Vol.2016-OS-136 No.11
2016/3/1

ログの自動挿入によるスレッド間のデータ依存関係の追跡

窪田 貴文[†1]　吉村 剛[†1]　河野 健二[†1]

概要：ログは実行エラー時のシステムの挙動を理解するのに有効な情報源である．これはログの出力にはデバッグに必要な情報が含まれていたり，また実行パスの推定にも有効だからである．しかしながら，近年のマルチスレッド環境に対応したシステムソフトウェアでは単純なログのみではシステムの挙動を理解するのは難しい．なぜなら，マルチスレッド環境では複数スレッドで共有されるデータによりデータフローがスレッドをまたぎ，スレッド間で依存関係が発生するからで ある．さらに，このような複雑なデータフローを追跡するログを手動で挿入するのは，ソフトウェアの専門的な知識と開発コストが必要な作業である．そこで，本研究ではスレッド間の依存関係を追跡するログの自動挿入を行なうツールを提案する．提案手法では，型情報を考慮したデータフロー解析を行なうことでスレッドをまたいで依存関係が発生するデータフローのペアを特定し，そのデータフローを追跡するログの自動挿入を行なう．本研究では提案手法を Linux I/O に適用し，実際に Btrfs におけるバグのデバッグにおいて有用性が示している．また，提案手法におけるオーバヘッドはデータベースアプリケーションにおいて約 2 %であった．

1. はじめに

近年のシステムソフトウェアでは, たとえオペレーティングシステムであったとしても, バグが多く存在する [7], [12]. しかしながら，デバッグや原因の特定には時間がかかり大変なものである．さらには，異常時のシステムの挙動はより複雑なものになってきている．例えば，マルチスレッド環境ではスレッド間でデータを共有することがよくあり，その共有データのためスレッド間でデータ依存関係が発生する．そして共有データへのアクセスはそのデータを共有しているスレッドの挙動に影響を与える．このことによって，あるプロセスが共有データを破壊することによって，別プロセスで異常が発覚することがある．このようなスレッドを超えるエラー伝搬は，異常の原因追求をより困難にしてしまう．

このような時間のかかる専門的なプロセスを円滑にすすめるには，ログが一般的で有効的である．ログは実行時の重要な情報を記録し，様々な解析に用いられている [3], [8], [16], [18], [22]．スレッド間のデータ依存関係の追跡においても，ログが重要な役割を果たしている．例えば, 分散システムにおけるログ解析では, ログのリクエストの ID から通信している２つのスレッドを関連付けている．

このようにログの重要性が認知されている一方で，ロギングには情報量とオーバヘッドのトレードオフが存在する [4], [19], [20], [23]．ログが少なすぎると必要な情報が集まらない可能性があり，多すぎるとオーバヘッドが大きくなってしまう (スループットの減少，CPU 使用率の増加)．

このトレードオフを解決するために，開発者はどこにどのようなログをとるのか専門的な決定をする必要がある．しかしながら，近年の調査 [4] では開発者のあいだで，どこにどのようなログをとるのか認識を一致させるのはマイクロソフトのような一流の会社においても難しい．結局，その過程には専門的な知識が必要であり，新たな開発者とって困難なタスクである．

いくつかの研究 [20], [23] ではこの問題を解決しようとしているが，残念ながらスレッド間のデータ依存関係の追跡は対象外である．しかしながら．依存関係の追跡は近年のソフトウェアのデバッグや解析において重要である．ここで，本研究では Linux I/O においてライトバックキャッシュと I/O スケジューリングの２つのデータ依存関係について考察した．そして，この考察をもとに K9 という２つの依存関係を追跡するログを自動で挿入するツールを提案する．K9 は依存関係を起こしるデータフローの組を特定し，それをトレースするログを自動挿入する．

本研究では，K9 の有用性を実際に報告された btrfs と CFQ のバグにおいて K9 ログの用いて解析において評価している．また，いくつかのワークロードでオーバヘッドの測定をおこなった．K9 は Ftrace を用いてログをメモリに出力している．その結果，２つのデータベースアプリケーションにおいてオーバヘッドが小さい (最大 2.12%).

[†1] 現在，慶應義塾大学大学
Presently with Keio Uniersity

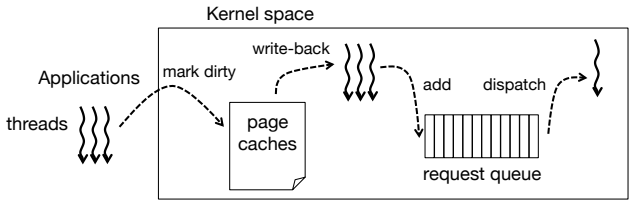


図 1 Linux block I/O における write のワークフロー

また，より深く K9 のオーバヘッドを測定するために，I/O が発生しないワークロードにおいてもオーバヘッドを測定した．出力されたログをプロファイリングし，オーバヘッドを削減するにあたって開発者が陥りやすい間違った手法を発見した．本研究では，最適化するさいの効率的な手法を共有する．

残りの構成は次のようになっている．2 章ではスレッド間依存関係の紹介とそれによってデバッグ難しくなることを示す．3 章では K9 の詳細な設計について述べる．4 章で K9 の有用性とオーバヘッドの評価を行う．最後に，5 章で関連研究を 6 章で結論を述べる．

2. 研究動機とスレッド間での依存関係

非同期処理は I/O などのブロック処理を延期，バッチ処理を行なうことでソフトウェアの性能を劇的に向上させる．しかしながら，非同期処理によって近年ソフトウェアのデバッグが複雑になってきている．本研究では，K9 の必要性を示すためディスク I/O における非同期処理が原因で存在した問題を示す．図 1 は Linux におけるブロック I/O の処理の簡略図である．図から分かる通り，ある 1 つの I/O 処理において，しばしばページキャッシュや I/O リクエストキューを通して複数スレッドをまたいで処理が行われる．

非同期処理におけるデバッグにおいて主要な課題はスレッド間でのデータ依存関係である．例えば，Linux ではページキャッシュによるクラッシュの原因を特定するのが難しい．ページキャッシュによって I/O 処理がプールされ 2 つのスレッド間でデータの依存関係が発生するからである．

また，スレッド間でのデータ依存関係を追跡するには重要なデータ構造体に関する知識も必要である．しかしながら，開発者が重要なデータ構造を把握していたとしても，依然として適切にログポイントとその出力内容を決定する必要がある．よって非同期処理を追跡するのに手動でのログ挿入は非常に時間のかかるものであり，現実的ではない．

2.1 ページキャッシュ

まず，ライトバックキャッシュによって生じるスレッド間での依存関係について示す．ライトバックキャッシュはライト次のパフォーマンスを向上させる一般的な手法であり，即座にディスクへのデータを更新を行わない．ライトバックキャッシュにおいて，まずプロセスはページキャッシュへと書き込みを行なう (例 ext4_da_write_begin())．そして，実際の I/O はバッファリングして後に行なうようにする．ページの変更内容の同期を行なうとき，カーネルスレッド (例 pdflush, kworker) によってダーティバッファが処理される (例 __bio_add_page() in ext4_writepages())．こうして I/O 処理がカーネルスレッドに委譲されることによって，図 2 のようなカーネルスレッドと別のスレッド間でページキャッシュを通してデータの依存関係が生じる．

```
#1   static int ext4_da_write_begin(...) {
#2     // struct address_space* mapping, struct page* page
#3     page = grab_cache_page_write_begin(mapping, ...);
#4     ...
#5   }                          Dependency from ext4_da_write_begin()
#6                              to __bio_add_page() via the dirty page
#7   static int ext4_writepages(...) {
#8     ...
#9     static int __bio_add_page(...) {
#10      // struct bvec* bvec, struct page*page,
#11      bvec = &bio->bi_io_vec[...];
#12      bvec->bv_page = page;
#13      bvec->len = len; //unsigned int len
#14      ...
#15    }
#16    ...
#17  }
```

図 2 ライトバックキャッシュにおけるスレッド間のデータ依存関係

```
#1   static noinline_for_stack int write_one_eb(...) {
#2     ...
#3     for (i = 0; i < num_pages; i++) {
#3       struct page *p = extent_buffer_page(eb, i);
#4       clear_page_dirty_for_io(p); // clear the p's dirty flag
#5       ret = submit_extent_page(p); // do I/O
#6       if (ret) {
#7         end_page_writeback(p);
#8         break;
#9       }
#10    }
#11    ...
#12    if(unlikely(ret)) { // error handling
#13      for (; i < num_pages; i++) {
#14        struct page *p = extent_buffer_page(eb, i);
#15        + clear_page_dirty_for_io(p); // Fix
#16        unlock_page(p);
#17      }
#17    }
#18    ...
#19  }
```

図 3 Linux kernel v3.17-rc5 における btrfs のバグとコード修正

ここで，btrfs ファイルシステム [9] で報告されたライトバックキャッシュ時のページキャッシュ管理でのバグについて議論する．図 3 はバグに対するコード修正を示している．submit_extent_page() が失敗した時に，write_one_eb() は clear_page_dirty_for_io() と submit_extent_page() を呼ぶのを中断してしまう．これによって，btrfs_release_extent_buffer_page() ないでカーネルパニックを引き起こしてしまう．

Btrfs はページのライトバックを行う前に，ページ解放に備えてページのダーティフラグを落ちしてからライトバックシなければいけない．これはページ解放時に解放されるページのダーティフラグは落ちていなければいけないという制約があるからである．また，フォールトトレランスを考慮して，OOM 時のメモリ割り当て失敗のような一時的

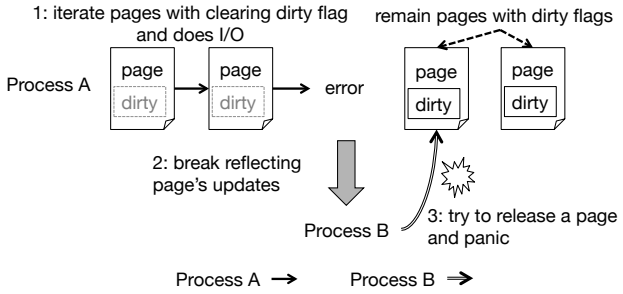


図 4 Simplified thread's behaviors in the btrfs failure

```
#1   static void noop_add_request (...) {
#2     // struct request_queue *q, struct request* req
#3     struct noop_data *nd = q->elevator->elevator_data;
#4     list_add_tail(&rq->queuelist, &nd->queue);
#5     ...
#6   }
#7
#8   static int noop_dispatch(...) {
#9     // struct request_queue *q, struct request* req
#10    struct noop_data *nd = q->elevator->elevator_data;
#11    ...
#12    rq = list_entry(nd->queue.next, ...);
#13    ...
#14  }
```

Dependency from noop_add_request() to noop_dispatch() via the I/O request and queue

図 5 Inter-thread data dependency in I/O scheduling; the dependency between the noop_add_request() and noop_dispatch() through the I/O request and queue

なエラーでは，btrfs はライトバック動作を別プロセスに委任する．このとき，ページのダーティフラグは落とした状態で委任しなければいけない．しかしながら，btrfs はエラーハンドリング時にフラグ操作をし忘れていたため，ダーティフラグが残ったままのページが存在してしまう．この結果，別プロセスによるページ解放時にカーネルパニックを引き起こしてしまう．図 4 はパニック時のスレッドの動作と依存関係を簡単に示している．

一般的なログではこのバグのデバッグは難しい．なぜなら，マルチスレッドを意識した設計のログになっていないからである．例えば，Linux kernel のエラーログである oops メッセージ [1] はどのページによってどの関数でパニックなったことしかわからない．これでは，どうしてそのページがパニックを引き起こしたのかがわからない．また．Linux kernel のトレーサである Ftrace [13] もまた細かなページ操作に関するロギングを対応していない. つまり，より高度なロギングがこのデバッグには必要になってくる．

2.2 I/O Queues

もう一つのデータ依存関係は I/O スケジューリングによって生じる．I/O スケジューリングはオペレーティングシステムのストレージ関連において主要な要素の１つである．I/O スケジューラはその多様なスケジュールポリシーにかかわらず，共通のデータ構造体を持っている: I/O リクエストとリクエストキュー．一般的に，I/O スケジューラは I/O リクエストの挿入 (e.g, noop_add_request()) と発行 (e.g., noop_dispatch()) を行なう．それぞれのスケジューリングポリシに応じて，挿入するスレッドと発行するスレッドが異なる場合がある．ライトバックキャッシュと同様に，このとき図 5 のようなスレッド間でのデータ依存関係が発生する．

ここで スレッド間の依存関係を把握できないため，I/O スケジューラが正しくスケジューリングできない問題を紹介する・CFQ スケジューリングは Linux でのプライオリティベースの I/O スケジューラである．近年の研究 [17] で CFQ スケジューラがライトバックキャッシュのためプライオリティを無視してしまう問題がある．

既存のブロック I/O トレーサー (例: blktrace [2]) ではこの問題を正しく把握することができない．wirte 時にはライトバックキャッシュと I/O スケジューリングの両方の依存関係が生じるが，blktrace では I/O スケジューリングの依存関係しかトレースできない．この問題を解析するためには，各依存関係をそれぞれ追跡するだけでは不十分である．なぜなら，ページキャッシュと I/O リクエストの対応関係をとらないと正しく I/O を生じさせたプロセスがわからないからである．つまり．依存関係間の対応関係を把握することも依存関係を含むデバッグの際に不可欠である．

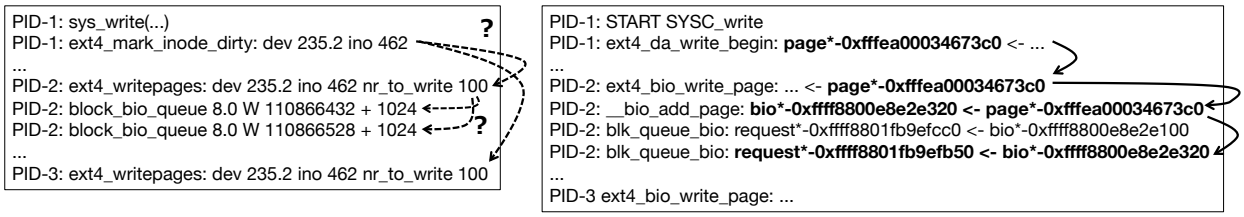
2.3 スレッド間の依存関係の追跡

2.1, 2.2 章で示した通り，スレッド間のデータ依存関係は現実で起きている問題を理解しにくくしている．既存のトレースツールの問題はこの依存関係を意識していない設計なっていることである．ここでは，図 **??** にある Ftrace と K9 のログの例からこの問題について議論する．

図 6 (a) は Ftrace の system calls, ext4, block イベントにおけるログを簡単にしたものである．それぞれのイベントでは各サブシステムにおける詳細な情報を出力してくれる．例えば，ページキャッシュがダーティになったとき，ダーティになったデバイス番号や inode 番号を出してくれる．また，ページのフラッシュ時は，フラッシュされるページ数も同様に出力される．block イベントではブロック I/O に関する情報が出力される．

しかしながら，この情報だけではライトバックキャッシュの依存関係を追跡でない．なぜなら，操作されたページを特定する情報がどこにも出力されていないからである．さらに，どのページがどの I/O リクエストを生じさせているのかもわからない．以上から，スレッド間の依存関係を追跡するためにはより細かに共有データに対する操作をロギングし，依存関係のデータフローも追跡する必要がある．

一方，K9 はスレッド間の依存関係を含めたシステムの挙動を追跡するのに十分な情報を出力できる．図 6 (b) より，ログにはデータのアドレスをグローバルな ID として使用



(a) Simplified logs from the syscall, ext4, and block events of Ftrace (b) A part of logs ouput by K9 that can trace the inter-thread data dependencies

図 6 Ftrace と K9 によるログの比較

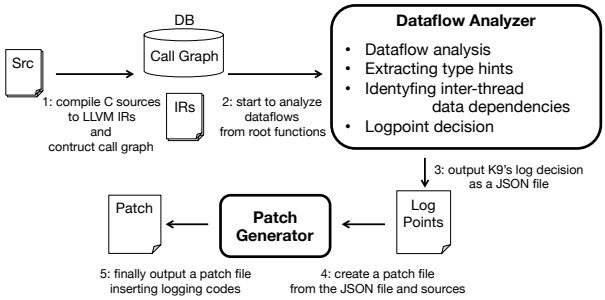


図 7 K9 の概要

している．この ID によってスレッドをまたいだログ間の対応関係が取れるようになる．例えばプロセス ID が 2 のプロセスによってフラッシュされるページは PID が 1 のプロセスがダーティにしたものだとわかる．また，Ftrace を違い，依存関係間も追跡することができる．

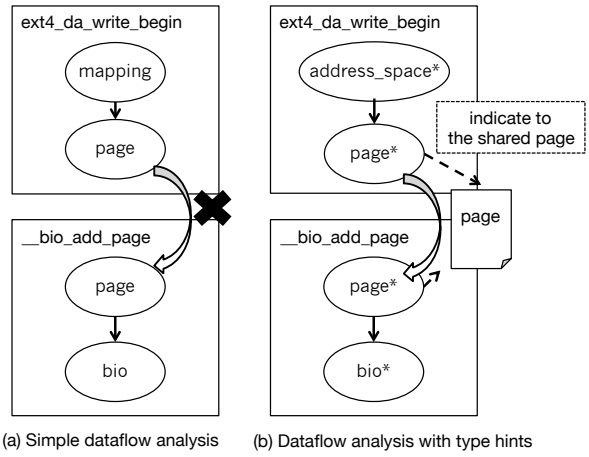
しかし，K9 のような非同期処理のための高度なロギングは大変である．開発者は共有データを特定し，どこで依存関係が発生するのか特定しなければいけない．また，依存関係間もトレース必要がある．これらを手動で行なうのは時間がかかり，現実的ではない．

3. K9 の設計

そこで本研究では K9 というトレースツールを実装した．K9 は静的解析を行い，スレッド間をまたいでデータ依存関係を引き起こしうるデータフローの組を特定し，自動でそれを追跡するログを挿入する．よって，K9 は複雑なデータフローを追跡することができ，デバッグ時に開発者の支援を行なう．

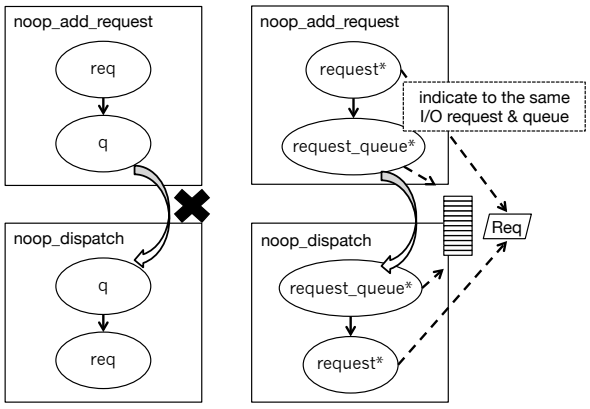
図 7 は K9 の概要を示している．今回, LLVM コンパイラフレームワーク [6] を利用し LLVM IR 上で解析を行った．K9 は解析を行なう前にソースコードを LLVM IR に変換し，コールグラフを作成する．

K9 はまず，IR ファイルと解析を始める関数を引数としてとり，データフロー解析を行う．本研究では，ライトバックキャッシュと I/O スケジューリングに注目する．これらの依存関係の調査をもとに，K9 は structure type sensitive, primitive type insensitive な解析を行なう．また，K9 は 3.2 章で示す型の特徴も同時に収集する．



(a) Simple dataflow analysis (b) Dataflow analysis with type hints

図 8 ext4_da_write_begin() , __bio_add_page() のデータフローグラフ



(a) Simple dataflow analysis (b) Dataflow analysis with type hints

図 9 noop_add_request() , noop_dispatch() のデータフローグラフ

データフロー解析後，K9 は型の特徴を利用し，自動でデータフローの組を特定してそのデータフローをログポイントとする．さらに，K9 は依存関係が把握できるようにデータフローを追跡しそれもログポイントとする．K9 はログポイントを JSON ファイルとして出力する．

最後に K9 は JSON ファイルからログを挿入するパッチファイルを生成する

3.1 データフロー解析

K9 は各関数ごとにデータフロー解析を行なう．そして，callee 関数から caller 関数へとデータフローを伝搬することで暗黙的なデータフローを解析する．

まず，K9 は各関数ごとにデータフロー解析を行い，データフローグラフを作成する．K9 は構造体，もしくは構造体のポインタがある代入文を解析する．これは，K9 は structure type sensitive，primitive type insensitive な解析をするためである．なので，データフローグラフには primitve な変数ノードは存在しない．

左辺が構造体または構造体のメンバのとき，また，右辺が関数呼び出しのとき，K9 は右辺の変数から左辺の変数へとデータフローが発生していると解析する．例えば，図 2 の 12 行目では *page* から *bvec* へのデータフローが発生する．右辺が関数呼び出しのとき，引数として渡されているすべての変数から左辺の変数へのデータフローとして解析する．

ログによるオーバヘッドを減らすために，K9 では不必要なデータフローを解析しない．K9 は自動で追加のログを挿入するが，これは非常にたかいオーバヘッドを起こす原因になりやすい．このため，K9 はすべての代入文をデータフローが発生するものとは解析しない．

左辺がポインタで構造体のメンバではないとき，K9 は左辺の変数を右辺の変数へのリダイレクトして解析する．例えば，図 2 の 11 行目で *bvec* は *bio* へのリダイレクトなる．そして，12 行目で *page* から *bio* のデータフローが起きていると解析する．

K9 は暗黙的に存在するデータフローを callee 関数から caller 関数に伝搬させることで追跡する．関数それぞれにデータフロー解析を行なうだけでは，callee 関数によって発生する暗黙的なデータフローを追跡できない．K9 はもし引数間でデータフローが発生し，引数のうち 1 つでも構造体のメンバとして渡せれているならば，そのデータフローを caller 関数に伝搬させる．例えば，list_head 関数によって生じる *req* から *q* のデータフローを noop_add_request 関数へと図 9 (a) のように伝搬する．

3.2 構造体の型の特徴

K9 はどの構造体の型が共有データになるのか把握するために，構造体の型の特徴をデータフローの最中に取得する．ここでは 3 つの特徴に注目する (1) データがヒープ領域にあるか (2) ロック可能か (3) リストであるか．

スレッド間でデータを共有するのに，そのデータは普通ヒープ領域に生成される．K9 では関数名からメモリ割り当ての関数 (e.g., kmalloc(), kmem_cache_alloc(), ...) を特定し，その返り値の型を解析することでどの構造体の型がヒープ領域に置かれるか解析する．

また，どの構造体がロック可能であるかも解析する．この情報はページキャッシュなどの共有データの特定に役立つ．K9 では関数名かロック取得関数 (e.g., spin_lock(), lock_page()) を特定し，その引数を解析することでどの構造体の型がロック可能か解析する．

最後に，どの構造体がリスト構造を持つか解析する．これは I/O スケジューリングなどのリスト操作による依存関係の検出精度を向上させる．K9 は次のようにして，リスト構造を特定する．まず，単純に自分自身へのポインタをもつ構造体をリストと考える．そして，list_head のような共通のインタフェースをもつ構造体もリスト構造と考える (e.g. *request* and *request_queue*).

3.3 依存関係の検出

K9 はスレッド間の依存関係を型の特徴を利用してデータフロー解析を行なうことで検出する．

ライトバックキャッシュ: K9 は次の条件を満たすときデータフロー組をライトバックキャッシュのような依存関係を発生させるものとして検出する．

$$df1 \in func1 : typeA \rightarrow typeB$$
$$df2 \in func2 : typeB \rightarrow typeC$$

Cond 1: *func1* $\neq$ *func2* && *func1* and *func2* are not callers of each other

Cond 2: *type A* and *type B* can be allocated on the heap and lockable

ます，K9 は候補のデータフロー (*df1* and *df2*) が別々の実行パスで発生するかチェックする **Cond 1**. K9 は 2 つの関数が同じもしくは互いの caller 関数になっていないかチェックする．そしてデータフローがライトバックキャッシュの特徴を示しているかチェックする．typeB は共有データの型を示しており，これはヒープ領域に置かれ，ロック可能でなければいけない．typeA は共有データを管理する型を想定しており，同様にチェックする．

例として，図 2 ある ext4 においての検出について示す．まず，図 8 (a) から 図 8 (b) のように型をデータノードの ID にする．そして *address_space** から *page** を通して *bio** へと続くデータフローを検出しようとする．

K9 は次のように検出する．まず，K9 は ext4_da_write_begin 関数と __bio_add_page 関数がお互いの caller 関数になっていないか調べる．そして, K9 は *page** 参照する型の *page* がヒープ領域に置かれ，ロック可能であるかチェックする．*address_space* も同様にチェックする．

***I/O* キュー**: K9 は次の条件を満たすときデータフロー組を I/O スケジューリングのような依存関係を発生させるものとして検出する．

$$df1 \in func1 : typeA \rightarrow typeB$$

Algorithm 1 依存関係間の追跡

Require: K9 tries to trace between *func1* and *func2* and log the dataflows from *typeA* to *typeB*

```
1:  procedure TRACE(func1, func2, typeA, typeB)
2:      Funcs ← GATHER(func1, func2)
3:      if Funcs is empty then
4:          for each caller of func1 do
5:              if caller has the typeA argument then
6:                  TRACE(caller, func2, typeA, typeB)
7:              end if
8:          end for
9:      else
10:         SEARCHANDLOG(Funcs, typeA, typeB)
11:     end if
12: end procedure
```

$$df2 \in func2 : typeB \rightarrow typeA$$

Cond 1: *func1* $\neq$ *func2*

Cond 2: *type A* and *type B* can be allocated on the heap and lists

Cond 3: *df1* and *df2* are list operations

まず，ライトバックキャッシュとは違い，K9 は 2 つの関数が同一でないかだけチェックする **Cond 1**．これは，I/O スケジューリングは一度発行されたリクエストが再度挿入される可能性があるからである．そして，I/O リクエストとキューの特徴を踏まえ，typeA と typeB の両方共 heep 領域に置かれ，リスト構造であるかチェックする **Cond 2**．最後に，LLVM IR を解析して 2 つのデータフローがリスト操作であるかチェックする．

例として，図 5 の場合での依存関係の検出に説明する．ライトバックキャッシュ時と同様に，図 9 (a) から 図 9 (b) のように型をデータノードの ID にするそして *request_queue** と *request** 間での I/O スケジューリングのデータフローを検出しようとする．

K9 は次のように検出する．まず, K9 は *request* と *request_queue* がヒープ領域に置かれ，リストであるかチェックする．そしてデータフローがリスト操作であることを LLVM IR から *list_head* を通してのデータフローだと解析することによって確認するライトバックキャッシュとは違い， noop_add_request 関数 と noop_dispatch 関数がお互いの caller 関数であるかは調べない．

3.4 依存関係間の追跡

依存関係を個別に追跡するだけでは複数の依存関係を含む処理を理解できないたとえば，Linux のブロック I/O では write 時にライトバックキャッシュと I/O スケジューリングの 2 つの依存関係が発生する．すでにそれぞれの検出方法について述べたが，ここで依存関係間の追跡方法について言及する．

アルゴリズム 1 は K9 がどのように依存関係間を追跡するのか示している．まず，K9 は依存関係間の関数を収集し，それから依存関係間を追跡できるようなデータフローを探す．例えばライトバックキャッシュと I/O スケジューリングの例では，*bio** から *request** へのデータフローを __bio_add_page 関数 と noop_add_request 関数の間から探す．

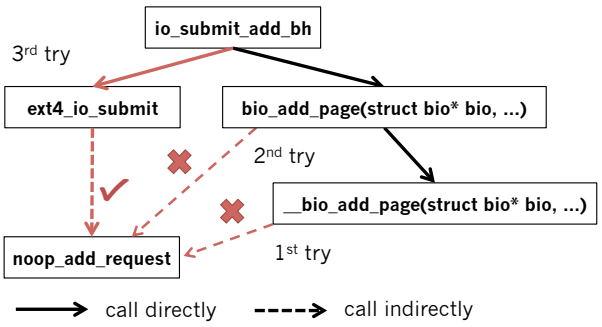


図 10 ライトバックキャッシュと I/O スケジューリング間の関数を収集する様子の一例

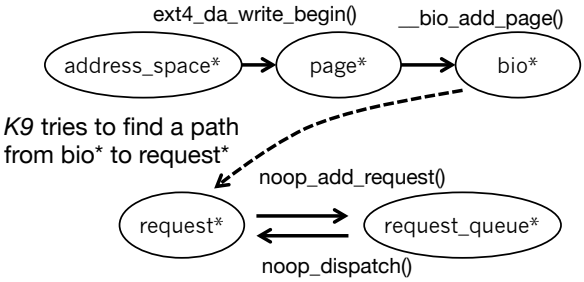


図 11 依存関係間を追跡できるようにデータフローを探索する例

まず，K9 は $func1$ と $func2$ の間の関数をコールグラフから収集する．もし関数が存在しなかったら，K9 は $func1$ の caller 関数から再度収集を試みる．しかしながら，際限なく caller 関数から収集するのは無関係な関数を多く含む可能性が高くなってしまう．そこで K9 は $typeA$ の引数を持たない関数の時は収集を諦める．$typeA$ は探そうとしているデータフローの始点である．関数の収集後，依存関係間を追跡できるようなデータフローを探しロギングする．

例えばライトバックキャッシュと I/O スケジューリングの例では，図 10 の用に関数を収集する．そして，図 11 のようにデータフローグラフを型単位に変換してデータフローのパスを探す．

3.5 ログの挿入

本研究では，Linux のトレースフレームワークでもある Ftrace をログをとるライブラリとして使用している．K9 はあらたにトレースイベントとトレースポイントを定義している．挿入されるログは下のようになる．

```
bvec->bv_page = page;

+ trace_dataflow(FuncName, LogID,
```

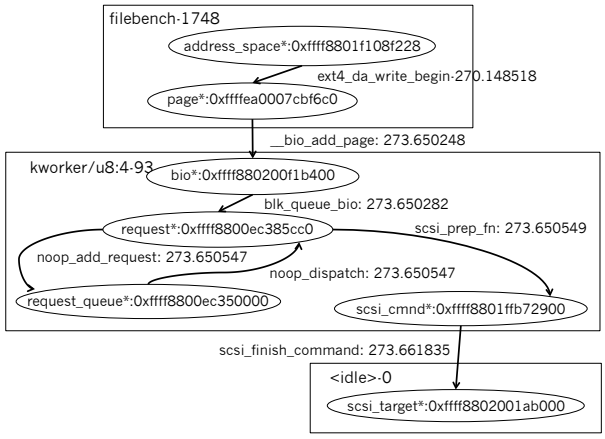


図 12 ログのフロー図: K9 がスレッドをまたぐ依存関係を追跡できていることがわかる

```
"bio*", bio, "page*", page);
```

書くログは関数名，プロセス名，プロセス ID をデータフローが発生した時のコンテキストの特定のため記録する．また，ログポイントに唯一な ID も付与する．さらに，データフローの情報も出力しタイムスタンプを出力する．

3.6 最適化する上での注意

完全に自動で新たにログを挿入することの問題はオーバヘッドである．K9 は型の特徴を利用することで解析の精度を上げているが，それでも不必要なログが存在する．しかししながら，K9 は開発者によるログポイントの変更を許可している．開発者は用途に応じでログポイントを変更しオーバヘッドを減らすことができる．本研究では，評価において効率的な最適化の方法について言及する．

4. 実験

ここでは次の 3 つの観点から K9 を評価する．(1) K9 のログによって言及した 2 つのデータ依存関係が追跡できるのか？(2) またデバッグの際にどれくらい有用なのか？(3) どれくらいのオーバヘッドが K9 のログによって生じるのか?

4.1 Tracing Dependencies

ここでは K9 のログによって正しくスレッド間の依存関係がトレースできていることを確認する．まず，write システムコールをトレースするために Linux カーネル v4.2.3 に K9 によってログを挿入した．そして，4 スレッドで 1000 の非同期な write をおこない sync コマンドを発行して，kworker に I/O を委任させた．

図 12 が示す通り，ログによってライトバックキャッシュと I/O スケジューリングの依存関係がトレースできている．また，SCSI コマンドの完了についてもトレースできている．よって，K9 は適切に依存関係を追跡するためのログを挿入でき，依存関係を含む処理を把握することができる．

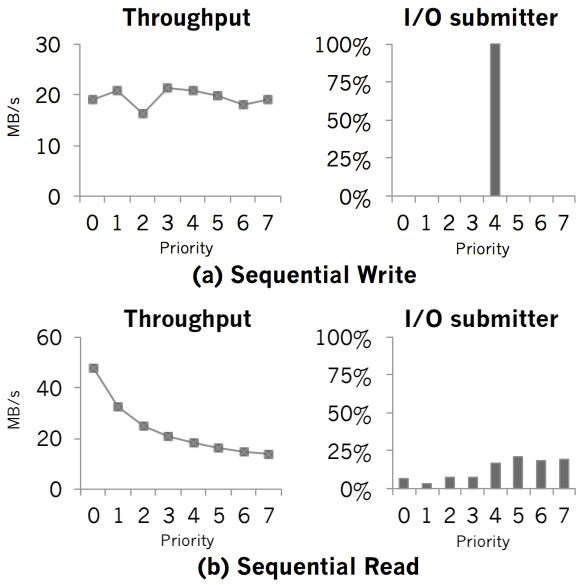


図 13 CFQ スループットと I/O 発行におけるプライオリティごとの比較

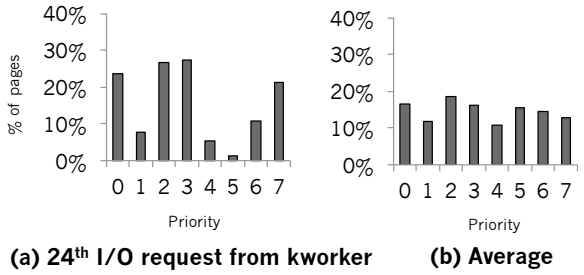


図 14 I/O の原因となったページの割合

ライトバックキャッシュと I/O スケジューリングを追跡できることにより，CFQ の問題 [17] を明確に理解することができる．CFQ スケジューリングミスを示すために，プライオリティを変えて write と read をシーケンシャルに行った．図 13 より，CFQ が write においてプライオリティを考慮できていないことがわかる．さらに，K9 ではより細かく解析ができるので，図 14 のように kworker によってプライオリティの違うページ更新がマージされていることがログよりわかる．以上から K9 は開発者が非同期処理においてより深い考察を可能にしている．

4.2 デバッグにおける有用性

2.1 章で示した btrfs のバグをトレースするために図 15 のようなログを K9 によって挿入した．ログ ID 1396 のログは clear_page_dirty_for_io 関数の直前にあり，どのページのダーティフラグが落とされるかわかる．ID 1397 のログによってどのページがエラーハンドリングされているかがわかる．ID 1401 のログはどのページが解放されるか示している．

```
just-wr-5165 ... : write_one_eb-ID:1396 page*-0xffffea00045ee800 <- extent_buffer*-0xffff88007065a230
just-wr-5165 ... : clear_page_dirty_for_io-ID:3251 address_space*-0xffff8800705e42e8 <- page*-0xffffea00045ee800
                                  trigger memory allocation error by fault injection
just-wr-5165 ... : write_one_eb-ID:1397 page*-0xffffea00045ee800 <- extent_buffer*-0xffff88007065a230
...                                                                                  in the error handling
just-wr-5165 ... : write_one_eb-ID:1397 page*-0xffffea0004682400 <- extent_buffer*-0xffff88007065a230
                                  does not call clear_page_dirty_for_io() for this page
                                  and this page's dirty flag remains
bash-5134    ... : btrfs_release_...-ID:1401 page*-0xffffea0004682400 <- extent_buffer*-0xffff88007065a230
                                  the kernel panics at BUG_ON(PageDirty(page)) after this log
```

図 16 パニック時の K9 のログ

```
#1   static noinline_for_stack int write_one_eb(...) {
#2       ...
#3       struct page *p = extent_buffer_page(eb, i);
#4       + trace_dataflow(__func__ , 1396, "page*", p, "extent_buffer*", eb);
#5       clear_page_dirty_for_io(p); // clear the p's dirty flag
#6       ...
#7       if(unlikely(ret)) { // error handling
#8           ...
#9           struct page *p = extent_buffer_page(eb, i);
#10          + trace_dataflow(__func__ , 1397, "page*", p, "extent_buffer*", eb);
#11          unlock_page(p);
#12      }
#13  }
#14  ...
#15  static void btrfs_release_extent_buffer_page(...) {
#16      ...
#17      + trace_dataflow(__func__ , 1401, "page*", page, "extent_buffer*", eb);
#18      if(page && mapped) {
#19          ...
#20          BUG_ON(PageDirty(page)); // trigger the kernel panic
#21          ...
#22  }
```

図 15 btrfs に挿入したログ

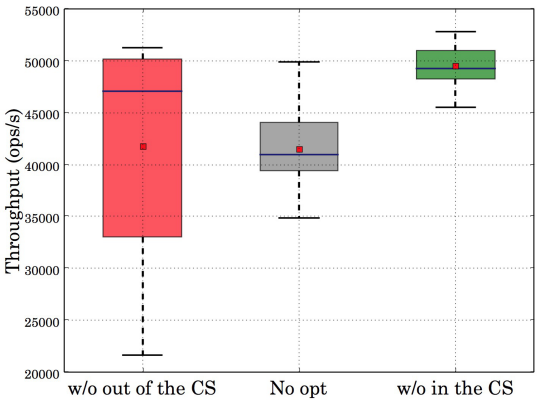


図 17 クリティカルセクション外のログよりもクリティカルセクション内のログから削除した方がいいことを示している

表 1 K9 によるパフォーマンスオーバヘッド

Workload	Throughput Slowdown		CPU Usage Increase	
	No Opt	Opt	No Opt	Opt
filebench-seqwrite	40.28%	28.80%	-0.17%	0.29%
filebench-varmail	3.46%	2.12%	1.84%	-0.18%
tpcc-mysql	-0.30%	1.63%	0.60%	0.22%
ycsb-mongodb	0.93%	-0.86%	0.17%	1.84%

図 16 パニックじの K9 のログの一部である. ログによると，アドレス 0xffffea0004682400 のページで解放時にカーネルパニックが起きている．なぜ，このページがエラー状態にあったのか理解するため，このページに関するログを追跡すると，このページに対して clear_page_dirty_for_io が呼ばれていないことがわかる．そして他のページのログと比較することによって，エラーが起きた時に clear_page_dirty_for_io を呼ぶ機会ないことがわかる．よって，エラーハンドリングないでの clear_page_dirty_for_io を呼ぶ必要性がわかる．

4.3 パフォーマンスオーバヘッド

ここでは K9 のオーバヘッドについて議論する測定環境は Linux カーネル v4.2.3, ext4 ファイルシステム, Intel Xeon E3-1220 v2 3.10GHz 1P/4C プロセッサ, 8GB メモリ, そして HP 500GB 7.2kprm SC 3.5-inch SATA HDD である．Ftrace によって K9 のログはメモリに出力され，この時のスループット減少と CPU 使用率の増加について 4 つのワークロードで測定した: *filebench-varmail*, *tpcc-mysql*, *ycsb-mongodb*, そして *filebench-seqwrite* である．

K9 のログによるオーバヘッドは I/O 処理にかかる時間のほうが長いため許容範囲である．ここでは Linux カーネルに write, read, pwrite, pread, open, close, そして fsync での依存関係を追跡するために 1267 個のログを挿入してオーバヘッドを測定した．また，後で示す最適化をほどこした 1252 個のログの挿入のときでも測定した．

表 1 が示す通り，メールサーバーのエミュレーションである *filebench-varmail* においたった 3.46% のオーバヘッドですんだ．また，データベースにおいても非常に小さいオーバヘッドですんでいる．これはデータベースのような信頼性求められるアプリケーションでは fsync を多用することによって I/O が多く発生するからである．このように K9 は現実のアプリケーションにおいて非常に小さいオーバヘッドでログを挿入できる．

より，詳しくオーバヘッドを議論するために，オーバヘッドの出やすい I/O を発行しないワークロードである *filebench-seqwrite* でのオーバヘッドも測定した．*filebench-seqwrite* は 4 スレッドで非同期な write をおこなう．挿入したログの数は write をトレースするための 1161 個である．このとき，表 1 にあるとおりスループットの減少が

40.28% と大きくなってしまう．

この原因と最適化を行なうために調査を行った．もし，各ログポイントの出力コストが一定ならば，出力割合が多いログポイントを削除していくのが直感的である．しかし，これでは逆にオーバヘッドが大きくなってしまう．

これは出力量だけ考慮するとクリティカルセクション外のログを多く削除してしまい．この結果，ロックの解放待ちが多くなりスループットが激減してしまう．これを確かめるために，一番出力を占めているパス探査に関するログ (全体の 28.8%) を削除した場合とクリティカルセクション内のログ (全体の 25.2%) から削除した場合を比較した．結果，図 17 が示す通り，クリティカルセクション内のログから削除したほうが安定的にスループット向上した．よって，スループットを向上するにはクリティカルセクション内のログから削除したほうがいい．

5. 関連研究

多くの研究がログの挿入は専門的な知識が必要で大変である言及している [4], [16], [19], [20], [21], [23]．LogEnhancer [19] は自動で既存のログの出力内容を改善する．しかし，LogEnhancer では新しくログを追加することはできない．これを解決するために幾つかのロギングツール [20], [23] が提案された．Errlog [20] と LogAdvisor [23] はエラーハンドリング内でログを挿入すべき場所に自動でエラーログの挿入もしくは提案を行なう．K9 もまた自動でログを挿入するが，スレッド間のデータ依存関係を追跡することに焦点をあてている．

いくつかの研究 [3], [8], [22] で非同期処理をログ解析することで分散システムの問題を解決している．これらの研究はスレッドをまたいだデータ依存関係を追跡することで分散システムを管理，プロファイリングする．しかしながら，これらの研究は開発者によってすでにログが挿入されていることを想定している．さらに，本研究ではオペレーティングシステムにおいても分散システムのようなデータ依存関係がおき，K9 の有用性を示している．

I/O レイヤにおいてデータ構造を変更し，依存関係を追跡する手法が存在する．これらの研究 [10], [15], [17] ではデータにタグをつけ，正しくスレッドをまたぐ依存関係を把握する．K9 のログ自体はその研究の問題を解決することはできないが，K9 はそれらの研究を支援するツールとして有用であると考えている．

静的解析を用いてある特定の問題 [5], [11], [14], [18] を解決する提案が多数存在する．Gist [5] は動的にハードウェアウォッチポイントを利用して動的にスレッドをまたぐ実行を追跡する．一方，K9 は静的にスレッド間のデータ依存関係を検出し，追跡するためのログを挿入する．SherLog [18] はエラーログを用いて実行パスを再構成し，バグの原因の切り分けをおこなう．しかし，これはスレッド間をまたぐ情報は対象外である．

6. 結論

本研究では，スレッド間のデータ依存関係を追跡できるログの自動挿入を行なうツール K9 の提案と実装を行った．K9 はライトバックキャッシュと I/O スケジューリングの依存関係をトレースできる．実際に btrfs と CFQ の問題について K9 の有用性を示した．また，K9 のオーバヘッドはデータベースアプリケーションにおいて 1.63% のスループット減少と 1.84% の CPU 使用率の増加であった．

参考文献

[1] Oops tracing. `https://www.kernel.org/doc/Documentation/oops-tracing.txt`.

[2] Jens Axboe. blktrace User Guide, 2007. `http://www.cse.unsw.edu.au/~aaronc/iosched/doc/blktrace.html`.

[3] MIchael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *In Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 217–231, Broomfield, CO, 2014.

[4] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where Do Developers Log? An Empirical Study on Logging Practices in Industry Where Do Developers Log? An Empirical Study on Logging Practices in Industry. In *In Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, Hyderabad, India, 2014.

[5] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-Production Failures. In *In Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, USA, 2015.

[6] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO '04)*, Palo Alto, California, 2004.

[7] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A Study of Linux File System Evolution. In *In Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13)*, pages 31–44, San Jose, CA, 2013.

[8] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *In Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, USA, 2015.

[9] Chris Mason. The Btrfs Filesystem, 2007. `https://oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf`.

[10] Michael Mesnier, Jason B. Akers, Feng Chen, and Tian Luo. Differentiated Storage Services. In *In Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, 2011.

[11] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking Semantic Correctness: The Case of Finding File System Bugs. In *In*

Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '15), 2015.

[12] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: Ten Years Later. In *In 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, 2011.

[13] Steven Rostedt. ftrace - function tracer, 2008. `http://www.kernel.org/doc/Documentation/trace/ftrace.txt`.

[14] Suman Saha, Jean-Pierre Lozi, Gael Thomas, Julia L. Lawall, and Gilles Muller. Hector: Detecting Resource-Release Omission Faults in Error-handling Code for Systems Software. In *In Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*, pages 1–12, 2013.

[15] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A Software-Defined Storage Architecture. In *In Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Pennsylvania, USA, 2013.

[16] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Detecting Large-Scale System Problems by Mining Console Logs. In *In Proceedings of the 22th ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, MT, 2009.

[17] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Split-Level I/O Scheduling. In *In Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, USA, 2015.

[18] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. In *In Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 143–154, New York, NY, USA, 2010.

[19] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Improving Software Diagnosability via Log Enhancement. *ACM Trans. Comput. Syst.*, 30(1):4:1–4:28, 2012.

[20] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *In Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 293–306, Hollywood, CA, 2012.

[21] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing Logging Practices in Open-Source Software. In *In Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, Zurich Switzerland, 2012.

[22] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *In Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 629–644, Broomfield, CO, 2014.

[23] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. Learning to Log: Helping Developers Make Informed Logging Decisions. In *In Proceedings of the 37 th International Conference on Software Engineering (ICSE '15)*, Florence/Firenze Italy, 2015.