情報処理学会論文誌 Vol.61 No.8 1275–1284 (Aug. 2020)

リアルタイムデータ記録のための ファイル書き込みレイテンシ削減方式

岡部 亮[1,a)] 外山 正勝[1]

受付日 2019年11月6日, 採録日 2020年5月12日

概要：Linux を搭載する組込み機器で短周期データのリアルタイム記録を実現する場合，データ発生周期より短い時間で $write()$ システムコールが完了する必要がある．しかし，代表的なファイルシステムの 1 つである Ext4 には，様々な遅延要因が存在する．本研究では，ファイルシステムの $write()$ のレイテンシを，ページキャッシュへの書き込みに必要となる最低限の時間に抑制可能なアプリケーション設計を提案する．提案方式の特徴は次のとおり．(1) ブロックの事前アロケートにより，$write()$ 内のブロック予約処理を抑止する．(2) $write()$ と並行して行うストレージ書き出し処理でメタデータを対象外とすることで，ジャーナリング処理とのロック競合を防ぐ．(3) カーネルの flusher スレッドを停止し，$write()$ 内での flusher スレッドの処理完了待ちを防ぐ．5 msec 周期で発生する 16 KB のデータを最大 100 MB まで記録するユースケースで提案方式を評価した結果，$write()$ の最大レイテンシは 309 μsec であった．提案方式は $write()$ のレイテンシを抑制しており，短周期データの記録に適用可能であることを確認した．

キーワード：リアルタイムシステム，組込みシステム，ジャーナリングファイルシステム，Linux，Ext4

Reducing File Write Latency for Real-Time Data Recording

Ryo Okabe[1,a)] Masakatsu Toyama[1]

Received: November 6, 2019, Accepted: May 12, 2020

Abstract: In order to implement real-time recording of periodic data on embedded systems with Linux, latency of the $write()$ operation on the file system must be shorter than the cycle time. However, there are various latency causes in Ext4, which is one of the most popular Linux file systems. We propose a design of real-time recording that reduces the latency of $write()$ to the minimum amount required for writing to the page cache. Our design is based on the user space so that no kernel modification is necessary. The design considerations are as follows: allocating blocks beforehand to prevent block reservations inside $write()$, disabling the kernel flusher thread, that can block $write()$, and excluding metadata when flushing to the storage to avoid lock contentions between $write()$ and journal committing. According to our measurement, the worst latency of $write()$ on the proposed design is 309 μsec, under a scenario where 16 KB of data are written every 5 msec until reaching the total size of 100 MB. The results show that the proposed design reduces the latency of $write()$, and is effective for real-time recording.

Keywords: real time systems, embedded systems, journaling file system, Linux, Ext4

1. 背景と研究目標

1.1 背景

組込み機器の中には，マイクロ秒からミリ秒の短周期で発生するデータを，SSD や SD カード，eMMC などのフラッシュストレージにリアルタイムに記録することが要求されるものがある．たとえば，制御機器で異常時解析に備えて周期入出力データや演算結果を記録する場合，データの取りこぼしを防ぐためには，短周期データを遅延なく保存できる必要がある．また，工場の稼働状況の分析や設備の故障予測のためのセンサデータの収集，機械学習に活用するための車両の周囲情報の収集などにおいても，短周期

[1] 三菱電機株式会社情報技術総合研究所
Information Technology R&D Center, Mitsubishi Electric Corporation, Kamakura, Kanagawa 247–8501, Japan
[a)] Okabe.Ryo@cb.MitsubishiElectric.co.jp

© 2020 Information Processing Society of Japan

のデータを遅延なく記録することが求められる．

従来，組込み機器はリアルタイム OS を用いて開発されることが一般的であったが，ネットワークや GUI などの要求の高度化にともない，組込み機器への Linux の採用も進んでいる．Linux には，ストレージを利用するための S/W スタックとして，様々な種類のメディアに対応したデバイスドライバや，Ext4 [1]，XFS，F2FS などの異なる特徴を持つ豊富なファイルシステムが用意されている．Linux が提供するこれらのファイルシステムを利用することで，アプリケーションは抽象化されたインタフェースを用いてストレージを操作することができる．また，ジャーナリングやメインメモリを用いたバッファリングなど，信頼性や性能の向上に有効な機構を利用できる．

1.2　課題

組込み機器において Linux のファイルシステムを利用する場合，API の最大レイテンシの保証や実行時間の予測可能性など，リアルタイム性の実現が課題となる．

本研究では，組込み機器向けに広く使用されている Ext4 ファイルシステムを検討対象とする．また，信頼性と性能のバランスから主流となっている ordered モードのジャーナリングを検討対象とする．本モードではファイルのメタデータのみがジャーナリングの対象となる．図 **1** は Ext4 ファイルシステムの構成図である．関連する主要なコンポーネントは，仮想ファイルシステム，Ext4 ファイルシステム，汎用ブロック層，ブロックデバイスドライバである．メインメモリ上のページキャッシュは，ファイルシステムが管理するバッファであり，ファイルデータやメタデータ，ジャーナリングのトランザクションが格納される．ジャーナリング対象であるメタデータは，まず，ストレージ上のジャーナル領域に書き出され（コミット処理），その後，ファイルシステムの正規領域に書き出される（チェックポイント処理）．これによりメタデータのアトミック性が保証され，電源断などによるファイルシステムの破損を回避できる．トランザクションとはコミット処理の単位となる一連のデータの集まりである．Ext4 のジャーナリングには，3 種類のトランザクションがある．メタデータの変更を受け入れ可能である running トランザクション，メタデータの変更の受け入れを停止してコミット処理中である committing トランザクション，および，コミット処理が完了してチェックポイント処理待ちの checkpointing トランザクションである．アプリケーションが *write*() システムコールを実行すると，ファイルデータはページキャッシュに保存される．また，メタデータはページキャッシュに保存されるとともに，running トランザクションに登録される．*write*() システムコールは，通常，ファイルデータとメタデータをページキャッシュに書き込んだ後，すぐにリターンする．ページキャッシュ上のデータは，flusher スレッドと呼ばれるカーネルスレッドが定期的に起動した際や，*fsync*()，*fdatasync*()，*sync_file_range*() のようなストレージ書き出し API をアプリケーションが呼び出した際に，非同期的にストレージに書き出される．

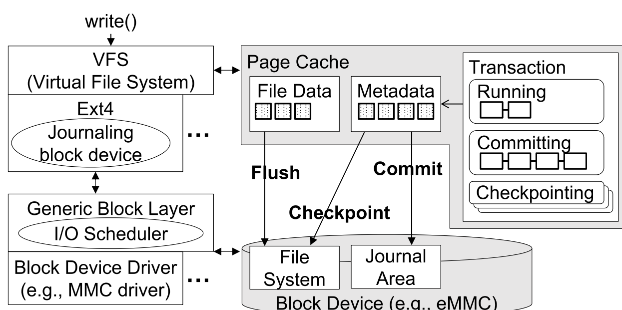

図 **1**　Ext4 ファイルシステムのソフトウェア構成
Fig. 1　Architecture of the Ext4 file system.

Ext4 ファイルシステムには，*write*() システムコールのレイテンシを悪化させる様々な要因が存在する．たとえば，ブロックをアロケートしながらファイル末尾にデータを追加する書き込みである append-write には，ブロックのアロケート予約や，ファイルサイズ増加にともなうメタデータの更新などの遅延要因がある [1]．また，*fsync*() などのストレージ書き出しを行うシステムコールは，ジャーナリングのコミット処理を引き起こし，コミット処理におけるトランザクション操作は，*write*() をはじめとする他のシステムコールとの間で様々なロック競合を引き起こすことが知られている [2]．また，ジャーナル領域の空き領域が一定以下となった場合，*write*() などのシステムコール呼び出しの内部でチェックポイント処理が実行される [2]．そのため，append-write によるメタデータの頻繁な更新や，*fsync*() システムコールによるコミット処理の頻繁な実行により，ジャーナル領域にデータが大量に蓄積すると，*write*() システムコール内部でチェックポイント処理が発生し，レイテンシの増加を引き起こす可能性がある．他に，カーネルの flusher スレッドによる非同期のストレージ書き出し処理は，*write*() システムコールの処理をブロックすることが知られている [3]．

短周期データのリアルタイム記録においてデータの取りこぼしを防ぐためには，データの発生周期よりも短い時間で *write*() システムコールが完了することを保証する必要がある．しかし，*write*() システムコールのレイテンシは，ファイルシステムの各層やストレージ内のファームウェアなどで積み重なり，場合によっては 1,000 msec オーダまで達することが知られている [3], [4], [5]．

1.3　研究目標

本研究の目標は，Ext4 ファイルシステムの *write*() システムコールについて，1.2 節で述べた遅延要因を排除し，ページキャッシュへの書き込みに必要となる最低限のレイ

© 2020 Information Processing Society of Japan

テンシに抑制することである．これにより，ミリ秒以下の短周期データのリアルタイム記録を Linux によって実現することが可能となる．

本研究ではこの目標を，ファイルシステムやその他のカーネルのソースコードを変更せずに，アプリケーションの設計やファイルシステムのパラメータ設定の最適化のみで実現する．カーネルの変更は影響が広範囲に及ぶため，検証工数が増加すること，および，カーネルの独自修正は保守性の低下を招くことが理由である．

想定するユースケースは，制御機器の周期入出力データの記録である．本ユースケースでは，最大記録時間や最大記録長が事前に決まっているものとする．加えて，周期入出力データの記録用にパーティションまたはストレージを占有的に割り当てることを前提とする．記録済みデータの読み出しにはリアルタイム性は要求されないものとする．

2. 関連研究

クラウド環境への適用を想定して，Ext4 の $write()$ のスループット向上やメニーコアに対するスケーラビリティ向上を目指した先行研究がある．たとえば，CPU コア数にスケールするスループットを得るためには，ファイルシステム内部のロック競合が問題であることに着目し，独立したサービス群でファイルシステムを構成することが提案されている [6]．同様にロック競合に着目し，トランザクションのデータ構造をロックフリー化することで，メタデータが頻繁に更新される負荷状況でのスループット性能を向上させる手法も提案されている [2]．

Ext4 における $fsync()$ のレイテンシ削減は，特にスマートフォン向けに活発な研究領域である．Ext4 では異なるファイルに対する変更が単一のトランザクションにまとめられるため，$fsync()$ 内部で行われるコミット処理の実行時間が長くなるという問題がある．そこで，$fsync()$ の実行の際，$fsync()$ 対象ファイルの変更内容のみをトランザクションから抽出してコミットする手法が報告されている [7], [8]．また，$fsync()$ 内でトランザクションをコミットする代わりに，シーケンシャルログとしてストレージに書き出すことで，書き出すデータのサイズを減らして $fsync()$ の応答時間を向上させる手法が提案されている [9]．他に，ファイルシステムのタイムスタンプの粒度が細かい場合，$fsync()$ 内のコミット処理におけるタイムスタンプ更新のデータ量が多くなることに着目し，ファイルシステムのタイムスタンプの粒度を粗くして，コミット処理のデータ量を抑制するという手法が提案されている [10]．ストレージデバイスの観点の研究としては，ジャーナル領域への書き込みの順序を保証するようにストレージデバイスの FTL を改良することで，コミット処理時の FLUSH コマンドの呼び出し回数を削減し，$fsync()$ の応答性を向上させる研究がある [11]．

Ext4 の性能チューニングと評価に関する研究としては，スマートフォンの負荷特性に対して書き込みスループット向上を実現するファイルシステムのパラメータ設定を明らかにしている研究 [12]，ストレージデバイスの低レイテンシ化に対してファイルシステムのスループットのスケーラビリティを評価している研究 [13]，ストレージデバイス上のデータレイアウトと書き込みレイテンシの関係性を明らかにし，ファイルシステムのパラメータ変更によるデータレイアウトへの影響を評価している研究 [14] がある．

POSIX や Linux では，非同期 I/O（AIO）が規定されている [15]．Linux の POSIX AIO はスレッドプールで実現されているため，スレッドの生成や削除，切り替えのオーバヘッドが懸念となる．Linux AIO については，ブロックのアロケート，ストレージ書き出し，ロック競合などによって，I/O 要求処理である $io_submit()$ の実行がブロックされることが問題である [16]．本研究では，同期 I/O である $write()$ のレイテンシ抑制に取り組み，POSIX AIO の適用可否は今後の検討課題とする．

提案方式では，ブロックの一括アロケート後の書き込みに $write()$ を用いる．これに対して，一括アロケートしたブロックを $mmap()$ でマップしてアクセスすることで，レイテンシをさらに削減できる可能性がある．しかし，$mmap()$ 経由のアクセスには，ページテーブル作成やページフォルト処理などの遅延要因がある [17]．ファイル全体に対するページテーブルを $mmap()$ 呼び出しの内部で一括作成することも可能だが，メモリ使用量が問題となる可能性がある．提案方式への $mmap()$ の導入可否は今後の検討課題とする．

Ext4 を改良して $write()$ のレイテンシの削減を目指した先行研究としては，次のものがある．Ext4 で既存ファイルを上書きする際，ストレージからの当該ファイルのフェッチを遅延させて非同期に行うことで，$write()$ の最大レイテンシを削減する研究がある [5]．また，カーネルの flusher スレッドが非同期 I/O を行っている最中は $write()$ のレイテンシが大幅に増加することを受けて，I/O スケジューラの改良が提案されている [3]．本先行研究では，$write()$ の最大レイテンシが 1,813.54 msec から 177.40 msec に改善されている．これに対して本研究は，ファイルシステムに変更を加えず，レイテンシをミリ秒以下に抑制することを目指す．一例として，flusher スレッドを無効化して非同期 I/O を排除する．

短周期データのリアルタイム記録に関しては，次の研究がある．ストリーミング受信における $write()$ のレイテンシの削減を目指す研究 [4] では，ファイルシステム内のブロック管理構造の階層を既存のファイルシステムより少なくすることで，$write()$ 内部のブロック検索処理における HDD のシーク距離を減らし，$write()$ のレイテンシの分散を低減している．また，SD カードを搭載するドライビン

© 2020 Information Processing Society of Japan

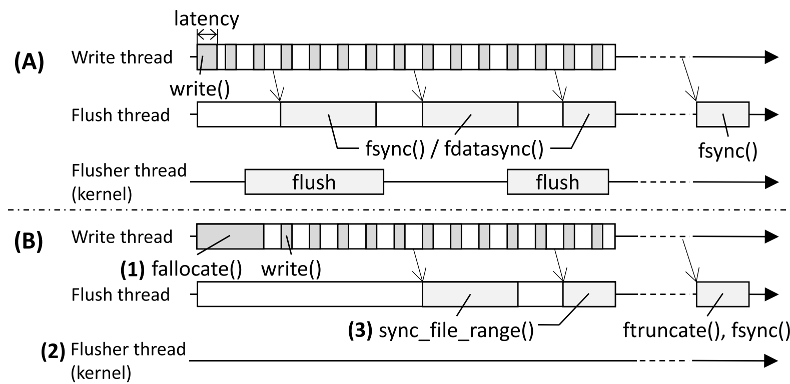

図 2 従来方式 (A) と提案方式 (B) による短周期データのリアルタイム記録の設計
Fig. 2 Design of real-time periodic recording with (A) the conventional method and (B) the proposed method.

グレコーダ向けに，電源断耐性を実現しつつ，断片化の抑制によりレイテンシの削減を実現した FAT ベースのファイルシステムが開発されている [18]．本先行研究は，ページキャッシュを利用せず，ストレージに直接書き込む際のレイテンシの削減を目指すものであり，10 msec オーダの平均レイテンシを実現している．これに対して本研究は，*write*() のレイテンシを，ページキャッシュへの書き込みに必要となる最低限の時間に抑制し，ミリ秒以下のレイテンシの実現を目指す．

以上の先行研究に対して，本研究は，ファイルシステムに変更を加えずに *write*() のレイテンシ抑制を目指す．具体的には，*write*() の処理をブロックさせないためのアプリケーション設計や制約を明確にする．

3. 設計

本章では，短周期データのリアルタイム記録において，*write*() のレイテンシを抑制するための設計を提案する．

3.1 従来方式

図 2 (A) は，従来方式による短周期データのリアルタイム記録の設計である．記録開始時にサイズ 0 の空ファイルを新規に作成し，その後，データの発生周期ごとに当該ファイルにデータを書き込む（append-write）．ストレージへの書き出しは，カーネルの flusher スレッド，および，アプリケーションのバックグラウンドスレッドからの *fsync*()/*fdatasync*() 呼び出しによって行う．フォアグラウンド処理の応答性を確保するために，*fsync*()/*fdatasync*() のようなストレージアクセスが発生する処理をバックグラウンドで実行することは，一般的な手法である [19]．Redis の Append-Only File による永続化 [20] はこの一例である．

基本的に，*write*() はページキャッシュに書き込みを行った後，すぐにリターンするが，1.2 節で述べたとおり，様々な要因でレイテンシが悪化する．たとえば，append-write には，ブロックのアロケート予約や，ファイルサイズ増加にともなうメタデータの更新などの遅延要因がある．また，*fsync*() などのストレージ書き出しを行うシステムコールは，ジャーナリングのコミット処理を引き起こし，コミット処理におけるトランザクション操作は，*write*() をはじめとする他のシステムコールとの間で様々なロック競合を引き起こす場合がある．また，ジャーナル領域の空き領域が一定以下となった場合，*write*() などのシステムコール呼び出しの内部でチェックポイント処理が実行される．そのため，append-write によるメタデータの頻繁な更新や，*fsync*() によるコミット処理の頻繁な実行により，ジャーナル領域にデータが大量に蓄積すると，*write*() の内部でチェックポイント処理が発生し，レイテンシの増加を引き起こす可能性がある．他に，カーネルの flusher スレッドによる非同期のストレージ書き出し処理は，*write*() 処理をブロックする場合がある．

3.2 提案方式

従来方式の問題点を受けて，提案方式の方針を次のとおり設定する．まず，ブロックのアロケート予約や，ファイルサイズ増加にともなうメタデータの更新を抑制する．次に，flusher スレッドによる非同期のストレージ書き出し処理を排除する．さらに，アプリケーションによるストレージ書き出しでは，メタデータが書き出し対象となる *fsync*()/*fdatasync*() を使用しない方式とする．

図 2 (B) および**図 3** に提案方式による設計を示す．本設計の特徴は次のとおりである．(1) *write*() で更新対象となるメタデータである，ファイルサイズとタイムスタンプの更新を抑制する．ファイルサイズについては，ファイルの書き込みの前にブロックを一括アロケートし（pre-allocate），ファイルサイズが書き込み途中で変わらないようにする．一括アロケートにより，*write*() 処理内でのブロック予約処理の抑止も図る．一括アロケートは *fallocate*() システムコールにより行う．記録終了時に未使用のブロックがある場合，*ftruncate*() システムコールにより切り捨てる．タイ

© 2020 Information Processing Society of Japan

```
write_thread() {
  fallocate();
  do {
    if (cyclic timer has expired) {
      Write data to a file with write();
      if (the size of unsynced file-data has exceeded a threshold) {
        Notify the sync_thread of the unsynced file range;
      }
    }
  } until (the end of the recording);
  Notify the sync_thread of the end of the recording;
}

sync_thread() {
  while (true) {
    if (notified of the unsynced file range) {
      Sync the unsynced file range with sync_file_range();
    }
    else if (notified of the end of the recording) {
      ftruncate();
      fsync();
      break;
    }
  }
}
```

図 3 提案方式の疑似コード
Fig. 3 Pseudocode of the proposed method.

ムスタンプについては，タイムスタンプの更新を可能な限りメインメモリ内に保留する．これは，マウントオプション lazytime により実現する．(2) flusher スレッドを停止して，アプリケーションのバックグラウンドスレッドでストレージ書き出しを行う．これにより，非同期のストレージ書き出しの発生を回避する．(3) バックグラウンドスレッドでのストレージ書き出しには，メタデータの書き出しが省略される *sync_file_range*() システムコールを利用する．ファイルに一定量のデータを書き込むごとに本システムコールを呼び出すことで，dirty ページの単調増加を防ぎ，メモリ使用量を一定以下に保つ．一度に書き出すデータ量が多いほど，書き出しスループットは一般的に向上するが，より多くの dirty ページを保持するためメモリ使用量が増加する．省略されたメタデータの書き出しは，記録終了時に *fsync*() を呼び出すことにより行う．*write*() 実行中はメタデータの書き出しを省略することにより，*write*() 処理がコミット処理との間でロック競合を起こすことや，*write*() 呼び出しの内部でチェックポイント処理が実行されることを抑止する．

fallocate()，*ftruncate*()，*fsync*() の実行タイミングは 2 通り考えられる．フォアグラウンドのスレッドで記録処理前後に行う方法と，バックグラウンドのスレッドで記録処理と並行して行う方法である．フォアグラウンドで行う場合，*fallocate*()，*ftruncate*()，*fsync*() の最中は，新しい記録処理を開始することができない．一方，バックグラウンドで行う方法では，新規の記録処理は常に開始可能であるが，バックグラウンド処理がフォアグラウンドの *write*() のレイテンシに影響を及ぼす可能性が考えられる．このレイテンシは評価で確認する．

4. 評価

3 章の提案方式による *write*() のレイテンシの抑制効果を評価した．制御機器で周期入出力データを記録するユースケースを想定し，5 msec 周期で発生する 16 KB のデータを最大 100 MB まで記録する場合の *write*() のレイテンシを測定した．

4.1 評価方法

次に示すとおり，3 章の各設計観点が *write*() のレイテンシの抑制にもたらす効果を，測定により評価した．

ブロックのアロケート方式
ブロックを一括アロケートする方式（W_{PRE}）と，append-write 方式（W_{APPEND}）の測定結果を比較することにより，一括アロケートの効果を評価した．

アプリケーションからのストレージ書き出し方式
fdatasync() によるストレージ書き出し（S_{ALL}）と *sync_file_range*() による書き出し（S_{DATA}）の測定結果を比較し，*sync_file_range*() によるメタデータ書き出し省略の効果を評価した．

flusher スレッドの利用有無
flusher スレッド有効（F_{ON}）と無効（F_{OFF}）の測定結果を比較することで，flusher スレッドの非同期ストレージ書き出しを停止することによる性能改善効果を評価した．

上記の設計観点の組合せとして，以下のテストケースについて測定した．一括アロケートする方式では，ブロックの事前アロケートや切り捨て，および，記録終了時のメタデータ書き出しが *write*() と並行動作する可能性があるため，*write*() 対象ファイルとは別のファイルに *fallocate*() や *ftruncate*() を実行する負荷をバックグラウンドでかけるテストケースを用意した．append-write 方式では，ブロックの一括アロケートや切り捨てを行うことはないため，この負荷は不要である．

NONE W_{APPEND}，S_{ALL}，F_{ON} を組み合わせたテストケースである．図 2 (A) の従来方式に相当する．

ALL W_{PRE}，S_{DATA}，F_{OFF} を組み合わせたテストケースである．図 2 (B) の提案方式に相当する．

ALL-BG$_{TRUNC}$ ALL に対して，別ファイルに *ftruncate*() を実行する負荷をかけた．

ALL-BG$_{ALL}$ ALL に対して，別ファイルに *ftruncate*() と *fallocate*() を実行する負荷をかけた．

W_{APPEND}-S_{DATA}-F_{OFF} 一括アロケートの効果を評価するため，提案方式の一括アロケートを append-write に置き換えたテストケースである．

W_{PRE}-S_{ALL}-F_{OFF} メタデータ書き出し省略の効果を評価するため，提案方式の *sync_file_range*() を *fdatasync*() に置き換えたテストケースである．

© 2020 Information Processing Society of Japan

W$_{PRE}$-S$_{ALL}$-F$_{OFF}$-BG$_{TRUNC}$ W$_{PRE}$-S$_{ALL}$-F$_{OFF}$ に対して，別ファイルに *ftruncate*() を実行する負荷をかけた．

W$_{PRE}$-S$_{ALL}$-F$_{OFF}$-BG$_{ALL}$ W$_{PRE}$-S$_{ALL}$-F$_{OFF}$ に対して，別ファイルに *ftruncate*() と *fallocate*() を実行する負荷をかけた．

W$_{PRE}$-S$_{DATA}$-F$_{ON}$ flusher スレッド停止の効果を評価するため，提案方式において flusher スレッドを有効化したテストケースである．

W$_{PRE}$-S$_{DATA}$-F$_{ON}$-BG$_{TRUNC}$ W$_{PRE}$-S$_{DATA}$-F$_{ON}$ に対して，別ファイルに *ftruncate*() を実行する負荷をかけた．

W$_{PRE}$-S$_{DATA}$-F$_{ON}$-BG$_{ALL}$ W$_{PRE}$-S$_{DATA}$-F$_{ON}$ に対して，別ファイルに *ftruncate*() と *fallocate*() を実行する負荷をかけた．

各テストケースにおいて *write*() のレイテンシを連続 10,000 回測定し，これを 1 セットとした．各テストケースについて 5 セットの測定を行い，最大レイテンシが最も大きかったセットの測定結果を採用した．

さらに，記録処理と並行して別の記録済みデータを読み出すユースケースを想定し，*read*() の性能を評価した．*write*() 対象ファイルとは別に 100 MB のファイルを用意し，2 MB 単位でシーケンシャル読み出しを実行した際のスループット性能を IOzone ベンチマーク [21] により測定した．

4.2 評価環境

動作周波数 1.0 GHz のデュアルコア CPU，1 GB SDRAM，eMMC 4.5 準拠ホストコントローラ，eMMC 5.0 準拠の 8 GB ストレージデバイスを搭載した組込み向け評価ボードを使用した．評価対象の Ext4 ファイルシステムは eMMC 上に構築した．パーティションサイズは 7,067 MB である．OS は Linux 4.4.0 である．加えて，メディアやカーネルバージョンの変更による影響を確認するために，これらを変えた評価も行った．この評価には，UHS スピードクラス 3 の 64 GB microSDXC カード，および，Linux 4.14.78 を使用した．

ファイルシステムに関連するパラメータの設定値は次のとおりである．ジャーナル領域のサイズはデフォルトの 128 MB とした．ジャーナリングのモードは，メタデータのみをジャーナリング対象とする ordered モードを指定した．マウントオプションは noatime，lazytime，delalloc，nodiscard を指定した．noatime はメタデータ atime（参照時刻）の更新を無効化する．lazytime は，メタデータ atime，mtime（更新時刻），ctime（inode 更新時刻）の変更をメインメモリ内に可能な限り保留する．これらの 2 個のオプションによって，ファイル書き込みの際のメタデータの更新を抑制し，*write*() のレイテンシを抑制できることを期待した．マウントオプション delalloc，nodiscard は Ext4 のデフォルト設定である．メモリ管理サブシステムのパラメータでは flusher スレッドの起動条件を設定した．flusher スレッドを無効化するテストケースでは，flusher スレッドが測定中に起動しないように，dirty データの割合が 100%到達時，もしくは，dirty データがページキャッシュに書き込まれてから 3,000 秒経過時に flusher スレッドによるストレージ書き出しが行われるように設定した．flusher スレッドを有効化するテストケースでは，起動条件はデフォルトのままとした．

図 2 の設計に基づき，測定プログラムを次のとおり実装した．*write*() の実行周期は 5 msec，1 回あたりの書き込みサイズは 16 KB とした．ストレージ書き出しは，*write*() の書き込みサイズが 2 MB に到達するごとに行った．なお，本評価で使用した eMMC は，2 MB 単位の書き出しにおけるスループットが平均 12 MB/s であり，データの発生量および発生周期に対して十分な書き出しスループット性能が出ることを事前に確認した．*ftruncate*() のバックグラウンド負荷は，サイズ 100 MB のファイルから 5 msec 間隔で 256 KB のブロックを切り捨てた．*fallocate*() のバックグラウンド負荷は，5 msec 間隔で 100 MB のブロックをアロケートした．

4.3 評価結果

表 1 は，測定結果として，各テストケースにおける *write*() のレイテンシの最小値，最大値，平均値を記載したものである．また，**図 4** は各テストケースにおけるレイテンシの分布を図示したものである．

従来方式（NONE）のレイテンシは平均 259 μsec，最大 4,454 μsec であった．これに対して提案方式（ALL）のレイテンシは平均 176 μsec，最大 309 μsec であり，負荷がない場合は 1 msec 以下のレイテンシを実現可能であることを確認した．一方，負荷をかけた場合，ALL-BG$_{TRUNC}$ の最大レイテンシは約 9.2 msec，ALL-BG$_{ALL}$ の最大レイテンシは約 12.5 msec であり，従来方式（NONE）より最大レイテンシが増加した．

次に，各設計観点によるレイテンシ抑制効果について説明する．負荷がない場合，提案方式（ALL）と比較して，*sync_file_range*() を使用しない W$_{PRE}$-S$_{ALL}$-F$_{OFF}$ は，最大レイテンシが 57%増加した．また，負荷をかけた場合，提案方式（ALL-BG$_{ALL}$）に対して，*sync_file_range*() を使用しない W$_{PRE}$-S$_{ALL}$-F$_{OFF}$-BG$_{ALL}$ は，最大レイテンシが 7.6%増加した．以上より，*sync_file_range*() が最大レイテンシの抑制に効果的であることを確認した．

提案方式（ALL）に対して，flusher スレッドを有効化（W$_{PRE}$-S$_{DATA}$-F$_{ON}$）すると，最大レイテンシが 17%増加した．また，負荷をかけた場合，提案方式（ALL-BG$_{ALL}$）と比較して，flusher スレッドを有効化（W$_{PRE}$-S$_{DATA}$-F$_{ON}$-

© 2020 Information Processing Society of Japan

表 1 各テストケースにおける *write*() システムコールのレイテンシ測定結果（単位：μsec）

Table 1 Measurement results of *write*() latency for different test cases. The time unit is microsecond.

	NONE	ALL	ALL-BG$_{TRUNC}$	ALL-BG$_{ALL}$	W$_{APPEND}$-S$_{DATA}$-F$_{OFF}$	W$_{APPEND}$-S$_{ALL}$-F$_{OFF}$	W$_{PRE}$-S$_{ALL}$-F$_{OFF}$	W$_{PRE}$-S$_{ALL}$-F$_{OFF}$-BG$_{TRUNC}$	W$_{PRE}$-S$_{ALL}$-F$_{OFF}$-BG$_{ALL}$	W$_{PRE}$-S$_{DATA}$-F$_{ON}$	W$_{PRE}$-S$_{DATA}$-F$_{ON}$-BG$_{TRUNC}$	W$_{PRE}$-S$_{DATA}$-F$_{ON}$-BG$_{ALL}$
Min	141	89	88	88	142	140	89	90	87	90	88	88
Max	4,454	309	9,178	12,482	107,546	4,821	486	9,334	13,431	361	13,273	121,756
Ave	259	176	176	189	298	275	173	182	178	178	177	201

表 2 メディアおよびカーネルバージョン変更時の *write*() システムコールのレイテンシ測定結果（単位：μsec）

Table 2 Measured results of *write*() latency with different media and kernel versions. The time unit is microsecond.

	eMMC				microSDXC			
	Linux 4.4		Linux 4.14		Linux 4.4		Linux 4.14	
	NONE	ALL	NONE	ALL	NONE	ALL	NONE	ALL
Min	141	89	143	92	139	89	141	91
Max	4,454	309	7,056	683	11,065	384	12,401	685
Ave	259	176	514	387	296	210	536	384

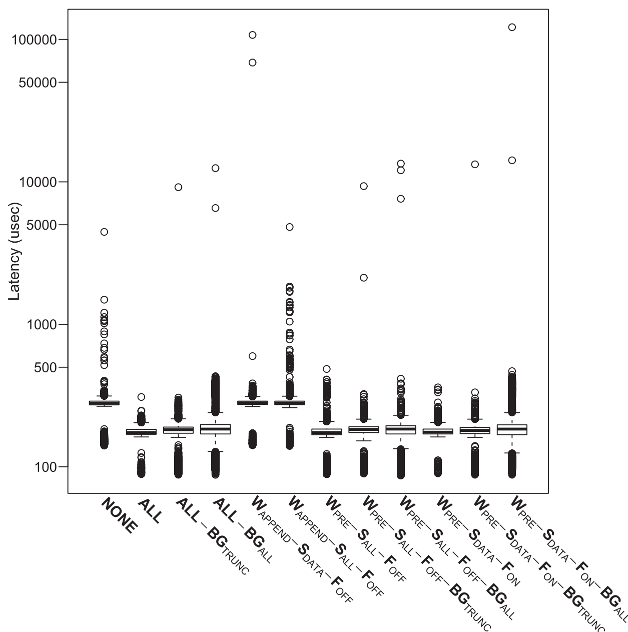

図 4 *write*() システムコールのレイテンシ測定結果

Fig. 4 Box plot of measured *write*() latency.

BG$_{ALL}$）すると，最大レイテンシは 9.8 倍まで増加した．以上より，flusher スレッドの無効化は，負荷をかけた場合に特にレイテンシ抑制効果が大きいことが分かった．

提案方式（ALL）で一括アロケートを append-write に置き換えると（W$_{APPEND}$-S$_{DATA}$-F$_{OFF}$），最大レイテンシが 100 msec を超えた．したがって，提案方式において一括アロケートは最大レイテンシの抑制に貢献しているといえる．なお，W$_{APPEND}$-S$_{DATA}$-F$_{OFF}$ において *sync_file_range*() を *fdatasync*() に置き換えた場合（W$_{APPEND}$-S$_{ALL}$-F$_{OFF}$）は最大レイテンシが約 4.8 msec まで減少したことから，append-write 方式では *sync_file_range*() が逆効果であることが分かった．

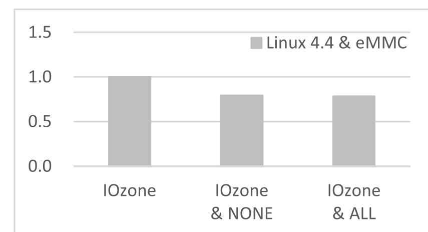

図 5 従来方式および提案方式のバックグラウンドでシーケンシャル読み出しを実行した際のスループット性能（IOzone 単独実行時の測定結果で正規化）

Fig. 5 Sequential-read throughput in the background of the conventional and the proposed method. The results are normalized to the performance of running IOzone alone.

表 2 は，メディアとカーネルバージョンを変更した場合における，従来方式と提案方式の *write*() のレイテンシの測定結果である．提案方式は，いずれの場合でも最大レイテンシを 1 msec 以下に抑制しており，メディアやカーネルバージョンの変更に対してある程度の耐性があるといえる．

図 5 は，従来方式および提案方式のバックグラウンドでシーケンシャル読み出しを実行した際のスループット性能である．IOzone 単独実行時の性能と比較して，従来方式のバックグラウンドでは 79%，提案方式のバックグラウンドでは 78%に低下した．提案方式によるシーケンシャル読み出し処理の性能低下は，従来方式と同等であるといえる．

表 3 は，バックグラウンドのシーケンシャル読み出し処

© 2020 Information Processing Society of Japan

表 3　バックグラウンドでシーケンシャル読み出しを実行した際の *write*() のレイテンシ測定結果（単位：μsec）

Table 3　Measurement results of *write*() latency under the sequential-read workload. The time unit is microsecond.

	Linux 4.4 & eMMC			
	w/o *read*()		w/ *read*()	
	NONE	ALL	NONE	ALL
Min	141	89	142	88
Max	4,454	309	355,958	418
Ave	259	176	392	179

理の有無で，*write*() のレイテンシを比較したものである．バックグラウンドで読み出し処理を実行した場合，従来方式の最大レイテンシが 80 倍に増加したのに対して，提案方式の最大レイテンシは 35%の増加にとどまった．

4.4　考察

まず，提案方式による短周期データのリアルタイム記録の実現可能性について考察する．提案方式は最大レイテンシが 309 μsec であり，ミリ秒以下の周期データのリアルタイム記録に利用可能であるといえる．一方で，別ファイルに対するブロックの一括アロケートや切り捨て，メタデータ書き出しを *write*() と並行して行うと，10 msec を超える最大レイテンシが見られた．これは，進行中の記録処理と並行して，その直前の記録処理で生成されたファイルのブロック切り捨てを行ったり，次の記録処理に備えて一括アロケートを行ったりすると，進行中の記録処理の *write*() に許容できない最大レイテンシが発生することを意味する．したがって，ブロックの一括アロケートや切り捨てを実行するための空き時間を，記録処理の合間に確保できることが，提案方式を適用するための前提条件となる．

次に，提案方式に負荷をかけた ALL-BG$_{\mathrm{TRUNC}}$，および，ALL-BG$_{\mathrm{ALL}}$ のレイテンシ増加原因を考察する．提案方式では，一括アロケートと lazytime によって，ファイルサイズとタイムスタンプのメタデータ更新を抑制し，*write*() 処理とコミット処理の間のロック競合を抑止できることを期待していた．一方で調査の結果，*write*() 処理は，ファイルサイズの変更が不要にもかかわらず，ファイルサイズの増加に備えてトランザクションのハンドルを取得しようとしていた．この際，コミット処理のために running トランザクションが locked 状態であったため，*write*() 処理は当該トランザクションのハンドルを取得できず，ブロックされていた．改善案として，一括アロケートによりファイルサイズが変わらないことを保証できる場合は，トランザクションのハンドルの取得を省略するようにファイルシステムを改修することが考えられる．

また，従来方式（NONE）の最大レイテンシの回避は次の理由で困難である．最大レイテンシの原因は，*write*() 処理とストレージ書き出し処理の間のセマフォ（i_data_sem）の競合であった．レイテンシが増加したケースでは，ストレージ書き出し処理がセマフォを取得したまま eMMC のリードを行っていた．リードの目的は，ブロックグループディスクリプタの参照により，ブロックの空き情報を取得することである．これに対して，事前にすべてのブロックグループディスクリプタをリードし，キャッシュを構築しておくことが考えられる．しかし，ブロックグループディスクリプタのキャッシュはページキャッシュに構築されるため，将来的に追い出される可能性があり，キャッシュに常時存在することを保証できない．

5.　考察

最初に，提案方式を適用する際の制約条件について考察する．評価の結果，一括アロケートやブロック切り捨てのための空き時間を，記録処理の合間に確保可能でなければならないという制約が判明した．加えて，研究目標で述べたとおり，最大記録時間や最大記録長が事前に決まっていること，および，パーティションまたはストレージを占有的に割り当てることを前提としている．

次に，カーネルをアップデートすることによる影響について考える．一般的に，制御機器におけるカーネルアップデートの動機は，機能追加ではなく，バグ修正やセキュリティ修正である．この場合のアップデートは，カーネルバージョンを固定して修正を取り込むことにより行われるため，提案方式に対する影響は小さいと考える．一方で，機能追加の要求を考慮する必要がある情報系機器では，カーネルバージョンの変更を想定する必要がある．この場合にはレイテンシの再評価が必要になる．ただし，Ext4 は成熟したファイルシステムであるため，カーネルバージョンの変更が提案方式に及ぼす影響は大きくないと考える．実際に，今回の評価では，4.4 と 4.14 という 2 つのカーネルバージョンにおいて提案方式が有効であることを確認した．

最後に，提案方式のページキャッシュ使用量と，その他のプロセスのメモリ使用量の関係について考察する．他プロセスのメモリ使用量が増加した場合，リアルタイム記録用のページキャッシュを獲得できなくなる可能性や，リアルタイム記録で使用中のページキャッシュが回収されてしまう可能性がある．提案方式では，*write*() の実行時に，その書き込みサイズに応じてページキャッシュが割り当てられる．*write*() により蓄積した dirty ページは，*sync_file_range*() で定期的に書き出されることで回収対象となる．ゆえに，提案方式のページキャッシュ使用量は常に一定以下であることが保証される．以上より，各プロセスのメモリ使用量を設計する際に，提案方式のページキャッシュ最大使用量の分を予約しておくことで，リアルタイム記録用のページキャッシュに対する影響を防ぐことができる．

© 2020 Information Processing Society of Japan

6. まとめ

Linuxを搭載する組込み機器において，短周期データのリアルタイム記録を実現する場合，データの発生周期よりも短い時間で*write*() システムコールが完了することを保証する必要がある．しかし，Linuxの代表的なファイルシステムの1つであるExt4には，様々な*write*() のレイテンシ悪化要因が存在する．

本研究では，ファイルシステムの*write*() システムコールのレイテンシを，ページキャッシュへの書き込みに必要となる最低限の時間に抑制可能なアプリケーション設計，および，ファイルシステムのパラメータ設定を提案した．提案方式の特徴は次のとおり．(1) ブロックの一括アロケートにより，*write*() 処理におけるブロックの予約処理を抑止する．(2) カーネルのflusherスレッドを停止することで，*write*() 内部でのflusherスレッドの処理完了待ちを防ぐ．(3) ストレージ書き出し処理で，メタデータを対象外とする*sync_file_range*() システムコールを利用することで，*write*() 処理とコミット処理との間のロック競合や，*write*() 内部でのチェックポイント処理の起動を抑止する．

5 msec周期で発生する16 KBのデータを最大100 MBまで記録するユースケースで提案方式を評価した結果，*write*() の最大レイテンシは309 μsecであった．提案方式は*write*() のレイテンシを抑制しており，短周期データのリアルタイム記録に利用可能であることを確認した．一方で，一括アロケートやブロック切り捨てと並行して*write*() を実行した場合，10 msecを超えるレイテンシが発生した．したがって，提案方式を適用する際の制約条件として，一括アロケートやブロック切り捨てのための空き時間を，記録処理の合間に確保する必要があることが分かった．

Linuxは，Linus Torvalds氏の日本およびその他の国における登録商標または商標です．その他，本稿に記載の製品名などは，各社の日本およびその他の国における登録商標または商標です．

参考文献

[1] Kumar, A., Cao, M., Santos, J. and Dilger, A.: Ext4 block and inode allocator improvements, *Proc. Ottawa Linux Symposium 2008*, pp.263–274 (2008).

[2] Son, Y., Kim, S., Yeom, H. and Han, H.: High-Performance Transaction Processing in Journaling File Systems, *Proc. 16th USENIX Conference on File and Storage Technologies* (*FAST '18*), pp.227–240 (2018).

[3] Jeong, D., Lee, Y. and Kim, J.: Boosting Quasi-Asynchronous I/O for Better Responsiveness in Mobile Devices, *Proc. 13th USENIX Conference on File and Storage Technologies* (*FAST '15*), pp.191–202 (2015).

[4] Won, Y., Kim, D., Park, J. and Lee, S.: HERMES: Embedded file system design for A/V application, *Multimedia Tools and Applications*, Vol.39, No.1, pp.73–100 (2008).

[5] Campello, D., Lopez, H., Useche, L., Koller, R. and Rangaswami, R.: Non-blocking Writes to Files, *Proc. 13th USENIX Conference on File and Storage Technologies* (*FAST '15*), pp.151–165 (2015).

[6] Kang, J., Zhang, B., Wo, T., Yu, W., Du, L., Ma, S. and Huai, J.: SpanFS: A Scalable File System on Fast Storage Devices, *Proc. 2015 USENIX Annual Technical Conference* (*ATC '15*), pp.249–261 (2015).

[7] Kang, Y. and Shin, D.: Per-Block-Group Journaling for Improving Fsync Response Time, *Proc. 18th IEEE International Symposium on Consumer Electronics* (*ISCE 2014*), pp.1–2 (2014).

[8] Park, D. and Shin, D.: iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call, *Proc. 2017 USENIX Annual Technical Conference* (*ATC '17*), pp.787–798 (2017).

[9] Chang, L., Sung, P., Chen, P. and Chen, P.: Eager Synching: A Selective Logging Strategy for Fast fsync() on Flash-Based Android Devices, *ACM Trans. Embedded Computing Systems* (*TECS*), Vol.16, No.2 (2017).

[10] Son, H., Lee, S., Choi, G. and Won, Y.: Coarse-grained mtime Update for Better fsync() Performance, *Proc. Symposium on Applied Computing* (*SAC '17*), pp.1534–1541 (2017).

[11] Park, D., Kang, D. and Eom, Y.: OFTL: Ordering-aware FTL for Maximizing Performance of the Journaling File System, *Proc. 55th Annual Design Automation Conference* (*DAC '18*) (2018).

[12] Kim, H. and Kim, J.: Tuning the Ext4 Filesystem Performance for Android-Based Smartphones, *Frontiers in Computer Education*, Sambath, S. and Zhu, E. (Eds.), Springer Publishing Company, pp.745–752 (2012).

[13] Santana, R., Rangaswami, R., Tarasov, V. and Hildebrand, D.: A Fast and Slippery Slope for File Systems, *ACM SIGOPS Operating Systems Review - Special Topics*, Vol.49, No.2, pp.27–34 (2016).

[14] He, J., Nguyen, D., Arpaci-Dusseau, C.A. and Arpaci-Dusseau, H.R.: Reducing File System Tail Latencies with Chopper, *Proc. 13th USENIX Conference on File and Storage Technologies* (*FAST '15*), pp.119–133 (2015).

[15] Seppanen, E., O'Keefe, T.M. and Lilja, J.D.: High performance solid state storage under Linux, *Proc. 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies* (*MSST 2010*), pp.1–12 (2010).

[16] Linux man-pages project: io_submit(2) - Linux manual page, Linux man pages online (online), available from ⟨http://man7.org/linux/man-pages/man2/io_submit.2.html⟩ (accessed 2019-10-24).

[17] Choi, J., Kim, J. and Han, H.: Efficient Memory Mapped File I/O for In-Memory File Systems, *Proc. 9th USENIX Workshop on Hot Topics in Storage and File Systems* (*HotStorage '17*), p.5 (2017).

[18] Kim, Y. and Shin, D.: Improving File System Performance and Reliability of Car Digital Video Recorders, *IEEE Trans. Consumer Electronics*, Vol.61, No.2, pp.222–229 (2015).

[19] Kim, S., Kim, H., Lee, J. and Jeong, J.: Enlightening the I/O Path: A Holistic Approach for Application Performance, *Proc. 15th USENIX Conference on File and Storage Technologies* (*FAST '17*), pp.345–358 (2017).

[20] Redis project: Redis Persistence, Redis official website (online), available from ⟨https://redis.io/topics/persistence⟩ (accessed 2020-02-10).

[21] IOzone project: IOzone Filesystem Benchmark, IOzone official website (online), available from ⟨http://www.iozone.org/⟩ (accessed 2020-02-11).

© 2020 Information Processing Society of Japan

岡部 亮

2006 年早稲田大学理工学部情報学科卒業．2008 年同大学大学院理工学研究科情報・ネットワーク専攻博士前期課程修了．同年三菱電機株式会社入社．組込みシステム向けソフトウェアプラットフォームの研究開発に従事．

外山 正勝 （正会員）

2000 年立命館大学大学院理工学研究科情報システム学専攻修士課程修了．同年三菱電機株式会社入社．組込みシステムのプラットフォーム，アーキテクチャ，システム設計に関する研究開発に従事．

© 2020 Information Processing Society of Japan