Logging Inter-Thread Data Dependencies in Linux Kernel

Takafumi KUBOTA, Naohiro AOTA, Nonmembers, and Kenji KONO, Member

SUMMARY  Logging is a practical and useful way of diagnosing failures in software systems. The logged events are crucially important to learning what happened during a failure. If key events are not logged, it is almost impossible to track error propagations in the diagnosis. Tracking an error propagation becomes utterly complicated if inter-thread data dependency is involved. An inter-thread data dependency arises when one thread updates a shared propagation and hinders the diagnosis. An inter-thread data dependency happens during a failure.

Unfortunately, the code locations that can cause inter-thread data dependencies are not obvious. Generally, deciding appropriate log points is difficult and time-consuming. Existing logging practice depends on the expertise of software engineers. Logging code is inserted ad hoc at code locations that they feel to be important and useful. However, such ad-hoc practice misses important logging opportunities, and the need for more systematic approaches has been advocated.

This paper presents the design and implementation of K9, a tool that automatically inserts logging code to trace inter-thread data dependencies. K9 identifies code locations where the inter-thread dependency can occur, and inserts logging code to record which thread reads from or writes on which data structure. To be a practical tool, K9 is designed to 1) scale to large multi-threaded systems and 2) cause negligible runtime overheads. K9 identifies code locations where the inter-thread dependency can occur, and inserts logging code to the Linux with negligible runtime overheads. To scale to large systems, K9 avoids complicated static analyses. Instead, K9 takes a best-effort approach. It identifies ‘typical’ data structures that are often shared among threads and detects ‘most’ of the accesses to them. To determine typical structures and accesses to them, K9 leverages coding conventions ubiquitous in real-world software systems.

To confirm the scalability of K9, we have applied it to the Linux kernel. K9 inserts logging code to the Linux with a reasonable amount of time. The logs generated by K9 provides useful information for diagnosis. K9 inserts three failures by injecting known bugs in Linux. All of these failures involve inter-thread data dependencies. K9 generated logs that tell us which thread accessed which data structure, and allowed us to trace back from the failure sites to the root causes.

Failure diagnosis starts with tracing an error propagation back to a root cause. This paper addresses the problem of inter-thread data dependency, which tangles up an error propagation and hinders the diagnosis. An inter-thread data dependency is incurred when one thread updates a shared data structure and another thread reads it later. Since an error propagates from a buggy thread to a failing thread, investigating the code executed by the failing thread never reveals the root cause; software engineers would understand the failing thread reads a corrupt data but would not be able to figure out which thread corrupted the data. Logging the inter-thread data dependency is critically important to diagnosing failures in multi-threaded systems.

 Unfortunately, the code locations that can cause inter-thread data dependencies are not obvious. Generally, deciding appropriate log points is difficult and time-consuming. Existing logging practice depends on the expertise of software engineers. Logging code is inserted ad hoc at code locations that they feel to be important and useful. However, such ad-hoc practice misses important logging opportunities, and the need for more systematic approaches has been advocated.

This paper presents the design and implementation of K9, a logging tool that automatically inserts logging code to trace inter-thread data dependencies. K9 identifies code locations where the inter-thread dependency can occur, and inserts logging code to record which thread reads from or writes on which data structure. To be a practical tool, K9 is designed to 1) scale to large multi-threaded systems and 2) cause negligible runtime overheads. To scale to large systems, K9 avoids complicated static analyses. Instead, K9 takes a best-effort approach. It identifies ‘typical’ data structures that are often shared among threads and detects ‘most’ of the accesses to them. To determine typical structures and accesses to them, K9 leverages coding conventions ubiquitous in real-world software systems.

To confirm the scalability of K9, we have applied it to the Linux kernel. K9 inserts logging code to the Linux with a reasonable amount of time. The logs generated by K9 provides useful information for diagnosis. We reproduced three failures by injecting known bugs in Linux. All of these failures involve inter-thread data dependencies. K9 generated logs that told us which thread accessed which data structure, and allowed us to trace back from the failure sites to the root causes. In addition, we have used K9 to diagnose a failure caused by a previously unknown bug. The bug-fix has been submitted to the Linux community and accepted.

The current design of K9 has some limitations. First, K9 does not capture all the code locations that can cause the inter-thread data dependency, since it relies on the coding conventions; K9 cannot capture the inter-thread dependency caused by the coding style deviating from the conventions.

1. Introduction

Logs are helpful for understanding the runtime behavior of software systems, and are widely used in software diagnosis and management, for example, in error debugging, performance analysis, anomaly detection, workload modeling, and system monitoring. Even though logs do not pinpoint root causes of system failures, they provide invaluable and helpful clues for software engineers to the root causes. The quality of a log is determined by what sort of events it records. If key events are not logged at all, the log is useless and it is hard to understand what happened during a failure.

Failure diagnosis starts with tracing an error propagation back to a root cause. This paper addresses the problem of inter-thread data dependency, which tangles up an error propagation and hinders the diagnosis. An inter-thread data dependency is incurred when one thread updates a shared data structure and another thread reads it later. Since an error propagates from a buggy thread to a failing thread, investigating the code executed by the failing thread never reveals the root cause; software engineers would understand the failing thread reads a corrupt data but would not be able to figure out which thread corrupted the data. Logging the inter-thread data dependency is critically important to diagnosing failures in multi-threaded systems.

Unfortunately, the code locations that can cause inter-thread data dependencies are not obvious. Generally, deciding appropriate log points is difficult and time-consuming. Existing logging practice depends on the expertise of software engineers. Logging code is inserted ad hoc at code locations that they feel to be important and useful. However, such ad-hoc practice misses important logging opportunities, and the need for more systematic approaches has been advocated.

This paper presents the design and implementation of K9, a logging tool that automatically inserts logging code to trace inter-thread data dependencies. K9 identifies code locations where the inter-thread dependency can occur, and inserts logging code to record which thread reads from or writes on which data structure. To be a practical tool, K9 is designed to 1) scale to large multi-threaded systems and 2) cause negligible runtime overheads. To scale to large systems, K9 avoids complicated static analyses. Instead, K9 takes a best-effort approach. It identifies ‘typical’ data structures that are often shared among threads and detects ‘most’ of the accesses to them. To determine typical structures and accesses to them, K9 leverages coding conventions ubiquitous in real-world software systems.

To confirm the scalability of K9, we have applied it to the Linux kernel. K9 inserts logging code to the Linux with a reasonable amount of time. The logs generated by K9 provides useful information for diagnosis. We reproduced three failures by injecting known bugs in Linux. All of these failures involve inter-thread data dependencies. K9 generated logs that told us which thread accessed which data structure, and allowed us to trace back from the failure sites to the root causes. In addition, we have used K9 to diagnose a failure caused by a previously unknown bug. The bug-fix has been submitted to the Linux community and accepted.

The current design of K9 has some limitations. First, K9 does not capture all the code locations that can cause the inter-thread data dependency, since it relies on the coding conventions; K9 cannot capture the inter-thread dependency caused by the coding style deviating from the conventions.
Second, K9 is designed for software systems written in C. In spite of these limitations, our experimental results suggest K9 captures code locations enough to diagnose bugs, involving inter-thread dependency, in the real world. In addition, there are still many software systems written in C.

The rest of this paper is organized as follows. First, we introduce inter-thread data dependency and discuss motivating bug examples in Linux (Sect. 2). Next, we discuss the design principles of K9 (Sect. 3). We then describe K9’s dependency model (Sect. 4) and its design and implementation (Sect. 5). After that we apply K9 to the Linux kernel and evaluate the logs inserted by K9 (Sect. 6). Finally, we discuss related work (Sect. 7) and conclude (Sect. 8).

2. Motivation

Inter-thread data dependencies complicate failure diagnosis because an error propagates from one thread to another through shared data structures. Section 2.1 introduces two types of inter-thread data dependency, and Sect. 2.2 shows Linux bug examples.

2.1 Inter-Thread Data Dependency

An inter-thread data dependency is a situation in which a thread’s state depends on a shared data structure that has been initialized or modified by other threads. Two threads have inter-thread data dependency if one updates a shared data structure and the other reads it afterwards, because the update of the shared data may affect the reading thread’s behavior.

Inter-thread data dependencies are classified into two categories: 1) direct and 2) indirect. A direct dependency occurs when two threads directly share a data structure; the one updates the shared data structure, and the other reads it later. Figure 1 (a) illustrates a direct dependency. Thread A updates the shared data structure and then thread B reads it. An indirect dependency is caused by an intermediate thread that correlates distinct shared data structures. Figure 1 (b) illustrates an indirect dependency. In the figure, thread Y correlates one shared structure M with another shared structure N. Thread Y reads shared structure M and updates another structure N with some value calculated from M. Threads X and Z do not have a direct dependency, but they have an indirect dependency through thread Y; the update by thread X may affect the value read by thread Z. An indirect dependency typically appears when a thread generates a new data structure from the existing one.

Inter-thread data dependencies often occur in multi-threaded system like the Linux kernel. Figure 2 illustrates inter-thread data dependencies in the Linux block I/O subsystem. In the figure, write system calls are issued concurrently, and eight threads (from t0 to t7) are serving the calls. Each thread updates the page cache managed by a radix tree. Then, some thread (t4 in Fig 2) gets pointers to dirty pages in the tree. Here, direct dependencies occur between thread t4 and all other threads (t0 to t7). Next, thread t4 generates I/O requests from the dirty pages in the tree and inserts them in the I/O request queue. Then, the thread scheduled by chance (t3 in Fig 2) dispatches the I/O requests to the disk driver. Here, an indirect dependency arises between t3 and t0=7 because thread t4 correlates the tree and the I/O request queue.

2.2 Bug Examples in Linux

Let us describe two real bugs in the Linux kernel to illustrate the need for tracing inter-thread dependencies. The bug reported at the commit 8146502 is related to Btrfs subsystem [14]. Figure 3 shows an excerpt of the code, in which there is a direct dependency between kworker and sync through shared structures extent_buffer and page. Kworker flushes dirty pages and sync releases the pages X and Z do not have a direct dependency, but they have an indirect dependency through thread Y; the update by thread X may affect the value read by thread Z. An indirect dependency typically appears when a thread generates a new data structure from the existing one.

Inter-thread data dependencies often occur in multi-threaded system like the Linux kernel. Figure 2 illustrates inter-thread data dependencies in the Linux block I/O subsystem. In the figure, write system calls are issued concurrently, and eight threads (from t0 to t7) are serving the calls. Each thread updates the page cache managed by a radix tree. Then, some thread (t4 in Fig 2) gets pointers to dirty pages in the tree. Here, direct dependencies occur between thread t4 and all other threads (t0 to t7). Next, thread t4 generates I/O requests from the dirty pages in the tree and inserts them in the I/O request queue. Then, the thread scheduled by chance (t3 in Fig 2) dispatches the I/O requests to the disk driver. Here, an indirect dependency arises between t3 and t0=7 because thread t4 correlates the tree and the I/O request queue.

2.2 Bug Examples in Linux

Let us describe two real bugs in the Linux kernel to illustrate the need for tracing inter-thread dependencies. The bug reported at the commit 8146502 is related to Btrfs subsystem [14]. Figure 3 shows an excerpt of the code, in which there is a direct dependency between kworker and sync through shared structures extent_buffer and page. Kworker flushes dirty pages and sync releases the pages

![Inter-thread data dependencies](image1)

![Inter-thread data dependencies in write system call in Linux. Direct dependencies arise between t0 \( \cdots \) t7 and t4 through the page cache and between t4 and t5 through the I/O request. An indirect dependency arises between t0=7 and t4 through the I/O request.](image2)

![A bug in Btrfs in Linux kernel v3.17-rc5. An error propagates from kworker to sync through shared extent_buffer](image3)
stored in extent_buffer. If an error is encountered when a page is being flushed, Btrfs stops flushing the dirty pages and clears the dirty flags of all the remaining pages, so that they can be freed later by sync, which assumes the page to be freed is not dirty. In this bug kworker forgets to clear the dirty flags after an error is encountered at line S10. Thus, sync panics at line S13.

Another bug exemplifies an indirect dependency. It is the priority violation in Linux CFQ I/O scheduler, which aims to allocate disk time in proportion to priority. As pointed out in a recent study [15], CFQ ignores priorities if many threads with different priorities perform asynchronous writes. When eight threads with different priorities from 0 (highest) to 7 (lowest) are launched, it is expected that a thread’s write throughput is proportional to its priority. Unfortunately, CFQ ignores the priorities and the write throughput is not proportional as shown in Fig. 4 (a).

The CFQ priority violation is caused by two reasons. First, the priority of the thread submitting I/O requests is different from that of the thread that updates the page cache. Second, the priority difference is not tracked in the Linux kernel. For example, Fig. 2 illustrates the situation. Eight threads (t0 – t7) write on page caches asynchronously, where each thread index corresponds to its priority (t0’s priority is 0). Later, a thread (t4 in this example) gets dirty pages from the cache and submits I/O requests with priority 4 to the queue. Thus, all the I/O requests are treated with priority 4 as shown in Fig. 4 (b). In this example, an indirect dependency arises through thread t4 between the dirty page cache and the I/O request because thread t4 updates thread t0 on address 0x...9988. If an error is encountered when the page is being flushed, Btrfs stops flushing the dirty pages, so that all the remaining pages can be freed later by sync, which assumes the page to be freed is not dirty. In this bug kworker forgets to clear the dirty flags after an error is encountered at line S10. Thus, sync panics at line S13.

The work-flow of K9

![Diagram of K9 work-flow](image)

K9 identifies data structures likely to cause dependencies. It identifies “typical” data structures that are often shared among threads, and it detects data dependency can occur. Instead, it focuses on key data structures that cause dependencies.

In designing K9, the scalability is the most relevant to us because the Linux kernel is undoubtedly one of the largest, most complex software systems in the world. It has 11,544,016 LOC in 23,144 files.

Guided by our motivation for the scalability, K9 is designed with a best effort approach. Inter-procedural analysis of pointers becomes inevitable when all the code locations causing inter-thread dependency are to be identified. Unfortunately, pointer analysis is notoriously hard. It has been proven that a complete analysis is computationally undecidable [16], [17]. Furthermore, propagating pointer information among interleaving threads is a challenge, because the shared data can be accessed non-deterministically [18].

Giving up on reaching a perfect solution, K9 rather takes a best-effort approach to avoid complicated and time-consuming static analysis. It identifies “typical” data structures that are often shared among threads, and it detects “most” of the accesses to them. In other words, K9 does not try to detect every code location where an inter-thread data dependency can occur. Instead, it focuses on key data structures that cause dependencies.

Now, let us describe the overview of K9 work-flow as shown in Fig. 6. K9 takes source files as inputs and outputs a patch file that inserts logging statements for tracing direct...
and indirect data dependencies. The analysis of K9 consists of three stages. First, K9 identifies data types that can be shared among threads. Second, it detects the data-flows of the identified shared data that cause direct data dependencies. Last, it analyzes the data-flows between direct data dependencies to discover indirect data dependencies.

4. Data Dependency Model

To identify data structures shared among threads, K9 leverages coding conventions. Typically, a data structure shared among threads is allocated in a heap, referenced by pointers, and often managed in data collections. To discover such data structures, K9 analyzes the type information and identifies the collection and item types from the source code. Then, it checks the identified data type can be allocated in a heap by examining the existence of pointers for the data type. If K9 discovers such a data structure, it regards the structure as shared among threads. Since all shared data do not satisfy this assumption, K9 misses some of the shared structures. Despite this limitation, it identifies the primary data structures, and it has provided helpful clues to diagnose an unknown bug in Linux.

If we could assume all the critical sections are locked correctly, K9 would be able to focus on critical sections to extract shared data structures and the access to them. K9 did not take this approach since it is a debugging tool.

4.1 Collections and Items

K9 defines collections and items to select candidate structures that can be shared among threads. A collection is a data structure that groups a fixed or variable number of data items. Important examples of collections are queues and trees. Each element in a queue or each node in a tree is called an item in K9, and queues and trees are considered to be groups of those items. K9 assumes that collections are C structs and classifies them into a) array collections or b) graph collections. K9 uses the following heuristics to identify candidate structures shared by threads. C structs, which do not conform to array or graph collection, are not considered to be shared structures in K9.

Array Collection: An array collection is a C struct that contains an array that stores pointers to C structs. Each element in an array or graph collection is a collection linked to each other with pointers. As shown in Fig. 8 (a), a graph collection contains a graph head, which is a head of the collection (e.g., a head of a linked list). It usually contains metadata for controlling the collection (e.g., the length of a list and the lock for the list) in addition to pointers to data items. Graph items represent the items stored in the collection. If a C struct contains self-pointers, K9 regards it as a graph item type. Figure 8 (b) shows an example of graph collections from the Linux kernel.

4.2 Dependencies between Collections and Items

K9 records accesses to array or graph items in collections in order to trace data dependencies across threads.

Direct dependency: K9 identifies 1) adding/removing an item to/from a collection, and 2) reading/updating an item in a collection. Since the adding/removing operations update pointers to item types in collection types, K9 logs the assignments to an array of pointers to the item types in the array collection, and the assignments to self-pointers in the item type for graph collections. The operation to read or update an item is called the referencing operation. To reference an item, a pointer to the item is dereferenced, and thus K9 logs dereferences of pointers to item types.

Indirect dependency: Logging indirect dependencies are more complicated. An indirect dependency occurs when an item of one item type is updated with a value calculated from an item of another item type. K9 keeps track of data flows from one item type to another, and inserts logging code to record the update by a value originating from another item type.

4.3 Log Points

Here, we define log points which are code locations of log-
KUBOTA et al.: LOGGING INTER-THREAD DATA DEPENDENCIES IN LINUX KERNEL

1637

ging statements [1], [9], [11]. In particular, K9 identifies the following code locations as log points.

- updating and reading operations for array collections: e.g., i=col[idx]; and col[idx]=i;
- updating and reading operations for pointers in graph collections: e.g., col->next=i; and i=col->next;
- code locations where data-flows from a direct dependency to another direct dependency occur.

5. Design and Implementation of K9

The current prototype of K9 is implemented with the LLVM framework [19] (v.3.6.0). K9 works on the LLVM intermediate representation (IR) as in the related work [18], [20]–[28], although we will use the source code in the following explanation. To generate LLVM IRs, K9 hooks and modifies compile commands during the build of the Linux kernel. K9 generates a patch that injects logging code into the original source code. It generates a patch from the IR in four stages: 1) collection/item type identification, 2) data-flow graph generation, 3) direct dependency analysis, and 4) indirect dependency analysis.

5.1 Collection Support Library

K9 is simplified by making use of the coding conventions. Unfortunately, collection support library, often used in large software projects, complicates the analysis of direct/indirect dependencies. A collection support library provides common implementations for graph collections such as lists and trees. For example, Linux provides list_head for double-linked lists. By embedding it in other structs, you can turn the embedded struct into a double-linked list. All the operations are done in polymorphic functions taking list_head as arguments. A collection support library is a headache for K9 because it ‘conceals’ type names of graph items and collections. As shown in Fig. 5, K9 records the names of item and collection types so as to indicate which item/collection is accessed. A collection support library is a headache for K9 because it ‘conceals’ type names of graph items and collections. As shown in Fig. 5, K9 records the names of item and collection types so as to indicate which item/collection is accessed. If the I/O request queue in Fig. 2 is implemented with list_head, a naive implementation records that list_head is accessed since all the operations are done through list_head.

To deal with this problem, K9 keeps track of inter-procedural data flows between functions working on item/collection types and those working on support library types. This inter-procedural analysis allows it to generate the expected log that records the item/collection types into which a support library type is embedded.

5.2 Data-Flow Graph of K9

K9 employs an inter-procedural and path-insensitive data-flow graph that is used for the direct and the indirect dependency analysis. An inter-procedural data-flow graph is required because data flows from collection types to item/library types span beyond procedure boundaries. The graph is path-insensitive to avoid costly path-sensitive analyses [29], [30]. Lack of path-condition information may result in redundant logs, but does not miss log points; K9 prefers having redundant logs to missing logs.

A data-flow graph in K9 consists of two types of node and two types of edge. A function node represents each function in the source code. A variable node represents each variable within a function. A variable node contains the name and type information of the represented variable and information on whether its type belongs to a collection/item/library type. A call edge keeps the inter-procedural relationship between the caller and callee functions. It also maps variable nodes of the caller arguments to those of the callee arguments. A data-flow edge represents a data flow between variable nodes and contains the source code location where the data flow occurs.

Figure 9 describes an example of a data-flow graph. In Fig. 9 (b), there is a data-flow edge from head to item due to the assignment at line S14 in Fig. 9 (a). The assignment at line S15 causes a data flow from item to head. The call edge from sock_add_skb to skb_queue is created due to the function call at line S16.
S7. This call edge keeps the argument mapping between sock_add_skb.item and skb_queue.new, and between sock_add_skb.collection and skb_queue.head.

A data-flow edge maintains an op_kind attribute. This attribute is used to indicate whether the corresponding data flow is an adding/referencing operation or not. If “adding” is set as the attribute, the data flow implies an adding operation on a collection. If the attribute is “referencing”, the data flow is a referencing operation on a collection. Otherwise, this edge represents other operations, and the attribute becomes “other”.

K9 distinguishes logs about adding/removing operations of items from logs about referencing operations of items. In the case of array collections, the adding/removing operation means updating array elements; a reference to an array element like “c->items[i]” appears on left side of assignments. If a reference appears on the right side, it is the referencing operation. Otherwise, the attribute becomes “other”.

For the graph collection, there is an update on the self-pointers of a graph item type in adding/removing operations. If a reference to a self-pointer like “item->next” appears on the left side of an assignment, K9 sets “adding” as the attribute. If a reference to a self-pointer appears on the right side of an assignment, K9 sets “referencing” as the attribute. Otherwise, K9 sets “other” as the attribute. For graph head types, if a pointer to an item type in the graph head type appears on the left side of an assignment, K9 assigns “adding” as the attribute. If it appears on the right side, “referencing” is set as the attribute.

In the case of the graph head type, there is one exception. If a graph head type is assigned to a graph item type, it is not considered an “adding” operation even though a graph item type appears on the left side of an assignment. Since there is already a pointer from the head to the item, the operation on the reverse pointer does not have to be logged.

In Fig. 9, K9 sets “adding” as the attribute of the data-flow edge that corresponds to the assignment at line S14, because an item type (sk_buf) of a graph head (sk_buf_head) appears on the left side of the assignment. K9 sets “other” as the attribute for the assignment at line S15, because a graph head type is assigned to a graph item type here. This assignment does not have to be logged because the assignment at line S14 already indicates that the graph item has been added to a graph head.

There is a subtle problem with call edges. If a function call occurs through a function pointer, K9 has to decide which function is called through the function pointer. To get such information, it analyzes the assignments to the function pointers and saves to a database the relation between the function pointers and the assigned functions. When K9 reaches a function call through a function pointer, it queries which functions can be called through the function pointer and then creates call edges to all the callable functions.

5.3 Direct Dependency Analysis

K9 keeps track of data-flows from one item type to another to identify direct/indirect dependencies. K9 employs an inter-procedural and path-insensitive data-flow graph. An inter-procedural data-flow graph is required because data flows from collection types to item/library types span beyond procedure boundaries. The graph is path-insensitive to avoid costly path-sensitive analyses [29], [30]. Lack of path-condition information may result in redundant logs, but does not miss log points; K9 prefers having redundant logs to missing logs.

To record direct dependencies, K9 injects a logging statement that outputs "<item type> @ <item address> <-<collection type> @ <collection address>".

Array Collection: K9 searches for data-flow edges from an array collection type to an array item type. This edge implies an array item is referenced from, added to, or removed from an array collection. Since a data-flow edge records the code location, K9 inserts the logging code there.

Graph Collection: We will start with the case in which collection support libraries are not used. Since a graph item can be added to/referenced from another graph item or a graph head, K9 first searches for the data-flow edges from a graph item/head type to graph item/head type. Even if such a data-flow edge is found, the collection type of an item type cannot be identified directly. In a graph collection, a graph item is accessed through a graph head. To identify the graph collection type, K9 identifies data-flows from the graph collection type to the graph item type through the graph head type. It backtracks data-flows from a node of an item/head type until a collection type is discovered. When a collection support library is used, all the operations are done on the support library type. K9 backtracks data-flows from a support library type to a collection/item type.

5.4 Indirect Dependency Analysis

An indirect dependency occurs when a thread gets an item from a collection, generates a new item from it, and adds the new item to another collection. To detect indirect dependencies, K9 keeps track of data-flows from the referenced item to the added item. K9 identifies three types of code location: 1) where a pointer to an item is obtained from a collection, 2) where another item is generated from the obtained item, and 3) where the generated item is added to another collection. K9 identifies the code locations of the first and third as direct dependencies. To discover the code locations of the second type, K9 performs the indirect dependency analysis.

To understand which item is created from which item, K9 uses the data-flow graph. When an item is referenced from a collection, there is an edge from a node of a collection type to a node of an item type. We call an item node that has an incoming edge from a collection node an item-source node. If an item is added to a collection, there is a data-flow edge from a node of an item type to a node of a
collection type. We name an item node that has an outgoing edge to a collection node as an item-destination node. If K9 finds a data-flow path from an item-source node to an item-destination node, it understands that the added item (item-destination) is generated from the referenced item (item-source). This analysis is performed for every combination of item-source and item-destination node. This naive reachability analysis does not scale because the data-flow graph becomes large and the cost of tracking data-flow paths is very high. For example, when we apply K9 to the Linux kernel, the data-flow graph contains 54,913 nodes and 51,983 edges.

To avoid the scalability problem, we introduce two heuristics. First, K9 simplifies a data-flow graph before it conducts the indirect dependency analysis. In particular, it combines different nodes of the same type into one node, called the type node. In so doing, multiple data-flow edges from the same type node to the same type node are unified into a single type-flow edge. This simplification reduces the graph size and makes the indirect dependency analysis scalable; the number of type nodes is reduced to 18,753, while the number of type-flow edges is reduced to 2,440 in the Linux kernel.

Even with this simplification, K9 does not overlook indirect dependencies because all the data-flow edges before the simplification exist in the type-flow graph. Note that this simplification increases false positives because logging code is inserted in data-flow edges which do not appear in a data-flow path from an item-source to an item-destination but do appear in the type-flow path.

For the second heuristic, we introduce the chain length, which specifies the maximum number typeflow edges along a typeflow path. It corresponds to the maximum number of type conversions from the source to the destination. By default, we set the chain length to four. From our experience in Linux, the number of type conversions is often less than three, and thus the chain length does not miss indirect dependencies in practice.

6. Evaluations

This section presents the results of an evaluation of K9 from the following viewpoints:

**Scalability:** We applied K9 to the Linux kernel. It successfully analyzed the kernel in a reasonable time.

**Accuracy:** K9 takes a best-effort approach to identify code locations that can cause inter-thread data dependencies. We manually estimated the precision of the log points (Sect. 6.2). We also demonstrate K9 provides useful clues for diagnosis. We reproduced two failures and one performance problem from existing bug reports and successfully diagnosed them. In addition, we used K9 to diagnose an unknown bug in the Btrfs (Sect. 6.3).

**Performance overheads:** The performance of the K9-enabled Linux was measured on the filebench varmail benchmark, MySQL, MongoDB, Apache web server, and Squid proxy server. The performance overheads were found to be less than 1.25% in terms of throughput, and the CPU usage increased by 0.18% on average (Sect. 6.4).

### 6.1 Scalability

We applied K9 to the Linux kernel (v4.1.39) in three use cases. In the first case, K9 analyzed the Linux kernel with the default configuration, and the range of the analysis was not limited. The second case supposed that the user wanted to diagnose failures related to file systems. Logging code was inserted at code locations reachable from system calls related to file systems. Since K9 builds a data-flow graph starting from them, the logging code was injected in the kernel core and the memory management as well as the Ext4 file system, the block I/O layer, and device drivers (SATA, SCSI, and so on). The third case supposed the user wanted to diagnose failures related to networking and the logging code is limited to networking. Table 1 describes the experimental environments.

Table 2 shows the result of the analyses. K9 analyzed 1,385,778 LOC comprised of 60,513 functions in 2,051 files in the default configuration. The direct and indirect dependency analyses took 44 min 54 s and 321 min, respectively. The indirect dependency analysis took much longer than the direct one because it required a reachability analysis. The patch was generated in 10 s. K9 identified 419 collection types and 386 item types. K9 detected 3,131 log points for direct dependencies and 1,286 log points for indirect dependencies. When the analysis was limited to the file system or networking, the code was about one-seventh the size of the default configuration and the analysis took only one-sixtieth the time.

#### 6.2 Precision of Log Points

We evaluated the precision of the log points identified by K9. In particular, we manually investigated the log points iden-
Table 3 Characterization of log points identified in the direct dependency analysis of the file system

| Item Type | Collection Type | # log points (TP / FP) | Description |
|-----------|-----------------|------------------------|-------------|
| mount     | mount           | 16 / 0                 | Graph: mount namespace in file system. |
| page      | pagevec         | 16 / 0                 | Array: page buffer and page cache. |
| audit_names | audit_context   | 0 / 15                 | Graph: audit context and its name. |
| request   | request_queue   | 15 / 0                 | Graph/Array: I/O request queue and I/O request. |
| blk_mq_hw_ctx | request_queue  | 0 / 13                 | Graph: software queue and hardware context in multi-queue block I/O. |
| perf_event | perf_event      | 12 / 0                 | Graph: perf events. |
| page      | address_space   | 11 / 0                 | Array: page cache tree and page cache. |
| cfq_queue | request         | 11 / 0                 | Array: I/O request and associated CFQ queue. |
| sidtab_node | sidtab         | 7 / 2                  | Array: SELINUX security identifier table and its node. |
| perf_event | perf_event_context | 9 / 0              | Graph: perf context and managed events. |
| mtd_dev   | mtddev          | 2 / 6                  | Graph: Linux multi device and its extended device (e.g., RAID). |
| scsi_device | scsi_host      | 7 / 0                  | Graph: SCSI object and its device. |
| page      | kmem_cache_node | 7 / 0                  | Graph: slab allocator and its slab. |
| scsi_cmd   | scsi_host      | 7 / 0                  | Graph: SCSI object and its command. |
| page      | page            | 6 / 0                  | Graph: LRU page list. |
| request   | deadline_data   | 6 / 0                  | Graph/Array: Deadline I/O scheduler and I/O request. |
| file_lock | file_lock       | 5 / 1                  | Graph: POSIX file lock. |
| inode     | backing_dev_info | 6 / 0            | Graph: low-level device information that contains the dirty inode list. |
| page      | lruvec          | 5 / 0                  | Graph: LRU page list. |
| perf_event_context | task_struct  | 5 / 0                  | Array: process scheduler attaches or detaches perf events to task structs. |
| page      | per_cpu_pages   | 5 / 0                  | Graph: per-cpu page list. |
| worker    | worker_pool     | 5 / 0                  | Graph: worker pool in the Linux kernel workqueue. |
| perf_event | cpu_hw_events   | 5 / 0                  | Array: perf events on hardware context. |
| task_struct | lb_env        | 5 / 0                  | Graph: load balancer and its task in the completely fair scheduler. |
| file_lock | inoide          | 5 / 0                  | Graph: POSIX file lock. |
| page      | zone            | 5 / 0                  | Graph: free page list. |
| hashtab_node | hashtab       | 0 / 5                  | Graph: SELinux hash table implemtation. |
| sched_entity | rt_prio_array | 0 / 5                  | Graph: request queue and its request in the real-time scheduler. |
| task_struct | rq            | 0 / 4                  | Graph: runqueue and task structure. |
| request   | blk_mq_hw_ctx  | 4 / 0                  | Graph: I/O request queue and I/O request in multi-queue block I/O. |
| …         | …              | …                     | …            |
| Total     |                 | 302 / 90              |             |

tified by K9 and classified them into correct (true positives) and incorrect (false positives). To reduce the burden of the manual investigation, we focused on the file-system configuration in Table 2. For each log point, we carefully examined the source code and determined whether it was correct or not, by using our expertise on the Linux kernel. The file-system configuration was chosen because one of the authors has intimate knowledge of the linux file system. In spite of our expertise, the correctness of some log points could not be determined with confidence. We regarded all of them as false positives.

Our evaluation did not treat true/false negatives. Counting true negatives would be nonsense because it means counting the code locations where logging code should not be injected and not by K9. For false negatives, we would need an oracle that tells us all the code locations in the linux kernel that can cause inter-thread data dependencies. Getting such an oracle is almost impossible, and thus, we did not address false negatives.

For the file-system configuration, as shown in Table 2, K9 identified 73 item types and 93 collection types. For the direct dependencies, 392 log points were identified, and Table 3 shows the true/false positives for the direct dependencies. For each category, Table 3 shows the numbers of the correct and incorrect log points and gives a brief description of the type pair. The top 30 results for direct dependencies are shown in terms of the number of correct log points. K9 identified 302 correct and 90 incorrect log points out of 461; 77.0% log points were correct, and 23.0% were incorrect.

For the indirect dependencies, K9 analyzed 19 item-source and item-destination pairs and detected 43 (62.3%) correct and 26 (37.7%) incorrect log points out of 69. Figure 10 shows two type-flow graphs for the two item-destination types (request and cfq_queue). Each node represents a type and boldface denotes the item-source type. An edge shows type conversion from a source node and a destination node, and its annotation indicates the number of correct/incorrect log points.

In our motivating example shown in Fig. 2, there is an indirect dependency between the page cache tree and the I/O request queue. An item of page is obtained from the page cache tree, from which an item of the request is initialized and inserted into the I/O request queue. This indirect dependency is captured by K9. Figure 10 (a) shows the type-flow paths from the page item type to request item type. K9 inserted 21 log points; 5 from page to buffer_head, 5 from page to bio, 1 from buffer_head to bio, 1 from bio to bio, and 9 from bio to request.

Now let us discuss false positives in K9. In the direct dependency analysis, K9 identified 90 incorrect log points...
(23.0%). There are two reasons for K9 inserting incorrect log points. First, K9 mistakenly identified shared data (16). K9 regarded the data allocated in the heap as shared data, but some were per-process data despite that they were allocated in heaps. For example, the audit context and audit names were detected as shared data, but both are per-process data.

Second, K9 regarded false dependencies as inter-thread data dependencies (34). Suppose one thread updates one field of a C struct and another thread reads another field of the same C struct. In this case, there is no dependency between the two threads. Since K9 keeps track of data flows in the granularity of C structs, not fields, it considers that there is an inter-thread dependency between those threads. The remaining 40 log points were hard to classify even with our expertise and thus were classified as false positives.

In the indirect dependency analysis, K9 identified 26 incorrect log points. All of the false positives were caused by the simplification described in Sect. 5.4. K9 combined different variable nodes of the same type into one node. Even if there are no data dependencies between two variables nodes, K9 considers there is if the nodes combined with them have data dependencies.

6.3 Diagnosing Failures

To demonstrate that K9 provides useful clues for failure diagnosis, this section reports four diagnosis cases. Despite that K9 gave false negatives, it provided useful information for debugging in failure diagnosis. Table 4 summarizes all four failures. Our report includes diagnosis of an unknown bug in Btrfs. This bug was reported to the Linux kernel community, and our patch fe01aa6 [13] has been accepted.

6.3.1 Diagnosing Known Bugs

Uncleared dirty bit: Btrfs forgets to clear the page’s dirty flag during error handling, which causes a kernel panic when the page is released later. This is a known bug (8146502 [14]).

CFQ priority violation: CFQ ignores priorities in the asynchronous writeback because priority is not propagated among the threads. This performance bug has been reported in previous research [15].

Data race in kernel workqueue: A race condition occurs when a thread tries to destroy a workqueue, which leads to a kernel panic. This bug is reported at fa2563e [31].

Uncleared writeback bit: Btrfs forgets to clear the writeback bit of a page when it fails to flush the page, which hangs up the kernel. This is a heretofore unknown bug.

Table 4 Diagnosed failures that are caused by three known bugs and one unknown bug.

| Bug Description                  | Log Points |
|---------------------------------|------------|
| Uncleared dirty bit             | 3          |
| CFQ priority violation          | 3          |
| Data race in kernel workqueue   | 3          |
| Uncleared writeback bit         | 3          |

Figure 10 Type-flow graphs showing the results of indirect dependency analysis. An edge is annotated with “true positives/false positive” that indicates the number of log points.

Figure 11 shows an excerpt of the code related to the kernel workqueue. In rmmod, destroy workqueue tries to free a kernel workqueue. It flushes all elements in the workqueue until it becomes empty (line S4~S10). After confirming it is empty and not used (line S11~S14), the workqueue is released (line S15). Since the workqueue is not locked in rmmod, the kworker thread can manipulate it.
Fig. 11 Kernel workqueue bug: data race on cwq->nr activate

Simultaneously with rmmod, kworker can requeue a new element, while the workqueue is being flushed. If this happens, cwq->nr_active is set to non-zero and the kernel panics at line S13.

Here, K9 detected the global cwq as a graph collection and the work_struct as its item. K9 inserted logging code at lines S22 and S27. Unfortunately, it did not insert any logging code at destroy_workqueue, because there is no collection access. Despite the lack of log information, the K9 log, shown in Fig. 11, is still helpful for diagnosing this failure. The log at line L3 informs of the rmmod crashes at line S13 in destroy_workqueue. Since there is no log in destroy_workqueue, the workqueue causing the failure cannot be identified. Looking at lines L1 and L2, we can see that the crash occurs after an element (work struct at 0x...a000) is flushed from and re-queued to the same workqueue (global cwq at 0x..c9c0). From this log, we can guess that a new item is added to a workqueue while it is being destroyed.

6.3.2 Diagnosing an Unknown Bug

We used K9 to diagnose a failure caused by an unknown bug. One of the authors is a community developer of Btrfs and encountered a situation in which sync command hangs. If there is a page cache whose writeback bit is activated, sync waits until the page cache is flushed and its bit is cleared. However, because of a bug in the code, the writeback bits are not cleared forever.

To diagnose this hang-up, we investigated the log generated by K9. Figure 12 depicts the failure logs and the related code. If there is a page cache whose writeback bit is activated, sync waits until the page cache is flushed and its bit is cleared. However, because of a bug in the code, the writeback bits are not cleared forever.

To diagnose this hang-up, we investigated the log generated by K9. Figure 12 depicts the failure logs and the related code. If there is a page cache whose writeback bit is activated, sync waits until the page cache is flushed and its bit is cleared. However, because of a bug in the code, the writeback bits are not cleared forever.

To diagnose this hang-up, we investigated the log generated by K9. Figure 12 depicts the failure logs and the related code. If there is a page cache whose writeback bit is activated, sync waits until the page cache is flushed and its bit is cleared. However, because of a bug in the code, the writeback bits are not cleared forever.

To diagnose this hang-up, we investigated the log generated by K9. Figure 12 depicts the failure logs and the related code. If there is a page cache whose writeback bit is activated, sync waits until the page cache is flushed and its bit is cleared. However, because of a bug in the code, the writeback bits are not cleared forever.
with page (0x...7180) as an argument, we guessed that sync waits forever for the page to be written back. To get the code location where the page is requested to be written back, we grep’d the log with 0x...7180 and discovered L1, which records the access to the same page at line S4 in extent_io.c by a.out. By checking the source code, we found that extent_write_cache_pages writes back each page in pvec.pages and clears the rewriteback bit if the rewriteback succeeds. But if the rewriteback fails, the error handler forgets to clear the writeback bit. As a result, if the writeback fails in submit_extent_page, the bit remains 1 and sync waits forever until the bit is cleared.

6.4 Performance Overheads

We measured the overheads of K9 on five macro benchmarks. In Table 5, Filebench-varmail is the mail server benchmark in filebench[32]. Tpcc-mysql runs the TPCC[33] workload on MySQL[34]. Ycsb-mongo is a NoSQL benchmark using MongoDB[35] with the YCSB workload[36]. For network applications, the performances of the Apache web server[37] and squid proxy server[38] were measured using apache benchmark[39]. The execution environment was the same as in Table 1. K9 was applied to system calls related to file systems in the first three benchmarks and to those related to networking in the remaining two benchmarks. Linux kernel v4.1.39 was used with the ext4 file system.

Before we focus on the detailed results, we briefly describe the summary of the results. In macro benchmarks, K9 incurs low logging overhead; e.g., 1.25% throughput degradation and 0.18% CPU usage increase on average. This is because disk I/Os and network I/Os become the bottleneck of the performance. However, in micro benchmarks which do not include I/Os, the logging overhead increases to 35.67%. The cause of the overhead increase is the lock contention in the LRU free page list and the process scheduling.

Table 5 shows the experimental results. The throughput slowdown and the CPU usage increase were measured for each benchmark. The throughput drop was from 0.53% in tpcc-mysql to 4.79% in ab-squid. The harmonic average slowdown was 1.25%. The CPU usage increase was from 0.04% in ab-apache to 0.41% in tpcc-mysql. The average increase was 0.18%.

To analyze the overhead, we measured the overhead of a microbenchmark, filebench-seqwrite, that generates dirty pages without writing them to disk. This benchmark highlights the overheads because it does not involve disk I/Os. The throughput deteriorates by 35.67% in this benchmark. Although one might expect that the CPU usage increases because of the logging overheads, it actually decreased by 0.34%. This is due to lock contention. If many logs are generated in a critical section, the execution time of the critical section increases because of the logging overhead. If the lock is held longer, other threads have to wait longer to acquire the lock, which results in a decrease in CPU usage.

To confirm that lock contention is one of the primary overheads, we counted the number of lock contentions and found that two locks (the one for LRU free page list and the other for process scheduling) are dominant in lock contents. If we remove two log points in the critical section for the LRU free page list, the throughput degradation decreases from 35.67% to 26.08%. If we also remove one log point for process scheduling, the degradation decreases to 24.50%. K9 users can reduce the logging overheads by specifying the code range to be logged; if they are not interested in process scheduling, the scheduling code can be excluded from the logging.

To analyze how much the overhead of K9 is sensitive to faster disks, we have evaluated disk I/O related benchmarks on a RAM disk whose performance is, in general, faster than other forms of storage media. As a result, the throughput is decreased by 3.77% in ycsb-mongo, by 3.07% in tpcc-mysql, and by 3.48% in filebench-varmail. The reason why the throughput degradation is smaller in the ycsb-mongo and tpcc-mysql is that the proportion of the execution of the user code is larger than filebench-varmail. Thus, the logging overhead of K9 becomes more acceptable in large applications.

7. Related Work

**Logging automation**: Deciding on appropriate log points and log messages is time consuming and requires expertise[2], [9], [10], [40]–[42]. Some studies [11], [12], [43] automatically insert logging codes. Errol[11] extracts error conditions from the source code that developers should insert log points and automatically adds the logging code. LogAdvisor[12] also automatically extracts common logging practices from existing logging code by machine learning and suggests log points to developers. Li et al. [43] enhance LogAdvisor to suggest the modification of existing logging code when it is changed, by using machine learning. Unlike K9, these studies are based on existing logging code in the source code and do not focus on inter-thread data dependency.

Log20 [44] is a fully automated logging tool and does not rely on existing logging code. Guided by information
theory, it computes how effective each log printing statement is in disambiguating code paths and automatically determines the placement of logging statements that maximize the calculated effectiveness under a specified overhead threshold.

K9 is also an automatic logging tool that does not need existing logging statements. K9 focuses on tracing inter-thread data dependencies, while Log20 aims to disambiguate executed code paths. Log20 automatically decides log points to disambiguate the executed code paths. Tracking inter-thread data dependencies is another clue with which to clarify the error status and determine root causes. Combining K9 with other tools helps developers in analyzing root causes.

**Logging library and Tracing framework:** NanoLog [45], ETW [46], Log4j2 [47], spdlog [48], glog [49] enable developers to specify arbitrarily formatted log statements in code. These tools are complementary to K9. Developers can get the information about where and what to log to trace inter-thread dependencies from K9. Google Dapper [50], Fay [51], DTrace [52], Pip [53], SystemTap [54], X-Trace [55], MagPie [56], and Pivot Tracing [7] help us to understand system behaviors across threads and machines by recording system runtime events. To trace the runtime information, they use manually inserted annotations or allow developers to write scripts or queries. K9 automatically infers the code locations that would cause data dependencies across threads. Our dependency model between the collection and the item accelerates such annotations; for example, the model can be used to detect code locations where the request queue and its request cause data dependencies across threads.

Robust interface: To increase system reliability, Healers [57] and Diagnosys [58] automatically generate wrapper functions for a user-level library and the Linux kernel, respectively. The wrapper functions include additional annotations that output useful information for debugging. These studies are based on the observation of common faults (e.g., deadlock and null pointer dereference) to decide the function to be wrapped and the content of the annotation. However, K9 focuses on tracing inter-thread data dependencies and automatically inserts logging code.

Tracing low-level shared memory access: To detect or debug concurrency bugs, some studies record the low-level shared memory access. There are two approaches to recording shared memory accesses. The software-only approaches [59], [60] record the shared memory access without hardware support. However, recording all shared memory operations incurs a high logging overhead in software-only approaches.

The hardware-associated approaches [27] record shared-memory operations by using dedicated hardware support. Gist [27] debugs concurrency bugs by using hardware watchpoints for tracing the data flow of the shared data. However, the hardware resource is limited (e.g., x86 has only four hardware watchpoints [61]). Therefore, hardware watchpoints are only suitable when the data to be monitored is known in advance.

K9 does not use hardware watchpoints to trace the shared memory access. This is because we do not know which shared data causes a failure in advance. K9 uses logs to trace the data accesses to the shared data that cause inter-thread data dependencies. To avoid high logging overheads, K9 does not record all shared memory operations. Instead, it focuses on the data accesses among the collection and the item that often cause the inter-thread data dependencies.

**Deterministic replay on multiple processors:** Deterministic replay is a useful technique in program debugging. In particular, replay on multiple processors [20], [62]–[67] is used to reproduce concurrency bugs. K9 is complementary to the deterministic replay tools. The logs of K9 provide debugging clues after failures are reproduced by deterministic replay.

Log Analysis: Many tools for log analysis [1]–[3], [6], [8] have been proposed. These tools make use of existing logging or messages. For example, Xu et al. [2] use machine learning to learn common patterns from console logs and detect abnormal log patterns that violate the common patterns. SherLog [1] is a debugger that combines the runtime logs and source code to reconstruct part of the failed execution path. Inferring information across threads is a future work of SherLog (Sect. 8 in [1]). Performance analysis and workload modeling in distributed systems [3], [6], [8] use log analysis to track end-to-end requests and for modeling workloads.

8. Conclusion

We described the complexities caused by inter-thread data dependencies. Motivated by our observations, we introduced a practical model for identifying these data dependencies. We then designed and implemented an automatic logging tool, K9, which, relying on the model, automatically inserts logging code for tracing the dependencies. We showed the effectiveness of K9 for software diagnosis. K9 causes only 1.25% throughput slowdown and 0.18% CPU usage increase in the real-world applications. Our hope is that K9 will facilitate and inspire programmers to implement future debuggers or profilers considering inter-thread dependencies.

**Acknowledgments**

This work was supported in part by the Japan Science and Technology Agency (JST CREST JPMJCR19F3) and the Japan Society for the Promotion of Science (JSPS KAKENHI JP16J03272 and JP16K00104). We thank the members of the HITACHI Yokohama Research Laboratory for their insightful comments. They gave us invaluable feedback on our proposal that greatly improved this paper.

**References**

[1] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy,
“SherLog: Error Diagnosis by Connecting Clues from Run-time Logs,” Proc. 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ‘10), New York, NY, USA, pp.143–154, 2010.

[2] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, “Detecting Large-Scale System Problems by Mining Console Logs,” Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP ’09), Big Sky, MT, 2009.

[3] X. Zhao, Y. Zhang, D. Lion, M.F. Ullah, Y. Luo, D. Yuan, and M. Stumm, “proof: A non-intrusive request flow profiler for distributed systems,” Proc. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’14), Broomfield, CO, pp.629–644, 2014.

[4] Q. Fu, J.G. Lou, Y. Wang, and J. Li, “Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis,” Proc. 2009 Ninth IEEE Int. Conf. Data Mining (ICDM ’09), 2009.

[5] J.G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, “Mining invariants from console logs for system problem detection,” Proc. USENIX Annual Technical Conference (ATC ’10), 2010.

[6] M. Chow, D. Meisner, J. Flynn, D. Peeck, and T.F. Wenisch, “The mystery machine: end-to-end performance analysis of large-scale internet services,” Proc. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’14), Broomfield, CO, pp.217–231, 2014.

[7] J. Mace, R. Roelke, and R. Fonseca, “Pivot tracing: Dynamic causal monitoring for distributed systems,” Proc. 25th ACM Symposium on Operating Systems Principles (SOSP ’15), Monterey, California, USA, 2015.

[8] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, “Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs,” Proc. 21th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’16).

[9] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Passupathy, “Improving software diagnosability via log enhancement,” ACM Trans. Comput. Syst., 2012.

[10] Q. Fu, J. Zhu, W. Hu, J.G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, “Where Do Developers Log? An Empirical Study on Logging Practices in Industry,” Proc. 36th Int. Conf. Software Engineering (ICSE ’14), Hyderabad, India, 2014.

[11] D. Yuan, S. Park, P. Huang, Y. Liu, M.M. Lee, X. Tang, Y. Zhou, and S. Savage, “Be conservative: enhancing failure diagnosis with proactive logging,” Proc. 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’12), Hollywood, CA, pp.293–306, 2012.

[12] J. Zhu, P. He, Q. Fu, H. Zhang, M.R. Lyu, and D. Zhang, “Learning to Log: Helping Developers Make Informed Logging Decisions,” Proc. 37th Int. Conf. Software Engineering (ICSE ’15), Florence, Italy, 2015.

[13] “Btrfs: add another missing end_page_writback on submit_extent_page failure,” 2 2017. https://github.com/torvalds/linux/commit/f01a6a65.

[14] L. Bo, “Btrfs: fix crash of btrfs_release_extent_buffer_page,” 9 2014. https://github.com/torvalds/linux/commit/814650.

[15] S. Yang, T. Harter, N. Agrawal, S.S. Kowsalya, A. Krishnamurthy, S. Al-Kiswany, R.T. Kaushik, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, “Split-Level I/O Scheduling,” Proc. 26th ACM Symposium on Operating Systems Principles (SOSP ’15), Monterey, California, USA, 2015.

[16] W. Landi, “Undecidability of static analysis,” ACM Lem. Program. Lang. Syst., vol.1, no.4, pp.323–337, 1992.

[17] G. Ramalingam, “The undecidability of aliasing,” ACM Trans. Program. Lang. Syst., vol.16, no.5, pp.1467–1471, 1994.

[18] Y. Sui, P. Di, and J. Xue, “Sparse flow-sensitive pointer analysis for multithreaded programs,” Proc. 2016 International Symposium on Code Generation and Optimization (CGO ’16), 2016.

[19] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” Proc. 2004 International Symposium on Code Generation and Optimization (CGO ’04), Palo Alto, California, 2004.

[20] A.J. Mashitizadeh, T. Garfinkel, D.T.D. Mazieres, and M. Rosenblum, “Towards Practical Default-On Multi-Core Record/Replay,” Proc. Twenty-Second Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS ’17), 2017.

[21] X. Wang, N. Zeldovich, M.F. Kaashoek, and A. Solar-Lezama, “Towards optimization-safe systems: Analyzing the impact of undefined behavior,” Proc. 24th ACM Symposium on Operating Systems Principles (SOSP ’13), New York, NY, USA, pp.260–275, 2013.

[22] B. Kasikci, C. Zamfir, and G. Candea, “Racemob: Crowdsourced data race detection,” Proc. 24th ACM Symposium on Operating Systems Principles (SOSP ’15), New York, NY, USA, pp.406–422, 2013.

[23] B. Hardekopf and C. Lin, “Semisparse Flow-Sensitive Pointer Analysis,” Proc. 36th ACM Symposium on Principles of Programming Languages (POPL ’09), Savannah, Georgia, USA, 2009.

[24] T. Zhang, D. Lee, and C. Jung, “Taxace: Efficient data race detection using commodity hardware transactional memory,” Proc. 21th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’16)th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’16), 2016.

[25] J. Mao, Y. Chen, Q. Xiao, and Y. Shi, “Rid: Finding reference count bugs with inconsistent path pair checking,” Proc. 21th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’16), 2016.

[26] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M.F. Kaashoek, “Improving integer security for systems with kint,” Proc. 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’12), Hollywood, CA, pp.163–177, 2012.

[27] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea, “Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-Production Failures,” Proc. 26th ACM Symposium on Operating Systems Principles (SOSP ’15), Monterey, California, USA, 2015.

[28] J. Huang, M. Allen-Bond, and X. Zhang, “Pallas: Semantic-aware checking for finding deep bugs in fast path,” Proc. 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’16)th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’17), 2017.

[29] R. Bodik and S. Anik, “Path-sensitive value-flow analysis,” Proc. 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL ’98), 1998.

[30] G. Ammons and J.R. Larust, “Improving data-flow analysis with path profile,” Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’98), 1998.

[31] T. Tuttle, “workqueue: lock cqw access in drain_workqueue,” 9 2011. https://github.com/torvalds/linux/commit/ff01a6a65.

[32] “Filebench.” http://filebench.sourceforge.net.

[33] “TPC-C.” http://www.tpc.org/tpcc/.

[34] “Mysql.” https://www.mysql.com/.

[35] “Mongodb.” http://www.mongodb.org/.

[36] “Yahoo! Cloud Serving Benchmark (YCSB).” https://github.com/brianfranke/cooper/YCSB/wiki.

[37] “Apache HTTP Server Project.” https://httpd.apache.org/.

[38] “Squid: Optimising web delivery.” http://www.squid-cache.org/.

[39] “ab - apache server benchmarking tool.” https://httpd.apache. org/docs/2.4/programs/ab.html.

[40] D. Yuan, S. Park, and Y. Zhou, “Characterizing Logging Practices in Open-Source Software,” Proc. 34th Int. Conf. Software Engineering (ICSE ’12), Zurich Switzerland, 2012.

[41] P. He, Z. Chen, and S.H.M.R. Lyu, “Characterizing the Natural Language Descriptions in Software Logging Statements,” Proc. IEEE/ACM Int. Conf. Automated Software Engineering (ASE ’18),
