INSPECTOR: Data Provenance Using Intel Processor Trace (PT)

INSPECTOR: Data Provenance using Intel Processor Trace (PT)

Jörg Thalheim, Pramod Bhatotia, and Christof Fetzer
TU Dresden

***Abstract*—Data provenance strives for *explaining* how the computation was performed by recording a trace of the execution. The provenance trace is useful across a wide-range of workflows to improve the dependability, security, and efficiency of software systems.**

In this paper, we present INSPECTOR, a `POSIX`-compliant data provenance library for shared-memory multithreaded programs. The INSPECTOR library is completely transparent and easy to use: it can be used as a replacement for the `pthreads` library by a simple exchange of libraries linked, without even recompiling the application code.

To achieve this result, we present a parallel provenance algorithm that records control, data, and schedule dependencies using a *Concurrent Provenance Graph* (CPG). We implemented our algorithm to operate at the compiled binary code level by leveraging a combination of OS-specific mechanisms, and recently released Intel PT ISA extensions as part of the Broadwell micro-architecture. Our evaluation on a multicore platform using applications from multithreaded benchmarks suites (PARSEC and Phoenix) shows reasonable provenance overheads for a majority of applications.

Lastly, we briefly describe three case-studies where the generic interface exported by INSPECTOR is being used to improve the dependability, security, and efficiency of systems. The INSPECTOR library is publicly available for further use in a wide range of other provenance workflows.

I. INTRODUCTION

A data provenance-aware system gathers and reports the lineage of execution. This allows the user to track, and understand, how the computation was performed. The provenance trace is useful for a wide-range of workflows to improve the dependability, security, and efficiency of software systems; including, program debugging [16], state machine replication [18], compiler optimizations [19], incremental computation [8], program slicing [31], memory management [22], and dynamic information flow tracking [34], etc.

More specifically, the data provenance trace provides an explicit intermediate program representation recording control and data dependencies for a program execution. Many existing systems provide support for data provenance (details in §IX); however, most existing solutions target sequential programs (or at the granularity of the entire process), while others that do support parallelism rely on restrictive application-specific programming model. As a result, the existing solutions have limited adoption in practice for the general shared-memory multithreaded programs.

In this paper, we propose an operating systems-based approach to data provenance for multithreaded programs. More specifically, we have the following three main design goals:

- Transparency: To support unmodified multithreaded programs without requiring any code changes to existing applications.
- Generality: To support the general shared-memory programming model with the full range of synchronization primitives in the `POSIX` API.
- Efficiency: To impose low overheads by designing the underlying provenance algorithm to be *parallel* as well so that it does not limit the available application parallelism.

To achieve these goals, we present INSPECTOR, a data provenance library for multithreaded programs. We implemented INSPECTOR as a dynamically linkable shared library. To run a program using INSPECTOR, the user just needs to preload the INSPECTOR library, and then, run the program as usual. Thus, our library supports existing binaries without any code changes or re-compilation. The library exports the provenance information to the `perf` utility as an extended interface.

Our high level approach is based on recording data, control, and schedule dependencies in a computation by constructing a *Concurrent Provenance Graph* (CPG). The CPG tracks the input data to a program, all sub-computations (a sub-computation is a unit of the computation), the data flow between sub-computations, intra-thread control flow, and inter-thread schedule dependencies for the multithreaded execution.

In this paper, we present a *parallel* algorithm to build the CPG. Our algorithm leverages the Release Consistency (RC) memory model [17] to efficiently record the inter-thread data and schedule dependencies in a completely decentralized manner. We implemented our algorithm as a dynamically linkable shared library by leveraging process-level isolation, MMU-assisted memory tracking, and Intel PT ISA extensions, released recently as part of the Broadwell micro-architecture. Furthermore, we extended the library to support a consistent snapshot facility, where the user can analyze the provenance on-the-fly while the program is still running.

In particular, we make the following contributions:

- We present a parallel algorithm for data provenance for multithreaded programs that records control, data, and schedule dependencies using a Concurrent Provenance Graph (CPG) (§IV).
- We implemented our algorithm as a dynamically linkable shared library, which we call INSPECTOR, leveraging MMU-assisted memory tracking, process-level isolation, and Intel PT ISA extensions. The INSPECTOR library can be loaded and linked at run-time as a replacement to the `pthreads` library, without any recompilation of the application code (§V).
- We further extended the library to support a live snapshot facility, where the user can analyze the

provenance on-the-fly while the program is still running. The library periodically takes a consistent snapshot [15] of the CPG in a decentralized fashion (§VI).

We empirically demonstrate the effectiveness of INSPECTOR by applying it to applications of PARSEC [12] and Phoenix [33] benchmark suites. Our experiments show that INSPECTOR incurs reasonable overhead to record data provenance for a majority of applications (§VII).

Furthermore, we briefly describe three on-going projects where the generic provenance interface exported by INSPECTOR is being used to improve the dependability, security, and efficiency of software systems (§VIII). INSPECTOR is an active open-source project and the library is publicly available to the research community for further use in other workflows.

II. OVERVIEW

We base our design on `POSIX` threads, commonly referred to as `pthreads`, a widely used threading library for shared-memory multithreading with a rich set of synchronization primitives.

Basic approach. At a high level, we record data provenance for a multithreaded execution by constructing a *Concurrent Provenance Graph (or CPG)*. Informally, the CPG records three types of dependencies; namely, control, data, and schedule dependencies for the multithreaded execution. To record these dependencies, we divide thread execution into sub-computations. We record the execution trace to construct the CPG that tracks the *data flow* between the sub-computations, *control flow* for each thread execution, and threads interleaving or *schedule dependency* in the multithreaded execution.

More specifically, the Concurrent Provenance Graph (or CPG) records a partial order $O = (N, \rightarrow)$ among sub-computations with the following property: given a sub-computation n (where $n \in N$) and the subset of sub-computations M that precede it according to $\rightarrow$, i.e., $M = \{M \subset N \mid \forall m \in M, m \rightarrow n\}$, if the writes made by m becomes visible to n then the partial order $\rightarrow$ captures this possible data flow between sub-computations.

Example. Using a simple example (shown in Figure 1), we next explain how we record these dependencies for a shared-memory multithreaded program. The example considers a multithreaded execution with two threads (T_1 and T_2) modifying two shared variables (x and y) using a lock. In the example, we assume that a thread execution is divided into sub-computations at the boundaries of synchronization primitives, such as `lock()/unlock()`. (We explain the reason behind this design choice in §III.) We identify these sub-computations as $T_1.a$ and $T_1.b$ for thread T_1, and $T_2.a$ for thread T_2. To understand the dependencies that need to be recorded for the required partial order ($\rightarrow$), we showcase three cases for recording the control, schedule, and data dependencies.

The first dependency that we need to record is the *control flow* execution of each thread. In particular, we need to record the intra-thread execution order of sub-computations. For

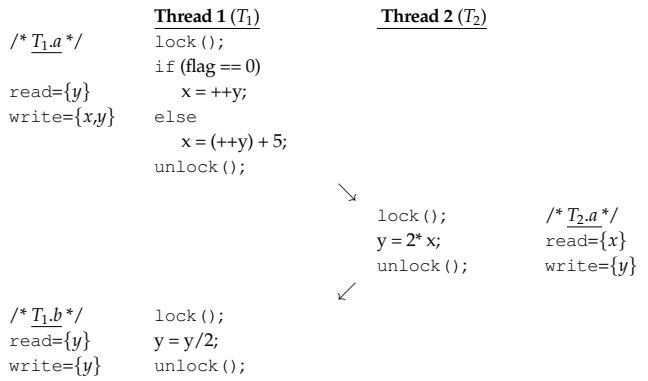

Figure 1: An example of shared-memory multithreading.

example, sub-computation $T_1.b$ follows $T_1.a$, and therefore, the control flow dependency records this partial order as $T_1.a \rightarrow T_1.b$. Additionally, we need to record the control flow path taken within a sub-computation. For example, sub-computation $T_1.a$ has a conditional branch (`if/else`) based on the value of `flag`. We supplement the control flow dependency with all control paths taken by a thread within each sub-computation; i.e., all branches taken at run-time.

Secondly, we need to record the inter-thread *schedule* dependencies. The sub-computations can be interleaved in different order across executions because of the non-deterministic thread scheduling by the underlying OS. For instance, when threads acquiring the lock in the reverse order where thread $T_1.b$ gets to acquire the lock before $T_2.a$. In this case, the final value of y is affected based on this new ordering. Therefore, we also need to record the schedule dependencies between sub-computations as part of the partial order. We record these schedule dependencies by tracking interleaving of sub-computations by recording the thread schedule (For example, $T_1.a \rightarrow T_2.a \rightarrow T_1.b$).

Lastly, we need to record *data dependencies* between sub-computations as a part of the partial order. For that, we track read and write sets for each sub-computation, i.e., the set of memory locations read or written by the sub-computation, respectively. The data dependencies are recorded implicitly using read and write-sets, and the partial order recorded using the control and schedule dependencies: if we know what data is read and written by each sub-computation, we can determine whether a data dependency exists by following the partial order, i.e. if a sub-computation is *transitively* reading the data that was modified by a sub-computation that precedes it in the partial order $\rightarrow$ then there exists a read-after-write data dependency.

III. SYSTEM MODEL

Before we formally describe the provenance algorithm (§IV), we first present the system model assumed by INSPECTOR.

Memory consistency model. Our approach relies on the use of the Release Consistency (RC) memory model [17], which requires that all shared memory accesses are done via synchronization primitives. For our purposes, this model

has the critical benefit of allowing us to restrict inter-thread communication (i.e. shared memory accesses) to the synchronization points. By reducing the number of points in an execution where inter-thread communication can occur, we avoid having to track individual `load/store` instructions, which would be extremely inefficient with current hardware.

Note that the RC memory model is weaker than, for example, the Sequential Consistency model (SC) [23], but still guarantees correctness and liveness for applications that are data race free. In fact, the semantics provided by INSPECTOR is as restrictive as the POSIX specification [1], which mandates that all accesses to shared data structures must be properly synchronized using `pthreads` synchronization primitives.

Synchronization model. We support the full range of synchronization primitives in the `pthreads` API, including *mutexes*, *cond_wait/cond_signal*, *semaphores*, and *barriers*. However, due to the weakly consistent RC memory model, our approach does not support *ad-hoc synchronization mechanisms* such as user-defined spin locks.

IV. DESIGN

In this section, we first formally define the CPG (§IV-A), and then present the algorithm to build the CPG (§IV-B).

A. Concurrent Provenance Graph

We define the *Concurrent Provenance Graph* (CPG) as a directed acyclic graph $G = (V, E)$ with vertices (V) and edges (E). The vertices of the CPG represent *sub-computations*. The edges represent the dependencies between the sub-computations.

Sub-computations. We define a *sub-computation* as the sequence of instructions executed by a thread between two `pthreads` synchronization API calls. We further divide each sub-computation as sequence of code thunks, or *thunks* to record the control path taken by the executing thread within the sub-computation.

Dependencies. We distinguish between three kinds of dependencies: control, synchronization, and data dependencies. We next described these dependencies.

I: Control edges. *Control edges* are used to record the intra-thread causal order between sub-computations of the same thread based on their execution order. Furthermore, we also record all control path taken by the executing thread within each sub-computation during the execution at the granularity of thunks.

We model the execution of thread t as a sequence of sub-computations (L_t). Sub-computations in a thread are totally ordered based on their execution order using a monotonically increasing thunk counter (α). We refer a sub-computation of thread t using the counter α as an index in the thread execution sequence (L_t), i.e., $L_t[\alpha]$.

We refer a thunk ($L_t[\alpha].\Delta$) as a sequence of instructions between two successive branches within each subcomputation. We denote a thunk of sub-computation $L_t[\alpha]$ using a counter β as an index in the sub-computation as $L_t[\alpha].\Delta[\beta]$.

II: Synchronization edges. *Synchronization edges* are used to record the inter-thread causal order between sub-computations based on the synchronization order between threads. We derive synchronization edges based on the ordering of synchronization operations (also known as a *sync schedule*). In particular, we build on the observation that synchronization primitives can be modeled as *acquire* and *release* operations. That is, during synchronization, the synchronization object is *released* by one set of threads and subsequently *acquired* by a corresponding set of threads blocked on the object [16]. For example, an `unlock(S)` operation releases S and a corresponding `lock(S)` operation acquires it.

We derive the partial order based on the happens-before relation ($\rightarrow$) [16, 32] between acquire and release operations. In particular, a release operation happens-before the corresponding acquire operation. Formally, two sub-computations $L_{(t_1)}[\alpha_1]$ and $L_{(t_2)}[\alpha_2]$ are ordered by the happens-before relationship ($L_{(t_1)}[\alpha_1] \rightarrow L_{(t_2)}[\alpha_2]$) if: (*i*) they are sub-computations of the same thread ($t_1 = t_2$), and $L_{(t_1)}[\alpha_1]$ was executed before $L_{(t_2)}[\alpha_2]$; (*ii*) $L_{(t_1)}[\alpha_1]$ is a release and $L_{(t_2)}[\alpha_2]$ is corresponding acquire on the same synchronization object S; (*iii*) due to transitivity if $L_{(t_1)}[\alpha_1] \rightarrow L_{(t_3)}[\alpha_3]$ and $L_{(t_3)}[\alpha_3] \rightarrow L_{(t_2)}[\alpha_2]$.

III: Data-dependence edges. *Data dependence edges* records the flow of data between sub-computations of the same or different threads. We derive the data dependencies between sub-computations using the read/write sets, and recorded partial order in control and synchronization edges. For a sub-computation $L_t[\alpha]$, the *read-set* ($L_t[\alpha].R$) and the *write-set* ($L_t[\alpha].W$) are the set of addresses that were respectively read from and written to by the thread while executing the sub-computation.

Essentially, data dependence edges establish the *update-use relationship* between sub-computations. The update-use relationship exists between two sub-computations if they can be ordered based on the happens-before relationship, and the write-set of the precedent sub-computations transitively intersects with the read-set of the antecedent sub-computations.

B. Provenance Algorithm

At high-level, our algorithm records the multithreaded execution to construct the CPG. Algorithm 1 presents the overview of the provenance algorithm, and details of the subroutines are presented in Algorithm 2.

Overview. The provenance algorithm (shown in Algorithm 1) is executed by all threads in parallel. During a thread execution, the thread traces memory accesses on `load/store` instructions, and adds them to the read and the write set of the executing sub-computation for deriving data dependencies. Additionally, the executing thread traces all branch instructions, and adds this information for thunks of the executing sub-computation to record control dependencies. The thread continues to execute instructions until a synchronization primitive call is made to the `pthreads` library. At the synchronization point, we define

Algorithm 1 Data provenance algorithm

```
∀S,∀i ∈ {1,...,T} : C_S[i] ← 0; // All sync clocks set to zero
executeThread(t)
begin
    initThread(t);
    while t has not terminated do
        startSub-computation(instruction);
        repeat
            Execute instruction of t;
            if (instruction is load or store) then
                onMemoryAccess(instruction);
            end
            if (instruction is branch then
                onBranchAccess(instruction);
            end
        until t invokes synchronization primitive;
        α ← α+1; // Increment sub-computation counter
        // Let S denote invoked synchronization primitive
        onSynchronization(S);
    end
end
```

Algorithm 2 Subroutines for the provenance algorithm

```
initThread(t)
begin
    α ← 0; // Initializes sub-computation counter (α) to zero
    ∀i ∈ {1,...,T} : C_t[i] ← 0; // t's clock set to zero
end
startSub-computation(instruction)
begin
    β ← 0; // Initialize thunk counter
    L_t[α].Δ[β] ← instruction; // Start new thunk
    C_t[t] ← α; // Update thread clock with sub-computation counter (α) value
    // Set sub-computation clock value to thread t's clock
    ∀(i ∈ {1,...,T}) : L_t[α].C[i] ← C_t[i];
end
onMemoryAccess(instruction)
begin
    // Update read/write sets of the executing sub-computation
    if instruction is load then
        L_t[α].R ← L_t[α].R ∪ {pageID}; // On read access
    else
        L_t[α].W ← L_t[α].W ∪ {pageID}; // On write access
    end
end
onBranchAccess(instruction)
begin
    β ← β+1; // Increment thunk counter
    L_t[α].Δ[β] ← instruction; // Add a new thunk
end
onSynchronization(S)
begin
    switch Syncronization type do
        case release(S):
            // Update S's clock to hold max of its and t's clocks
            ∀i ∈ {1,...,T} : C_S[i] ← max(C_S[i],C_t[i]);
            sync(S); // Perform the synchronization
        case acquire(S):
            sync(S);
            // Update t's clock to hold max of its and S's clocks
            ∀i ∈ {1,...,T} : C_t[i] ← max(C_S[i],C_t[i]);
    end
end
```

the end point for the executing sub-computation. Thereafter, we let the thread perform the actual synchronization operation. At synchronization points, the algorithm derives control and synchronization edges at the granularity of sub-computation by recording the happens-before order between sub-computations. Finally, we start a new sub-computation and repeat the process until the executing thread terminates.

Details. For the CPG, control and synchronization dependencies are derived by happens-before ordering of sub-computations. To do so, we use vector clocks (C) [27], a widely used mechanism to generate a partial order of events and to infer causality. Our use of vector clocks is motivated by its efficiency for recording a partial order between sub-computations in a complete decentralized manner instead of having to serialize all synchronization events in a total order.

In particular, each thread maintains a vector clock, i.e., an array/vector of size equal to the number of threads in the system. During a synchronization event, the clock of the thread performing the acquire operation is updated based on the clock value of the thread performing the release operation. More precisely, the vector clock is updated as follows: if a thread t_2 acquires the synchronization object S released by a thread t_1, then each entry in t_2's vector is updated to hold the maximum of its old value and the corresponding value of t_1's vector at the moment of release.

To implement this mechanism, our algorithm maintains vector clocks for three kinds of entities: threads, synchronization objects, and sub-computations. A *thread clock* (C_t) for a thread t tracks the local logical time of the thread, which is incremented each time a new thunk is created. A *synchronization clock* (C_S) for a synchronization object S acts as a messaging medium between threads synchronizing on S to update the thread clock. Finally, a *sub-computation clock* ($L_t[\alpha].C$) determines the position of the sub-computation $L_t[\alpha]$ in the CPG, and is set to the clock value of the thread while executing the sub-computation .

Based on the intuition developed so far, we next present the subroutines used in the recording algorithm (see Algorithm 2). Let T denote the number of threads in the system, which are numbered from 1 to T. Initially, each thread t initializes (using routine `initThread`(t)) its monotonically increasing thunk counter (α) and the thread clock (C_t) to zero. In addition, vector clocks (C_S) of all synchronization objects S are also initialized to zero. In the beginning of a new thunk (using routine `startSub-computation()`), the clock value (C_t) of the thread t is updated based on the sub-computation counter (α) to keep track of the local logical time of t. The thread clock is updated by assigning the α to t^{th} index of the thread clock $C_t[t]$. The updated value of thread clock (C_t) is also assigned to the sub-computation's clock ($L_t[\alpha].C$). Finally, the read set and the write set ($L_t[T_t].R/W$) of the new sub-computation are initialized to empty set.

During a sub-computation execution, we trace reads and writes (using routine `onMemoryAccess()`) at the granularity of the memory pages (`pageID`), and update the respective read/write set ($L_t[T_t].R/W$) of the executing sub-computation.

Similarly, we also trace branch instructions (using routine `onBranchAccess()`), and update the thunk within the executing sub-computation.

At synchronization points, we define the end of the

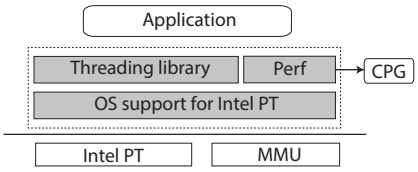

Figure 2: INSPECTOR architecture.

current sub-computation, and therefore, we increment the sub-computation counter (α) by one. The executing thread performs the synchronization operation (using routine `onSynchronization()`). Recall that in our model, a synchronization operation is either a release or an acquire operation. Therefore, we handle `onSynchronization()` accordingly. If it is a release operation on the synchronization object S by the thread t, the releasing thread updates the synchronization object's clock (C_S) to hold the maximum of its own clock value (C_t) and the clock (C_S) of S. Then the releasing thread performs the actual release operation on object S. Alternatively, if its an acquire operation then the acquiring thread first performs the acquire operation on object S. After the acquire operation on the synchronization object S by thread t, the acquiring thread updates its own clock (C_t) to hold the maximum of the clock value (C_S) of S and its own clock value (C_t). In this way, the synchronization clock (C_S) acts as a propagation medium to pass the vector clock value from the thread doing the release to the thread doing the acquire operation.

In the end of the provenance algorithm, all sub-computations (along with their read/write sets) have a recorded value of sub-computation's vector clock ($L_t[\alpha].C$). The standard comparison of vector clocks defines the happens-before partial order, through which causal order is derived between sub-computations.

V. IMPLEMENTATION

This section describes the architecture and implementation of INSPECTOR. We implemented INSPECTOR as a dynamically linkable shared library for the GNU/Linux OS that can be loaded and linked at runtime for `POSIX` threads (replacing the `pthreads` library). The application executables can simply link the library (without any recompilation) either using `LD_PRELOAD` or the `-rdynamic` flag, specifying the path of the INSPECTOR library. The INSPECTOR library exports the CPG as an extended interface in the `perf` utility for supporting data provenance. The architecture of INSPECTOR (shown in Figure 2) consists of two main components: threading library (§V-A) and OS support for Intel PT (§V-B). We next describe these two components in detail.

A. Threading Library

The threading library derives the data and schedule dependencies. The architecture of the threading library is shown in Figure 3.

Memory protection. A central challenge of the implementation of the algorithm is keeping track of the data dependencies for the shared-memory accesses by all possible interleaving

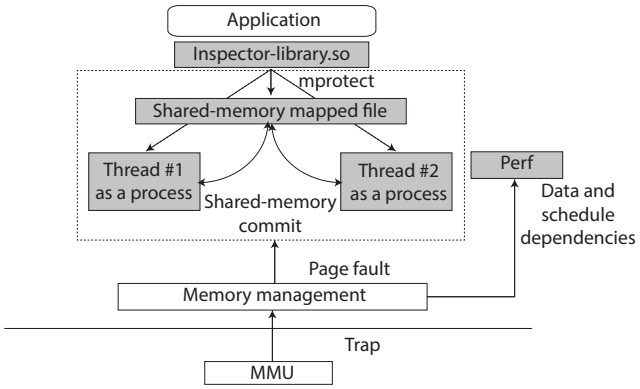

Figure 3: Architecture of the threading library.

threads. Since monitoring every load and store to each memory word would be too costly, we instead rely on the OS's (hardware-assisted) segmentation fault mechanism to keep track of reads and writes at the granularity of memory pages.

To derive the read and write sets during sub-computation execution, INSPECTOR uses standard memory protection mechanism and signal handlers. In particular, INSPECTOR protects the address space using `mprotect(PROT_NONE)` at the beginning of each sub-computation. This forces a trap (and the corresponding OS signal) the first time a page is read or written to in a given sub-computation. The respective signal handler, which is implemented by the INSPECTOR library, records the information about the access, and also resets the protection bits so that subsequent accesses to the same page by the same thread in the same sub-computation can proceed without generating a trap.

However, a naive page protection mechanism raises an important problem because all threads in a process share the same virtual memory structures (namely the TLB and page table entries with the respective protection bits). This makes it difficult to keep track of which threads are responsible for which memory accesses or to enforce different protections for different threads. Otherwise, we need to re-protect the page after serving every load and store instruction causing a large number of segmentation faults. To address this problem, INSPECTOR implements threads as separate processes (an idea proposed by Grace [4] and Dthreads [25]).

Threads as processes. INSPECTOR implements threads as separate processes thus allowing each thread has its own private address space and control over the virtual memory structures. This gives us the ability to manipulate the page protection of threads individually while providing a simple way to implement the Release Consistency (RC) memory model. In particular, INSPECTOR uses the `clone` system call to fork off a new process on `pthread_create()`. The process that implements the newly created thread (i.e., the child process) already shares parts of the execution context with the parent process (which implements the calling thread) such as file descriptors and signal handlers.

But this raises a new problem, which is that, unlike threads, processes do not share their address spaces. We address this by taking advantage of the RC memory model we defined

for INSPECTOR, where threads share the updates only at the synchronization points.

Shared memory commit. To implement the RC memory model, we use shared memory commit (originally proposed in distributed shared memory architectures such as TreadMarks [20] and Munin [14]) that allows threads to communicate at well-defined synchronization points. Our shared memory commit is implemented using memory mapped files. In particular, the virtual address ranges for the shared portions (globals and heap) of the address space are mapped to memory mapped files, which are managed by the INSPECTOR library. These address ranges correspond to the heap and the static (i.e., globals) regions. During thread creation, INSPECTOR marks these address ranges as a private copy-on-write mapping (using `MAP_PRIVATE` in `mmap()`). The effect of this is that whenever the child thread tries to write to a memory location, the OS makes a thread-private copy of the memory page containing the modification. At synchronization points, the thread computes a *diff* for each dirty page by performing a byte-level comparison between the dirty page and the shared page. The deltas are then atomically copied to the shared memory page; if there are overlapping writes to the same memory location we resolve them using a last-writer wins policy.

Input support. In addition to providing wrappers for `pthreads` and `malloc` related API calls, we also implemented shim layer for a number of input `glibc` library calls to record the data-flow from the input. For instance, we provide wrappers for `mmap` for reading the input. In particular, the threading library differentiates between the `mmap` calls made by the library itself and the target application. This allows us to record the mapping of the input file in the input address space. And, as described before, the library uses `mprotect()` to derive the data flow from the input.

B. OS Support for Intel PT

To obtain the control flow dependencies, we use Intel Processor Trace (PT) ISA extensions. We next present the implementation details of the OS support for Intel PT.

Intel Processor Trace (Intel PT). Intel PT is an extension of Intel Architecture that logs information about software execution with minimal performance impact. The processor collects information such as control flow, execution modes and timings and formats it into highly compressed binary packets. Traditionally, Intel architectures provided Branch Trace Store (BTS) for tracing branch execution. However, BTS was slow and imprecise. Therefore, it was not adopted in practice. To overcome the limitations of BTS, Intel recently introduced PT ISA extensions as part of the Broadwell (also available in Skylake) micro-architecture.

OS support. The Intel PT tracing facility is integrated into the operating system, which makes it possible to use different trace buffers for different processes, and to make the facility available for non-root users. In Linux this processor feature is exposed to the user-space as a Performance Measuring Unit (PMU) in the `perf` event interface. We make use of the Intel PT PMU to derive the control flow dependencies. Figure 4 shows the architecture for the OS support for Intel PT.

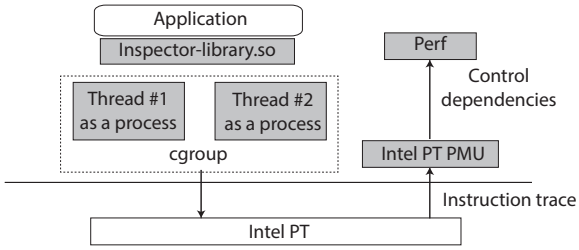

Figure 4: Architecture of the OS support for Intel PT.

In particular, the `perf` interface on Linux consists of a syscall, which gives back a file descriptor. Events are accessed by obtaining buffers via `mmap(2)` and can be further controlled via `ioctl()` syscall on the given file descriptor. Along with interface the user-space `perf` allows to dump and filter from these buffers. In our case, this filtering is done by using Linux control groups (also known as `cgroups`). `cgroups` is a kernel feature to apply constraint like resource usage to a group of processes. It has the property, that by default every child process belongs to the same process as its parent. Also for `perf_events` such a `cgroup` exists.

We create such a `cgroup` exclusively for the application using INSPECTOR. This is done because our threading library causes applications using threads to create multiple processes instead, whose process ids are not known in advance.

The subcommand `perf record` is then used to dump the trace produced by Intel PT. Intel PT generates a stream of TNT packets, which denotes the conditional branches taken and TIP packets for indirect branches and function returns. The data is referenced as a sample event in the `perf` event list and stored in a ring buffer called *AUX area*. If perf tool cannot keep up with processor trace it is possible (for example an interrupt occurs), there will be gaps in the trace. (We provide a snapshot facility (§VI) to overcome this limitation.)

After execution the result can be further processed by using a set of tools for example `perf script`. The branch information is still in a compressed form and needs to be decoded. We make use of the Intel Processor Decoder Library for Intel PT that is integrated in the `perf` utility. To map the trace onto binaries, it needs access to executables and linked libraries of the application. For that, we track mmap events to know the location of each loadable during the execution.

VI. SNAPSHOT MECHANISM

An additional challenge that we need to address in the implementation of INSPECTOR is to deal with the excessive log data produced by Intel PT, especially for long running applications. Therefore, we further extend the library to support a live snapshot facility, where the user (or an application using INSPECTOR) can analyze the provenance on-the-fly while the program is still running. Thus, the snapshot facility provides a practical alternative to restrict the space overheads imposed for storing the CPG.

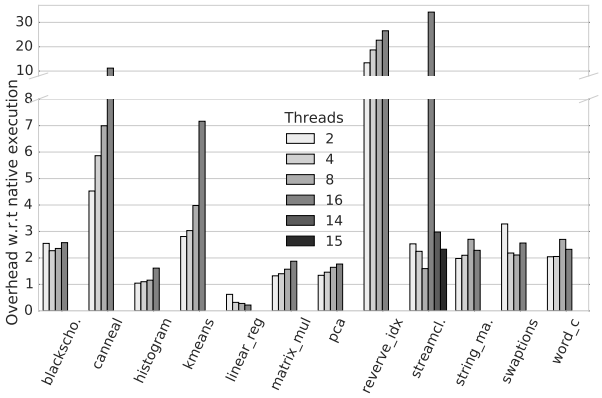

Figure 5: Performance overhead over the native execution with increasing number of threads. The corresponding work measurement plot is available here: web-link.

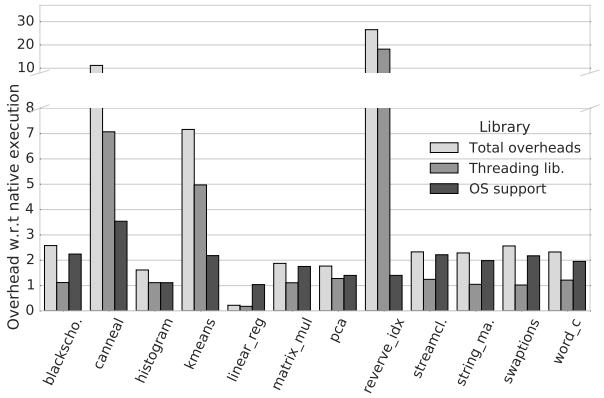

Figure 6: Performance overheads breakdown with 16 threads—except for *streamcluster*, where the breakdown for 15 threads is shown. The corresponding work measurement plot is available here: web-link.

For the snapshot facility, the library periodically takes a consistent cut of the CPG. A cut is *consistent* if, for any synchronization operation on object S in the trace, *acquire*(S) operation being in the cut implies that corresponding *release*(S) is also included in the cut [15]. To achieve so, we make use of modeling synchronization primitives as *acquire* and *release* operations (described in §IV). Each thread invokes the snapshot operation on the latest synchronization event (*acquire* or *release*) in the recorded trace.

We implemented the consistent cut facility using Intel PT interface for `perf`, which provides mechanism for the full trace, and a snapshot mode. When the full trace is enabled then the kernel does not overwrite the data that the user-space has not collected yet. This results in gaps in the trace, if the user-space process is not fast enough in collecting the log data. Whereas, in the snapshot mode, however, the old data in this ring buffer is constantly overwritten so that an application can start and stop tracing around a certain event. The `perf` tool exposes this feature by installing a handler on signal `SIGUSR2`, which triggers the start of a trace. INSPECTOR makes use of the signal and forwards it to `perf` to record a consistent snapshot of the trace based on the aforementioned checkpointing mechanism. Using this signal, we implemented a simple ring buffer with a configurable number of slots (each slot size is set to 4MB). As the user (or the application using INSPECTOR) finishes the live analysis on the recorded snapshots of the CPG, we reuse those slots for storing the new incoming snapshots of the CPG.

VII. EVALUATION

In this section, we present an experimental evaluation of INSPECTOR based on the implementation described in §V. Our evaluation answers the following questions.

- What performance overheads does INSPECTOR impose for recording the provenance graph? (§VII-A)
- What are the sources for these overheads? (§VII-B)
- How do these overheads scale with increase in the size of the input data? (§VII-C)
- What are the space overheads for the CPG? (§VII-D)

Experimental platform. We used an Intel Xeon processor based on Broadwell micro-architecture as our host machine. The host system consists of 8 cores (16 hyper-threads) of Intel(R) Xeon(R) CPU Processor D-1540 (12M Cache, 2.00 GHz) and 32 GB of DRAM main memory. The host machine is running Linux with kernel 4.3.0 in 64-bit mode.

Applications and dataset. We evaluated INSPECTOR using applications from two multithreaded benchmark suites: Phoenix 2.0 [33] and PARSEC 3.0 [12]. Table 7 lists the applications used for the evaluation along with the input data and benchmark parameters.

Performance metrics: Time and Work. For each run, we consider two types of measures: *time* and *work*. Time refers to the amount of (end-to-end) run-time to complete the parallel computation. Work refers to the total amount of computation performed by all threads and is measured as the overall CPUs utilization for all threads.

Measurements. All applications were compiled using GCC 5.2.1 compiler with `-o3` optimization flag. For all measurements, we report the average over 6 runs with minimum and maximum values discarded (truncated mean).

We measured work and time numbers for both `pthreads` and INSPECTOR executions with the same number of threads. For time measurements, we report the run-time comparison between the native `pthreads` execution, and INSPECTOR execution. To measure work, we used the CPU accounting controller in `cgroups` to account the CPU usage of all threads.

Finally, the log produced by `perf` was written to `/tmp` on `tmpfs` to allow high throughput.

Additional results. Due to the space limitation, the work measurements are covered in a technical report [35] and also available here: web-link.

A. Performance Overheads for Data Provenance

First, we explain the provenance overheads imposed by INSPECTOR w.r.t. the native `pthreads` execution. Figure 5 shows the provenance overheads of INSPECTOR w.r.t. the

native `pthreads` execution with varying number of threads (from 2 to 16 threads). As expected, the provenance overheads increases with the increase in the number of threads. This is because the shared memory commit (§V-A) takes longer time with a higher number of threads, as each thread spends less time computing on the input data.

The experiment shows that the provenance overheads using INSPECTOR vary across applications. We observe that a majority of applications (9/12) have a reasonable overhead between 1× up to 2.5× w.r.t. the native execution. However, three applications have exceptionally high overheads: *canneal*, *reverse_index*, and *kmeans*. The high overheads is explained as follows: *canneal* modifies a lot of memory pages that leads to a high number of page faults for deriving read and write sets (see Table 7). Whereas, *reverse_index* does a lot of small memory allocations across threads leading to a large number of segmentation faults (details omitted — see web-link). Finally, *kmeans* creates more than 400 threads until the clusters co-efficient converges, when we specify 500 as the parameter for the iterative convergence algorithm (see Table 7). Since, creating a process takes more time than creating a thread, we see a slowdown in *kmeans*.

On the other hand, *linear_regression* performs better than `pthreads`, which is explained by the fact that our implementation of threads as processes (§V) avoids false sharing, as previously noted by Sheriff [24], which leads to improved performance.

Lastly, in the case of *streamcluster*, we were limited by our physical memory to store the log in `tmpfs` for 16 threads (see §VII-D). Therefore, we also show the overheads with 14 and 15 threads, where the provenance log can fit into the main memory. To better understand the breakdown of provenance, we chose 15 threads for *streamcluster* in §VII-B.

B. Performance Overheads Breakdown

Next, we investigated the breakdown of the provenance overheads. Recall that our system implementation has two major components: (1) the threading library (§V-A), and (2) the OS support for Intel PT (§V-B). Figure 6 shows the breakdown of overheads with 16 threads normalized to the native `pthreads` execution. We quantify the breakdown as the time taken by the threading library and the OS support for Intel PT. The result shows an interesting pattern: the applications with unreasonably high overheads (*canneal*, *reverse_index*, and *kmeans*) spend a majority of time in the threading library for the above mentioned reasons. Whereas, the overheads for tracing the control flow due to Intel PT is a dominant factor for the other applications. These results highlight that for a majority of applications (9/12) the underlying hardware is still a bottleneck to achieve low provenance overheads.

C. Scalability with the Input Data

In addition to scalability w.r.t. threads, we also measured the performance overheads with increase in the size of the input data. For that, we report the performance overheads for four applications that are available with three input sizes: small (*S*), medium (*M*), and large (*L*). These four applications are: *histogram*, *linear_regression*, *string_match*, and *word_count*.

Application	Dataset / Parameters	Page faults	Faults/sec
blackscholes	16 in_64K.txt prices.txt	2.49E+04	2.58E+04
canneal	15 10000 2000 100000.nets 32	2.11E+06	21.57E+04
histogram	large.bmp	4.27E+04	10.78E+04
kmeans	-d 3 -c 500 -p 50000 -s 500	1.16E+06	13.99E+04
linear_regression	key_file_500MB.txt	2.88E+04	11.11E+04
matrix_multiply	2000 2000	2.32E+05	11.65E+04
pca	-r 4000 -c 4000 -s 100	5.34E+05	10.22E+04
reverse_index	datafiles	2.61E+04	10.35E+04
streamcluster	2 5 1 10 10 5 none output.txt 16	1.64E+05	1.163E+04
string_match	key_file_500MB.txt	3.11E+04	1.993E+04
swaptions	-ns 128 -sm 50000 -nt 16	4.66E+04	1.207E+04
word_count	word_100MB.txt	1.56E+05	54.34E+04

Figure 7: Runtime statistics for all benchmarks with 16 threads (Detailed log analysis is available here: web-link).

In this experiment, we kept the number of threads to a constant (16 threads), and we varied the input sizes for these applications. Figure 8 shows the results for our experiment. The bar plot shows the performance overheads w.r.t. to the native `pthreads` execution on the *Y*1-axis for three input sizes (*S*, *M*, *L*). For the reference, the input sizes are also shown by a line plot in the same figure on the *Y*2-axis.

The result shows that the gap between `pthreads` and INSPECTOR narrows with bigger input sizes. This is due to the fact that most applications use a data-parallel programming design pattern for parallelization, where the main threads divides the input data evenly between the worker threads. As the input size increases, each thread needs to perform more work (or compute on a larger input size) than the time spent for synchronization. As a result, each thread spends relatively more time outside the shared-memory commit to compute on the data, and thus, it results in improved performance.

D. Space Overheads for the Provenance Graph

Finally, we present the space overhead for storing the provenance graph. A major limitation of using Intel PT is that it produces a large amounts of trace data. Furthermore, the threading library also produces trace data to record the data and schedule dependencies. Table 9 shows the space overhead for all applications with 16 threads. Note that we report the combined space overheads for INSPECTOR, the individual breakdown between the threading library and the OS support for Intel PT is available online: web-link.

The space overheads vary across applications: it can be as low as 183MB for *linear_regression* and as high as 29.3GB for *streamcluster*. The result shows a strong correlation between the log bandwidth and branch instructions with a correlation coefficient of 0.89, which was expected, because the log consists of taken branches.

Fortunately, the provenance log written by `perf` turns out to be highly compressible. We were able to achieve a compression ratio of between 6× and 37× times using the *lz*4 compression algorithm. Furthermore, the snapshot facility (described in §VI) restricts the active area of space

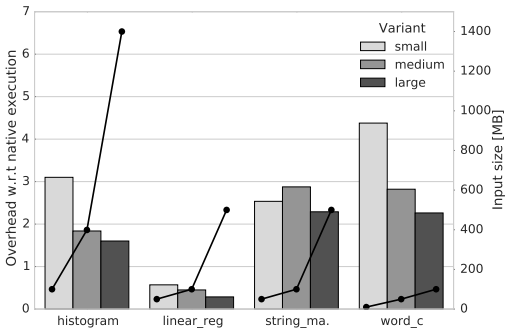

Figure 8: Scalability of overheads with increase in the input data sizes with 16 threads. The corresponding work measurement plot is available here: web-link.

usage, and the user can reuse the space in the ring buffer after analyzing (or collecting) the provenance graph.

VIII. Discussion: Case-studies

While data provenance is useful across a wide range of workflows, we discuss three active projects where INSPECTOR is being used to increase the dependability, security, and efficiency of software systems.

Dependability: Debugging programs [16]. Multithreaded programs are notoriously difficult to debug because of the inherent non-deterministic thread scheduling by the OS. Currently, debugging techniques rely on examining the memory state during the program execution or by analyzing core dumps after the crash. These techniques mainly target "what" is the state of the program without revealing much about "why" is it the state of the program is like that. Our library can be extended to aid the developers to better understand the failed execution by augmenting the existing debugging techniques with the provenance of the memory state.

Security: Dynamic Information Flow Tracking (DIFT) [34]. DIFT protects software against data leaks by restricting the suspicious I/O calls. Our library can be extended to support DIFT by carrying a taint for the sensitive data as part of the provenance, and restricting the output activities at the level of system calls. In particular, a policy checker can analyze the taint provenance to disallow sensitive data leaks. The policy checker can be embedded at the level of `glibc` wrappers for the output system calls. Note that we currently target accidental or buggy, but not a malicious threat model because our library is a user-space solution.

Efficiency: Memory management for NUMA [22]. The recent advancement in NUMA architectures offers a wide range of configurations for the interconnects with varying memory bandwidth, and it is unclear how these different configurations affect the OS support for memory management. Our library can be extended to investigate the potential impact of interconnect topologies on memory management, and can be extended to optimize the memory layout for a given interconnect topology. This optimization requires the memory access patterns that could be easily derived from the CPG.

Application	Provenance log details [MB] Size	Compressed	Ratio	Bandwidth [MB/sec]	Branch instr. [Instr/sec]
blackscholes	851	57.3	15×	882	2.49E+09
canneal	5343	315.0	17×	547	1.55E+09
histogram	381	11.3	34×	961	4.17E+09
kmeans	11900	522.0	23×	1438	5.79E+09
linear_regression	183	5.5	34×	707	3.81E+09
matrix_multiply	2101	97.0	22×	105	4.05E+08
pca	1900	116.0	16×	364	1.42E+09
reverse_index	192	5.7	34×	764	2.87E+09
streamcluster	29300	787.0	37×	2083	7.78E+09
string_match	2751	430.0	6×	1763	5.61E+09
swaptions	7061	929.0	8×	1830	4.84E+09
word_count	4121	508.0	8×	1435	2.80E+09

Figure 9: Space overheads for all benchmarks with 16 threads.

IX. Related Work

Data provenance is a well-studied concept because of it's wide applicability in different complex computer systems. Next, we review the related work from different domains.

Database systems. Provenance has been shown to be important in databases for materialized views, probabilistic databases, data integration, and curated databases (see a survey paper for more details [13]). Almost, all existing provenance work in databases leverage the explicit database schema and structured layout of the input records in tables to build the provenance graph; whereas, INSPECTOR does not assume any structured layout of the input data.

"Big Data" analytics. Data provenance is being increasingly used in "big data" processing for debugging complex workflows [29, 36–38], and also for incremental computation [5–7, 9–11, 21]. In particular, these systems construct the provenance graph based on the data-flow graph generated from the data-parallel programming model. Instead of relying on the constrained task-based programming model, INSPECTOR derives the graph automatically for shared-memory multithreaded programs.

Distributed and network systems. Many distributed and network systems propose provenance techniques for tracing the execution of distributed protocols to provide accountability, fault detection, forensics, verifiability, network debugging, negative provenance [39–41]. These systems leverage the semantics of distributed protocols to derive a state-machine, and capture the lineage information by manually modifying the state-machine. Instead, we do not require any protocol-specific state-machine. Albeit, we currently do not support distributed systems.

Storage systems. Storage systems, such as PASS [28], supporting provenance collect meta-data of newly created objects in the system (via the OS support), and maintain their lineage information such as the chain of ownership and the transformations performed on objects. In contrast to PASS that tracks objects in storage systems, our focus is on tracing the lineage of shared-memory accesses in multithreaded programs at the granularity of memory pages. Like PASS, we also rely on the OS support for tracking of memory pages.

Memory tracing. Our approach is complementary to numerous run-time [30] and compile-time [26] tools that

allow fine-grained byte-level memory read and writes made by threads. In contrast, our tool makes a trade-off of memory tracking at the granularity of memory pages, and uses a combination of OS support and the new ISA extensions to track the data flow for the entire program.

Operating systems. Linux Provenance Module (LPM) [3] provides OS support to collect system-wide provenance. In contrast to LPM, INSPECTOR is a user-space solution and does not require any modifications to the underlying OS. Secondly, unlike LPM, which collects provenance at the granularity of a process, we collect data provenance at a finer granularity of a thread. On the other hand, LPM benefits from the integrated OS approach to secure the provenance information.

Programming languages. Programming languages researchers develop language-based provenance approaches relying on a new language with special data-types. These language-based approaches derive the provenance graph using techniques such as self-adjusting computation [2]. In contrast, our work supports existing programs without relying on any language-level support or a new type system.

X. CONCLUSIONS

In this paper, we presented INSPECTOR, a data provenance library for multithreaded programs. Our approach targets existing executables, relies on OS-specific mechanisms and new ISA extensions of Intel PT to efficiently build the *Concurrent Provenance Graph (CPG)*. The CPG records control, data, and schedule dependencies for the shared-memory multithreaded program execution. Our solution is straightforward to deploy: it simply replaces the `pthreads` library, allowing existing applications to benefit from our approach with no recompilation or code changes. INSPECTOR's source code is publicly available for further use in a wide-range of workflows for data provenance: https://github.com/Mic92/inspector.

Acknowledgements. This work is supported in part by cfaed at TU Dresden.

REFERENCES

[1] `pthreads` Memory Model. http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html. Accessed: Dec, 2015.

[2] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.

[3] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer. Trustworthy whole-system provenance for the linux kernel. In *USENIX Security*, 2015.

[4] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. In *OOPSLA*, 2009.

[5] P. Bhatotia. *Incremental Parallel and Distributed Systems*. PhD thesis, Max Planck Institute for Software Systems (MPI-SWS), 2015.

[6] P. Bhatotia, U. A. Acar, F. Junqueira, and R. Rodrigues. Slider: Incremental Sliding Window Analytics. In *proceedings of the 15th Annual ACM/IFIP/USENIX Middleware conference (Middleware)*, 2014.

[7] P. Bhatotia, M. Dischinger, R. Rodrigues, and U. A. Acar. Slider: Incremental Sliding-Window Computations for Large-Scale Data Analysis. In *Technical Report: MPI-SWS-2012-004*, 2012.

[8] P. Bhatotia, P. Fonseca, U. A. Acar, B. B. Brandenburg, and R. Rodrigues. iThreads: A Threading Library for Parallel Incremental Computation. In *ASPLOS*, 2015.

[9] P. Bhatotia, R. Rodrigues, and A. Verma. Shredder: GPU-Accelerated Incremental Storage and Computation. In *proceedings of the 10th USENIX conference on File and Storage Technologies (FAST)*, 2012.

[10] P. Bhatotia, A. Wieder, I. E. Akkus, R. Rodrigues, and U. A. Acar. Large-scale incremental data processing with change propagation. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2011.

[11] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: MapReduce for Incremental Computations. In *SoCC*, 2011.

[12] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[13] P. Buneman and W.-C. Tan. Provenance in databases. In *SIGMOD*, 2007.

[14] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *SOSP*, 1991.

[15] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *TOCS*, 1985.

[16] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, 2009.

[17] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *ISCA*, 1990.

[18] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the speed of multi-core. In *EuroSys*, 2014.

[19] R. Gupta, E. Mehofer, and Y. Zhang. Profile guided compiler optimizations, 2002.

[20] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *USENIX*, 1994.

[21] D. R. Krishnan, D. L. Quoc, P. Bhatotia, C. Fetzer, and R. Rodrigues. IncApprox: A Data Analytics System for Incremental Approximate Computing. In *WWW*, 2016.

[22] R. Lachaize, B. Lepers, and V. Quema. MemProf: A Memory Profiler for NUMA Multicore Systems. In *USENIX ATC*, 2012.

[23] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, 1997.

[24] T. Liu and E. D. Berger. SHERIFF: Precise Detection and Automatic Mitigation of False Sharing. In *OOPSLA*, 2011.

[25] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient Deterministic Multithreading. In *SOSP*, 2011.

[26] B. Lucia and L. Ceze. Data provenance tracking for concurrent programs. In *CGO*, 2015.

[27] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*, 1989.

[28] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *USENIX ATC*, 2006.

[29] Olston et al. Nova: continuous pig/hadoop workflows. In *SIGMOD*, 2011.

[30] M. Payer, E. Kravina, and T. R. Gross. Lightweight memory tracing. In *USENIX ATC*, 2013.

[31] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. Functional programs that explain their work. In *ICFP*, 2012.

[32] E. Pozniansky and A. Schuster. Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. In *PPoPP*, 2003.

[33] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA*, 2007.

[34] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, 2004.

[35] J. Thalheim, P. Bhatotia, and C. Fetzer. Inspector: A Data Provenance Library for Multithreaded Programs. In *arXiv technical report*, 2016.

[36] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Brief Announcement: Modelling MapReduce for Optimal Execution in the Cloud. In *proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of Distributed Computing (PODC)*, 2010.

[37] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Conductor: Orchestrating the Clouds. In *proceedings of the 4th international workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2010.

[38] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the Deployment of Computations in the Cloud with Conductor. In *proceedings of the 9th USENIX symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[39] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. Diagnosing missing events in distributed systems negative provenance. In *SIGCOMM*, 2014.

[40] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure network provenance. In *SOSP*, 2011.

[41] W. Zhou, S. Mapara, Y. Ren, Y. Li, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr. Distributed time-aware provenance. In *VLDB*, 2013.