ちる可能性がある．strict_guarantees は，ストレージデバイス内のスケジューラによる I/O の処理順の並べ替えを防ぐことができるため，BFQ でスケジュールした順序で処理することを保証できる．一方で，ストレージデバイスへの送信頻度が低下するため，スループットは低下する．これらのように，使用用途に合わせて各パラメータを設定することで性能を調整できる．

2.2 cgroup v2 io.weight

cgroup io.weight は，グループ毎に weight を与え I/O 帯域幅を weight に従った比率で分配する機能である [6]．weight を使用するためには，CFQ または BFQ スケジューラを使用する必要がある．これは，cgroup io.weight の機能がスケジューラの機能と深く関係しているためである．Ubuntu20.04 では，マルチキュー用のスケジューラが用意されており，CFQ は使えなくなっている．そのため，BFQ での cgroup io.weight の使用が一般的である．

cgroup では，プロセス単位毎に所属するグループを設定する．グループ毎に weight を持ち，それに従って I/O 帯域幅が分配される．同グループ内では割り当てられた帯域幅を上限に I/O が処理される．例えば，図 4 のように Group1 の weight を 100, Group2 の weight を 400, Group3 の weight を 500 と設定すると，Group1：Group2：Group3 = 1：4：5 の割合で帯域幅が割り当てられる．また，Group1 の下に Group1.1：Group1.2 = 3：2 とグループを作成し，ツリー構造にできる．その場合は，Group1 に割り振られた 10%から更に割り振るため，Group1.1 は全体の 6%，Group1.2 は，全体の 4%となる．重要なプロセスに対して高い weight を与えることで，重要でないプロセスよりも優先的に帯域幅が割り当てることができ，多くのプロセスが並走していても重要なプロセスが処理されることを保証できる．

問題も残っており，1 つは Buffered Write が制御できないことである．Buffered Write は，Direct Write と異なり，

Vol.2022-ARC-248 No.31
Vol.2022-SLDM-198 No.31
Vol.2022-EMB-59 No.31
2022/3/10

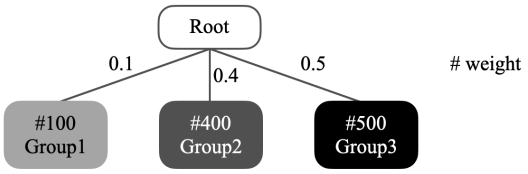

図 4 cgroup weight の帯域幅分配

ストレージデバイスに書き込む前にページキャッシュに書き込む．ページキャッシュに入りきらなくなった時や一定時間経過後に，ページキャッシュのデータをストレージデバイスに書き込む．ページキャッシュへの書き込みが終了した時点でシステムコールがリターンするために，高速である．しかし，cgroup io の制御箇所に到達する前にページキャッシュで I/O がまとめられ，その後 カーネルフラッシャースレッドによってフラッシュされるため，I/O と元のプロセスの関係が切れてしまい，cgroup がプロセスの消費した I/O 量を計算できないためである．

Buffered Write 以外には，グループの階層を作った時にうまく配分されていなかった問題であったり [7]，書き込み時の journal プロセスまでは制御対象に含まれていない等がある．

3. BFQ スケジューラの評価

3.1 遅延発生有無の確認

今回の目的は，様々な I/O パターンに対しても，基本機能となるアプリケーションに対して悪影響を出さないことである．そこで，Direct/Buffered I/O，Read/Write, ブロックサイズが 1MB/100MB の 8 パターンで動画に遅延が発生するかを BFQ スケジューラを使って調査した．評価環境を表 2 に示す．Direct I/O は，直接ストレージデバイスに対してアクセスするのに対して，Buffered I/O では，read はストレージデバイスからページキャッシュにデータを読み出し，そこからユーザーバッファにコピーする．write もページキャッシュに書き込むことで高速化している．ブロックサイズとは，１回の I/O 命令で発行する I/O サイズである．動画再生プロセスとしての vlc と I/O 負荷生成プロセスとしての fio を同時実行し，動画再生に遅延が発生するかを目視で確認した．fio は，設定ファイルを参照して，Direct I/O や Buffered I/O，ブロックサイズなどの設定した I/O パターンで負荷をかけることができるため，ベンチマーク測定に使われている．動画は full HD，mp4 形式（H.264），長さは 1 分である．fio の設定は，libaio ライブラリを使用し，非同期 I/O で，1GB のファイルを上記の 8 パターンで読み書きし，それを動画の再生終了までループさせている．

評価結果を表 3 に示す．Buffered Write の時と，ブロックサイズが 100MB で Direct I/O の時に，動画再生に遅延が発生した．Buffered Write は，root プロセスに処理が移

表 2 評価環境（Liva Z2[8]）

CPU	Intel Pentium (4cores)
RAM	8GB
Storage	eMMC(64GB)
OS	Ubuntu 20.04 LTS
Kernel	Linux 5.10.83
動画再生	vlc
I/O 負荷生成	fio[9]

表 3 様々な I/O パターンでの動画への遅延発生有無

ブロックサイズ [MB]	Direct Read	Buffered Write	Direct Read	Buffered Write
1	無	無	無	有
100	有	無	有	有

行するため cgroup io.weight で制限できないという課題があるため，予想通りの結果である．今回は，大きいブロックサイズの時に Direct I/O が制御できなくなり，動画に遅延が発生することに注目した．

動画再生に遅延が発生するようになった原因として，分配率が正確に守られなくなった可能性がある．そのため，bfq.io.weight を使用して，fio から発行される I/O のブロックサイズの増加による分配率と動画再生への影響を評価した．また，BFQ の任意に設定可能なパラメータを使用し，動画の遅延や分配率に影響すると考えられる fifo_expire_(a)sync，max_budget，strict_guarantees を使用して，さらに実験した．

3.2 デフォルト設定時の評価

動画再生に遅延を発生させた Direct Read と Direct Write に対して，ブロックサイズを 1MB〜100MB に変化させて，スループットと分配率，動画再生での遅延発生有無を評価した．weight の設定は，動画再生：fio = 2：1 である．分配率の測定では，fio2 つを同時実行させ，fioA：fioB = 2：1 に weight を設定した．評価結果を表 4，表 5 に示す．ブロックサイズが増加するほど，分配率が減少した．35MB を越えた際には分配率が 1.8 を下回り，40MB では動画再生に遅延も発生するようになった．

3.3 BFQ の各パラメータ設定時の評価

Direct Read を使用し，fifo_expire_(a)sync，max_budget，strict_guarantees パラメータを設定して計測した．パラメータの設定値は，スループットへの影響は無視し，動画の遅延をなくすことを優先している．

3.3.1 timeout

fifo_expire_(a)sync は秒単位で設定するため，最小の 1s で設定した．これによって，最大でも 1s 毎にキューの切り替えアルゴリズムが動作することになるため，分配率が改善し，高い weight を設定してある動画再生の応答性が高ま

表 4 デフォルト設定時の Read 評価結果（fioA：fioB = 2：1）

ブロックサイズ [MB]	A の平均スループット [MB/s]	B の平均スループット [MB/s]	分配率	動画の遅延発生有無
1	128	65.3	1.96：1	無
30	152	79.0	1.92：1	無
35	135	77.0	1.75：1	無
40	133	80.9	1.64：1	有
45	128	83.5	1.53：1	有
50	125	91.2	1.37：1	有
100	125	99.2	1.26：1	有

表 5 デフォルト設定時の Write 評価結果（fioA：fioB = 2：1）

ブロックサイズ [MB]	A の平均スループット [MB/s]	B の平均スループット [MB/s]	分配率	動画の遅延発生有無
8	44.6	22.8	1.95：1	無
10	39.1	20.3	1.92：1	無
15	33.9	18.9	1.80：1	有
20	28.5	17.6	1.62：1	有
100	23.0	19.9	1.15：1	有

るはずである．評価環境はデフォルト設定時の評価と同様であり，Direct Read を使用し，ブロックサイズを 1MB〜100MB に変化させて，スループットと分配率，動画再生での遅延発生有無を評価した．

評価結果を表 6 に示す．デフォルト設定時と同様に，ブロックサイズが増加するほど分配率が減少している．しかし，動画再生に遅延が発生するのは 50MB 以上であり，デフォルト設定時よりも大きいブロックサイズまで応答性が確保されるようになった．予想通り，タイムアウトが 1s となることで応答性が向上することが確認できた．しかし，タイムアウトでは budget を使い切る前に切り替え処理に移ることがあるため，分配率の改善には繋がらなかったと考えられる．

3.3.2 max_budget

max_budget のパラメータはセクタ単位である．今回は 1024 セクタで設定した．理由は 2 つあり，1 つ目は blktrace で取得したログから，動画再生や分配率に問題が現れなかった時に処理されている I/O サイズ以下であるため，2 つ目はストレージデバイスにディスパッチされる最大サイズであり，これ以上分割するとスループットが大きく落ちると考えられるためである．これによって，timeout と同様にリクエストキューの切り替え頻度が上がり，分配率の改善と高い weight が設定されている動画再生の応答性向上に繋がると考えられる．評価環境は今まで同様であり，Direct Read を使用し，ブロックサイズを 1MB〜100MB に変化させて，スループットと分配率，動画再生での遅延発生有無を評価した．

評価結果を表 7 に示す．今までと異なり，ブロックサイズが増加しても一定の分配率を保ち，45MB から急激に低下した．また，動画再生での遅延も同じタイミングで発生した．timeout と同様に，切り替え頻度が上がったことでデフォルト設定時と比較すると遅延発生を抑えられたが，分配率は timeout の結果と異なった．

3.3.3 strict_guarantees

strict_guarantees は，ストレージデバイスに対して，同時に 1 つの I/O しか送らないことで，分配率を保証する．評価環境は同様に，Direct Read に対して，ブロックサイズを 1MB〜100MB に変化させて，スループットと分配率，動画再生での遅延発生有無を評価した．

評価結果を表 8 に示す．デフォルト設定時や timeout の時と同様に，ブロックサイズが増加するほど，分配率が減少しているが，timeout の時と比べてさらに大きいブロックサイズでも分配率が正確に守られるようになった．しかし，大きいブロックサイズではスループットも低下していき，100MB では 20%も低下している．一方で，動画再生への遅延は 100MB 未満であっても発生しなかった．timeout や max_budget を低く設定してリクエストキューの切り替え数を増やした時にはスループットに影響がでなかったが，遅延は約 50MB 未満までしか抑えられなかった．strict_guarantees では，遅延を抑えることはできても分配率の解決はできず，さらに全体のスループットが無視できないほど低下することがわかった．

4. 遅延要因の分析

BFQ スケジューラは高い応答性を目的に開発されているが，デフォルト設定時だけでなく，性能調整用のパラメータを使用しても，分配率と動画再生での遅延の問題を解決できなかった．strict_guarantees 使用時の評価は，動画での遅延は解決したが，分配率が改善されないという結果だった．ここから，動画での遅延発生と分配率が崩れるという 2 つの問題は，動画での遅延要因とは異なると考えられる．

2 つの問題の要因を明らかにするために，まずアプリケーションからの I/O が全て発行されているかを確認するため

表 6 fifo_expire_(a)sync = 1s での評価結果

ブロックサイズ [MB]	A の平均スループット [MB/s]	B の平均スループット [MB/s]	分配率	動画の遅延発生有無
1	127	63.4	2.00：1	無
30	136	68.0	2.00：1	無
35	133	79.5	1.67：1	無
40	133	79.6	1.67：1	無
45	124	83.8	1.47：1	無
50	132	95.9	1.38：1	有
100	109	85.7	1.27：1	有

表 7 max_budget = 1024 での評価結果

ブロックサイズ [MB]	A の平均スループット [MB/s]	B の平均スループット [MB/s]	分配率	動画の遅延発生有無
1	128	64.4	1.98：1	無
30	138	69.9	1.97：1	無
35	133	67.2	1.98：1	無
40	135	68.4	1.97：1	無
45	131	67.9	1.93：1	無
50	104	99.6	1.04：1	有
100	122	89.4	1.36：1	有

にシステムコールでの I/O サイズを計測し，次にストレージデバイスに I/O が全て発行されているかを確認するために，ブロックレイヤーでの I/O サイズを計測した．最後に，BFQ アルゴリズムから 2 つの問題の要因となりうるアルゴリズムを調査した．

4.1 システムコールでの I/O サイズ

システムコールの計測には，strace を使用した．動画再生時に使用された read, readv の返り値を合計することで，実際に処理した I/O サイズを計測した．計測の結果，遅延時と非遅延時で 39.95MB となり，変化はなかった．よって，遅延時にアプリケーションからの I/O 自体が欠落していないと考えられる．なお，今回使用した動画のサイズは 41.9MB であるため少し足りないが，これはファイルの再生に不要なデータが読み込まれなかったためだと考えられる．

4.2 ブロックレイヤーでの I/O サイズ

ブロックレイヤーでの計測には，blktrace を使用した．blktrace ではいくつかのトレースポイントがあるが，動画再生プロセスによる I/O がブロックレイヤーに入ってきた地点での I/O サイズを計測した．計測の結果，総量は遅延時と非遅延時で 39〜42MB であり，変化はなかった．ブロックレイヤーではファイル情報が欠落してしまい，動画本体の読み取りのみに限定することができないため，計測した I/O サイズにブレがあったが，大きな遅延発生時も総量が減ることはなく，I/O がストレージに届く前に欠損していないと考えられる．

4.3 BFQ アルゴリズム

4.3.1 budget の動的調整

BFQ は，budget を利用した管理をしている特徴がある．この部分で遅延につながる要因が 2 つ考えられる．

1 つ目は，budget の動的な調整によって I/O 負荷生成プロセスに対して大きな budget が割り当てられ，I/O の割り当て量が公平でなくなっていくことである．budget を使い切ってもリクエストキューに I/O が残っている場合，budget を増やしていくアルゴリズムになっているため，プロセス A に 2 倍の weight が与えられていても，プロセス B が 2 倍の budget になれば同じ量が処理できてしまう可能性があると考えた．

BFQ のソースコードを調査した結果，BFQ では処理量の計算に [service/weight] の単位を増加させるため，budget が増加した場合，この増加量を抑えることができないとわかった．処理量を増やすには，増加量を抑える weight が高いことが必須であるため，動的に調整をした結果，分配率が崩れてしまうことはないことがわかった．しかし，budget が増加すると一度に処理できる量は増えるため，他プロセスに処理が渡るまでの時間が長くなり，動画再生に遅延が発生する原因となる可能性はある．

2 つ目は，1 つの I/O サイズが budget を超える場合である．この場合，表 1 で更新した値を使用した後に budget = max(next io_size, updated budget) で，超過したままのサイズが次の I/O サイズに設定される．これは，キューを選択された時の先頭の I/O が budget を超過している場合でも，I/O を詰まらせないためである．また，ブロックレイヤーではストレージデバイスに送るために 1024 ブロックに分割するが，1 つの I/O はひとかたまりにしたまま扱っていると考えられる．しかし，これも一時的な budget の

Vol.2022-ARC-248 No.31
Vol.2022-SLDM-198 No.31
Vol.2022-EMB-59 No.31
2022/3/10

表 8　strict_guarantees 使用時の評価結果

ブロックサイズ [MB]	A の平均スループット [MB/s]	B の平均スループット [MB/s]	分配率	動画の遅延発生有無
1	121	66.3	1.83：1	無
30	120	63.3	1.90：1	無
35	125	62.9	1.99：1	無
40	110	57.9	1.90：1	無
45	129	68.0	1.90：1	無
50	107	56.5	1.89：1	無
100	84	64.0	1.30：1	無

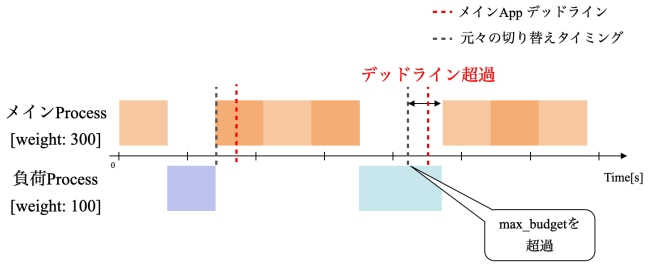

図 5　budget を超過した I/O による動画への遅延

増加であり，選択アルゴリズム用の [service/weight] では実際に実行された量である service で計算されるため，実行された分だけ増加するため，長期的に見た時に分配率に影響はないと考えられる．一方，図 5 に示すように budget を超過したプロセスによって，スケジューラの想定外の実行時間を得ることができてしまい，動画再生用の I/O のデッドラインを超過し，動画再生に遅延を発生させる可能性はあると考えられる．

4.3.2　タイムスタンプの調整

weight の分配率が崩れる要因を明らかにするため，BFQ アルゴリズムにおけるプロセス毎に処理できる I/O 量を調べた．その結果，以下のアルゴリズムが要因ではないかと予想した．

あるプロセスが長時間待ち状態でいると，多量に I/O を処理している他のプロセスによって vtime が増加し，あるプロセスのタイムスタンプが vtime よりはるかに低くなる可能性がある．その結果，待ち状態だったキューは，その後にタイムスタンプの差を埋めるためにデバイスを占有する．これは，idle 状態であるため他プロセスの実行時では I/O 要求が止まっていたのに対して，処理が止まっていた時間分にできたタイムスタンプの差分も I/O が処理できるため，分配率が崩れてしまう．この問題を減らすために，あるプロセスの finish が vtime の現在の値と等しくなるようにタイムスタンプを移動させる．

作成意図としては，idle 状態時に加算された分を計上しないための機能であるが，今回のように大きな I/O によって長時間待ちが発生した場合も機能してしまうと予想される．そのため，finish と vtime の差分だけ分配率が崩れてしまう．ブロックサイズが大きいほど，この差が大きくなり，分配率に影響が崩れていると考えられる．

4.4　分析結果

分析の結果，動画で遅延が発生する問題は，リクエストキューに処理が割り当てられた場合，budget を使い切るまで他のプロセスの処理に移ることができないことが要因で発生していると予想される．分配率が崩れる問題は，長時間占有しないために，大量の I/O が一度に処理された後に，全プロセスのタイムスタンプを揃えるため，実際には処理していない分タイムスタンプが増加することが要因であると予想される．

動画再生での遅延発生は，strict_guarantees を使用することで抑制できた．しかし，第三者アプリケーションによる基本機能となるアプリケーションへの悪影響を防止するという目標のためには，以下の課題が残っている．

- 分配率は崩れたままである．
- ブロックサイズが大きい際の遅延を抑制すると，全プロセスによる合計スループットが低下する．ブロックサイズが 100MB の際は，20%も低下した．
- 動画再生では遅延を抑制できたが，他のアプリケーションでも抑制できるかは保証できない．
- 実行したいアプリケーションが変わるたびに，不具合の発生有無を検証するのは大変である．

他のアプリケーションに対しても，高負荷時に悪影響を受けないことを保証したいため，遅延に加えてこれらの問題にも対処する．

各評価結果から，動画再生が遅延しない時の共通点として，分配率に従った分配ができているという点がある．そのため，動画再生の遅延に加えて，分配率の改善をすることによって，基本機能となるアプリケーションへの悪影響が防止できることを明確に示すことができると考えた．

5.　提案手法

分配率が改善することで，基本機能にあたるプロセスの応答性を確保することを目的に，I/O スケジューラで処理する 1 つの I/O サイズを分割して小さくする．これによって，1 つのプロセスの処理時間が短くなるため，長時間占有されなくなり，分配率が改善するはずである．さらに，strict_guarantees と違い，複数の I/O をストレージデバイスに送信できるため，スループットも低下しないと考えら

Vol.2022-ARC-248 No.31
Vol.2022-SLDM-198 No.31
Vol.2022-EMB-59 No.31
2022/3/10

れる．

5.1 I/O リクエストの事前分割

大きい I/O をどの程度の大きさに分割するかは，分配率が 1.9 以上かつ動画への遅延が発生しない最大の I/O サイズを基準とした．これは，1 つは分割後の動作はその時の動作と同じになると考えられるためであり，もう 1 つは分割数が多いほど分割コストが高くなり，スループットが低下すると予想されるためである．

5.2 fio の変更点

I/O スケジューラで扱う I/O サイズを小さくするには，I/O スケジューラまでのどこかで I/O の分割処理を追加すれば良い．今回は，fio が発行する io_submit のサイズに最大値を設定し，それよりも大きいブロックサイズを発行した時に，分割するようにした．例えば，最大値を 30MB に設定した際にブロックサイズが 100MB の I/O を発行した時は 30+30+30+10 となり，io_submit が 4 回呼ばれる．（実際に追加した分割処理は付録 A.1 参照）

5.3 提案手法の評価

5.3.1 評価方法

データサイズが大きい I/O を分割した結果，動画再生への遅延や分配率が改善したかを既存システムの評価をした時と同じ評価環境で確認した．表 4，表 5 に従い，分割サイズは Read が 30MB，Write が 10MB である．分配率が様々な設定でも忠実に守られるようになったかを確認するために，cgroup io.weight の比率について，既存システムを評価した時の fioA：fioB = 2：1 に加えて，fioA：fioB = 3：2 と fioA：fioB：fioC：fioD = 3：2：2：1 を測定した．

5.3.2 2 プロセスの同時実行時の評価結果

2：1 で評価した結果を図 6，図 7 に示す．bs は，fio が発行した I/O のブロックサイズである．分配率の改善と動画再生への遅延の防止に加えて，スループットの低下も抑えることができており，strict_guarantees の結果（表 8）よりも良い性能を実現できた．

strict_guarantees では，ストレージデバイスに対して同時に 1 つの I/O しか送らないことで，ストレージデバイス内が BFQ からの処理順と同じであることを保証する機能であったが，処理単位の大きい I/O がある時には大きな待ち時間が生じてしまうため，分配率が崩れていた．これを提案手法によって解決したため，提案手法と strict_guarantees を併用することで，eMMC 以外のストレージデバイスを使用した際でも，今回の評価結果と同様の結果が得られると考えられる．

3：2 の指定においても，図 8 から分配率が 2.24：2 から 2.94：2 に改善されたことが分かる．

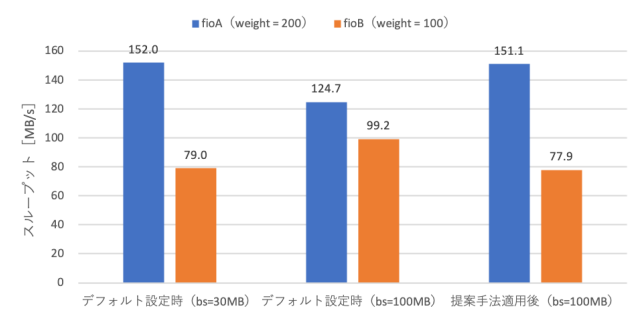

図 6 分割処理追加後の Read 評価結果（fioA：fioB = 2：1）

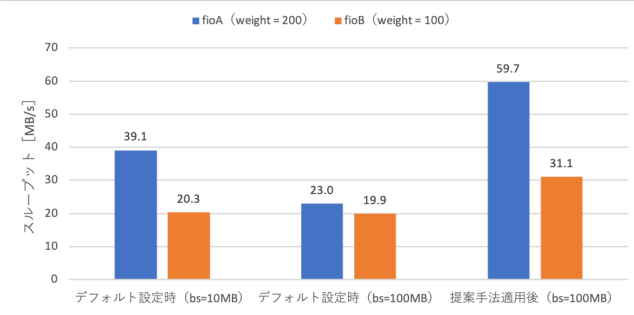

図 7 分割処理追加後の Write 評価結果（fioA：fioB = 2：1）

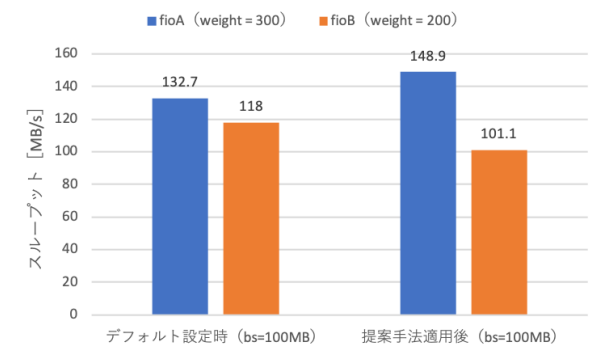

図 8 スループットと分配率の変化（fioA：fioB = 3：2）

5.3.3 多プロセスの同時実行時の評価結果

これまでは，状況を簡単にするために 2 プロセスの同時実行で評価していた．しかし，実際の使用環境では多くのアプリケーションと同時実行されることが予想される．そのため，4 プロセスに異なる weight を指定して同時実行し，その分配率を計測した．設定した weight の比率は fioA：fioB：fioC：fioD = 3：2：2：1 である．デフォルト設定時の結果と比較するために，デフォルト設定時のブロックサイズが 30MB，100MB，分割処理追加後のブロックサイズが 100MB の時を計測した（図 9）．分割後のブロックサイズが 100MB の時の分配率は 2.47：1.85：2.09:1 だった．同時実行するプロセスが増えたことで，30MB の上限では分配率が崩れており，デフォルト設定時で既に 2.43：1.84：1.99：1 だった．これは，分割処理追加後の分配率が分割時に上限に設定した I/O サイズの時の分配率に忠実であることを示している．そのため，使用用途によって満たした

Vol.2022-ARC-248 No.31
Vol.2022-SLDM-198 No.31
Vol.2022-EMB-59 No.31
2022/3/10

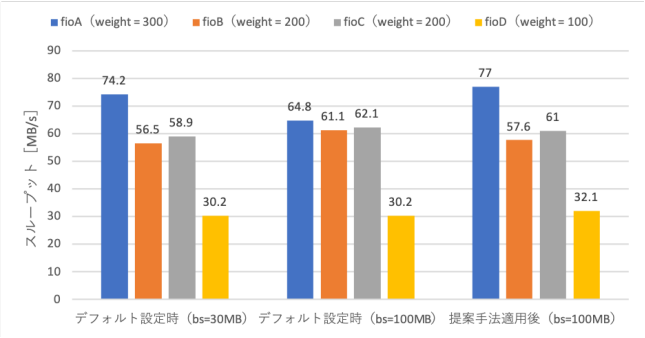

図 9 スループットと分配率の変化（fioA：fioB = 3：2：2：1）

い分配率や動画への遅延状況を測定し，最適な分割サイズを知る必要があるとわかった．

今回の手法は，制約が厳しくなった際でも，より小さい分割値を設定することで分配率に充実に分割できる．しかし，分割しすぎると I/O 分割のコストが大きくなると予想されるため，スループットが低下する可能性がある．分割数は必要最低限とし，スループットと分配率への忠実さ，応答性のバランスをとることが重要である．

6. 関連研究

動画再生と fio の同時実行によって，動画再生に遅延が発生したのは，Buffered Write と，ブロックサイズが大きい時の Direct I/O だった．Buffered Write で正しく分配できない問題に対処している例や，今回対処した大きいブロックサイズでの Direct I/O が組込み環境においては有用になるという例を紹介する．

6.1 Buffered Write の weight ベース割り当て

Buffered Write が cgroups io.weight で制御できない問題に対しては，いくつか対処している研究があるため，2 つ紹介する．

1 つは，cgroup 処理部にプロセスの情報が渡るようにしたものである [10]．cgroup が正しく割り当てられるように，ブロックレイヤーに加えてシステムコールとページキャッシュの 3 レイヤーに一貫したスケジューラを作成している．これによって，ブロックレイヤー内にある cgroup でも I/O とそれを発行したプロセスを知ることができる．結果として，I/O 処理全体の状況を知ることができるため，Buffered Write での分配率の改善だけでなく，Deadline スケジューラでの最大遅延時間も減少している．

もう 1 つは，ページキャッシュの割合を weight に忠実に分配するようにしたものである [11]．先の論文は，Linux カーネルに大きく手を加えているが，この論文での改良箇所はメモリのみである．具体的には，ページキャッシュに対しても cgroup io と同じく weight を設定し，ページキャッシュへの割り当てとページキャッシュからの追い出し両方に weight に従った優先的な処理をしている．ページへの割り当てでは，weight の高いページが先に割り当てられるように待ちキューの順番を入れ替える．ページからの追い出しでは，全体のページ数と weight から各プロセスがページキャッシュに持つページ数を決定し，超過しているプロセスのページから追い出している．これによって，理想値との誤差がまだまだ大きいものの，cgroup io.weight のみの制御よりも，weight に従ったスループットの分配ができるようになっている．

似た課題として書き込み時の journal に注目し，データのみでなく journal も計算に含めることで，より公平に分配できるようにしたものがある．journal も優先度を無視して一つにまとめてからフラッシュするため，処理が遅延するという問題がある [12]．また，カーネルが処理するため，正しい cgroup に割り振れないという問題もある [13]．

これらの問題では，I/O スケジューラからページキャッシュまで範囲を広げることで，Buffered Write や journal を weight 通りに分配できるようにしている．本論文では，これらでは挙げられていない大きいブロックサイズの際の Direct I/O に着目し，発生した問題を分析し，改善する手法を提案した．また，cgroup の改良では，長期的に見た分配率の改善のみに着目しているが，本論文では動画再生の遅延を評価することで，応答性も検証している．

6.2 Direct I/O の有用性

組込みシステムにおける I/O 処理部がボトルネックとなりうるかを検証している [14]．組込みシステムにおいても，高速化したストレージデバイス利用できるようになってきており，Linux ベースのものも増加している．従来ではストレージデバイスの方が速度が遅かったが，組込みシステムのような CPU とメモリの性能が低い場合では，OS 側のページキャッシュ等の処理がボトルネックになりうることを示している．スループットの検証では，ブロックサイズの増加による性能上昇の限界は Direct I/O の方が高かった．また，CPU 使用率も，Direct I/O の方が Buffered I/O よりも余裕があり，その分だけブロックサイズの増加とともにスループットを向上できている．スループット限界値に到達したブロックサイズで，CPU 使用率もほぼ 100%になることが確認できている．

ブロックサイズを大きくするほどスループットが向上するが，どこかがボトルネックとなるため，性能上昇には上限がある．CPU やメモリ性能が高くない組込み環境では，ストレージデバイス側でなく，CPU やメモリがボトルネックとなる場合があるため，Direct I/O を大きいブロックサイズで実行できることは重要である．

7. おわりに

BFQ スケジューラにおいて，100MB のデータサイズが大きい Direct I/O が発行された時に，cgroup io.weight で

指定した比率が崩れた．さらに，基本機能に相当する動画再生プロセスと，第三者アプリケーションに相当する I/O 負荷生成プロセスを同時に実行した際に，動画再生で遅延が発生した．BFQ の strict_guarantees パラメータを使用することで，動画再生の遅延を抑えることはできたが，分配率は崩れたままだった．これらを解決するために，動画再生に遅延が発生した要因と分配率が崩れる要因を調査した．動画再生での遅延は，I/O サイズが大きい場合に，budget 上限を超えて処理できてしまうため，待ち時間が長くなることを制御できないことによると考えられる．分配率が崩れた理由は，停止中のプロセスだと勘違いされることによって，cgroup io.weight のカウント量がリセットされてしまうことが要因だと考えられる．

動画再生での遅延は，strict_guarantees を使用することで抑制できたが，分配率が崩れ，スループットも最大で 20%低下した．基本機能に該当する他アプリケーションに適用した際でも悪影響が及ばないと言うには不十分であり，性能低下も大きかった．これらを解決するため，大きい I/O サイズを，分配率に影響が出なかった I/O サイズに分割する手法を提案した．

第三者アプリケーションを想定した I/O 負荷生成プロセスからの I/O を，分配率に影響が出ない大きさに分割することで，データサイズが大きい I/O に対しても weight に忠実な分配率を実現し，動画再生での遅延発生を抑えることができた．また，I/O の分割処理を追加することによるスループットの低下は見られなかった．しかし今回の手法では，分割する I/O サイズを決定するために，実際に動作させて確認する作業が残っている．

fio のソースコードに I/O サイズを分割するコードを直接追加したが，これでは追加するアプリケーションのソースコードを毎回修正する必要がある．そのため，BFQ の I/O をリクエストキューに挿入する部分に分割処理を追加することで，アプリケーション側の修正をなくしたい．さらに，現在の分配率から，分割する I/O サイズを動的に調整できるようにすることで，同時実行するアプリケーションが変わる度に分割する I/O サイズを調整する手間もなくしたいと考えている．これらによって，あらゆるアプリケーションに対して適用しやすくしたいと考えている．また，分配率を改善することはできたが，分配率が崩れる要因は予想で留まっているため，タイムスタンプを調整する機能の発生頻度や実際のタイムスタンプの動きを確認したい．

参考文献

[1] : New version of BFQ, benchmark suite and experimental results (2014).
[2] 長谷川博紀，松原　豊，加藤寿和，山本　整，高田広章：アプリケーションからのストレージアクセス分析手法，情報処理学会研究報告 (2021).
[3] : block - Linux source code (v5.10.83) - Bootlin, https://elixir.bootlin.com/linux/v5.10.83/source/block. (Accessed on 12/20/2021).
[4] Nguyen, D. T., Zhou, G., Xing, G., Qi, X., Hao, Z., Peng, G. and Yang, Q.: Reducing Smartphone Application Delay through Read/Write Isolation, *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, New York, NY, USA, Association for Computing Machinery, p. 287–300 (online), DOI: 10.1145/2742647.2742661 (2015).
[5] : BFQ (Budget Fair Queueing) — The Linux Kernel documentation, https://www.kernel.org/doc/html/latest/block/bfq-iosched.html. (Accessed on 12/20/2021).
[6] : Control Group v2 — The Linux Kernel documentation, https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html. (Accessed on 12/20/2021).
[7] Oh, K., Park, J. and Eom, Y. I.: H-BFQ: Supporting Multi-Level Hierarchical Cgroup in BFQ Scheduler, *2020 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pp. 366–369 (online), DOI: 10.1109/BigComp48618.2020.00-48 (2020).
[8] : LIVA Z2 (N4000) 64G, https://www.links.co.jp/item/liva-z2-n4000-64g/. (Accessed on 12/20/2021).
[9] : GitHub - axboe/fio: Flexible I/O Tester, https://github.com/axboe/fio. (Accessed on 12/20/2021).
[10] Yang, S., Harter, T., Agrawal, N., Kowsalya, S. S., Krishnamurthy, A., Al-Kiswany, S., Kaushik, R. T., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: Split-Level I/O Scheduling, *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, New York, NY, USA, Association for Computing Machinery, p. 474–489 (online), DOI: 10.1145/2815400.2815421 (2015).
[11] Park, J., Oh, K. and Eom, Y. I.: Towards Application-level I/O Proportionality with a Weight-aware Page Cache Management, *36th International Conference on Massive Storage Systems and Technology (MSST 2020) October 29th and 30th* (2020).
[12] Park, D. and Shin, D.: iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call, *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA, USENIX Association, pp. 787–798 (online), available from ⟨https://www.usenix.org/conference/atc17/technical-sessions/presentation/park⟩ (2017).
[13] 飛松秀三郎，青田直大，Asraa, A. A. M.，河野健二：コンテナ環境におけるジャーナリング I/O の制御，情報処理学会研究報告 (2018).
[14] Joyce, R. and Audsley, N.: Exploring Storage Bottlenecks in Linux-Based Embedded Systems, *SIGBED Rev.*, Vol. 13, No. 1, p. 54–59 (online), DOI: 10.1145/2907972.2907980 (2016).

Vol.2022-ARC-248 No.31
Vol.2022-SLDM-198 No.31
Vol.2022-EMB-59 No.31
2022/3/10

付　　録

A.1　fio の改良後ソースコード

engine/libaio.c（185 行目 fio_libaio_prep 内）

```
if (io_u->ddir == DDIR_READ) {
        MAX_BS_SIZE = 30 * 1024 * 1024; //30MB

        if(io_u->xfer_buflen > MAX_BS_SIZE){
                io_prep_pread(iocb, f->fd, io_u->xfer_buf, MAX_BS_SIZE, io_u->offset);
        } else {
                io_prep_pread(iocb, f->fd, io_u->xfer_buf, io_u->xfer_buflen, io_u->offset);
        }
        if (o->nowait)
                iocb->aio_rw_flags |= RWF_NOWAIT;
} else if (io_u->ddir == DDIR_WRITE) {
        MAX_BS_SIZE = 10 * 1024 * 1024; //10MB

        if(io_u->xfer_buflen > MAX_BS_SIZE){
                io_prep_pwrite(iocb, f->fd, io_u->xfer_buf, MAX_BS_SIZE, io_u->offset);
        } else {
                io_prep_pwrite(iocb, f->fd, io_u->xfer_buf, io_u->xfer_buflen, io_u->offset);
        }
        if (o->nowait)
                iocb->aio_rw_flags |= RWF_NOWAIT;
}
```