Optimistic Shared Memory Dependence Tracing

Yanyan Jiang*, Du Li†, Chang Xu*‡, Xiaoxing Ma*‡, Jian Lu*
*State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
†School of Computer Science, Carnegie Mellon University
jiangyy@outlook.com, duli@cs.cmu.edu, {changxu,xxm,lj}@nju.edu.cn

***Abstract*—Inter-thread shared memory dependences are crucial to understanding the behavior of concurrent systems, as such dependences are the cornerstone of time-travel debugging and further predictive analyses. To enable effective and efficient shared memory dependence tracing, we present an optimistic scheme addressing the challenge of capturing exact dependences between unsynchronized events to reduce the probe effect of program instrumentation. Our approach achieved a wait-free fast path for thread-local reads, and simultaneously achieved precise tracing of exact read-after-write, write-after-write and write-after-read dependences on the fly. We implemented an open-source RWTrace tool, and evaluation results show that our approach not only achieves efficient shared memory dependence tracing, but also scales well on a multi-core computer system.**

***Index Terms*—concurrency; shared memory dependence; dynamic analysis**

I. Introduction

Shared memory and locks, though being widely adopted in the ever-increasing concurrent programing paradigm, are notoriously difficult to reason. To understand the behavior of such systems, a key challenge is how to efficiently keep track of *shared memory dependences* in a concurrent program's execution. Tracing shared memory dependences enables record and replay of a previous, oftentimes erroneous, execution [1], [2], [3], as well as facilitates trace analyses for data races [4], [5] and atomicity violations [6], [7].

To trace the precise order of shared memory accesses, a straightforward way is to guard every shared memory access with a lock [2], [8]. These lock-based techniques, however, may drastically slow down programs. To make shared memory dependence tracing more efficient, recent research work found that thread-local accesses are predominant for real-world programs, and *optimistic* approaches are therefore proposed to reduce the overall probe effect [9], [10], [11]. For example, Octet [9] made thread-local memory accesses extremely efficient, at the cost of expensive roundtrip coordination for inter-thread dependences. Stride [10] ingeniously eliminated all read-time synchronization, but requires a post-execution inference whose soundness may suffer from hash collisions. CARE [11] logs even less shared memory dependences, but only provides value-deterministic replay guarantee. Overall, existing approaches attempted to strike a tradeoff between performance and determinism, but they also came with limitations, either requiring offline interference or introducing expensive synchronization operations on certain occasions.

‡ Corresponding authors.

We in this paper present a novel optimistic shared memory dependence tracing technique to overcome the aforementioned challenges in order to make it more practical. We are able to *efficiently* and *precisely* trace all three kinds of inter-thread shared memory dependences *on the fly*: read-after-write, write-after-write and write-after-read dependences.

Aligning with previous work, we propose an optimistic design such that thread-local reads are wait-free [12]. Our approach complements Octet [9] when shared memory communication is frequent, as our slow path is a lock-protected critical section and heavy round-trip coordination is avoided. To achieve a wait-free fast path, we trace read-after-write dependences by combining insights of wait-free notification in Stride [10] and speculation-retry design in CARE [11]. We also address the even more challenging task of online tracing exact write-after-thread-local-read dependences by using a carefully designed hash table of which fast-path lookups are wait-free. These algorithms are briefly introduced in Section II, and comprehensively discussed in Section III.

Our second contribution is extending our technique to systems of relaxed memory model. As mentioned earlier, our thread-local read fast path is wait-free. We further prove that, for the most widely spread commodity x86 multiprocessor systems [13], we achieved precise read-after-write dependence tracing without any memory fence instruction or atomic operation. The proof also draws a positive conclusion to the applicability of previous deterministic record and replay work [10], [11], as their correctness can be proved by exactly the same technique. Memory fences are carried out to retain fast path wait-freeness in the case that write-after-read dependences are traced, or the memory model is weaker than total store order. Section III-E presents these justifications.

Our final contribution is open-source implementation of the shared memory dependence tracing tool RWTrace. We evaluated RWTrace with a series of concurrent benchmarks including desktop, scientific and server programs on a multi-core machine. Evaluation results show that RWTrace outperforms any lock-based approaches, and is tens of times faster for several memory-intensive benchmarks. Evaluation results also show that our RWTrace tool scales well even if the non-uniform memory access architecture is present. The RWTrace implementation is presented in Section IV followed by evaluation in Section V.

Finally, we present qualitative discussion of related work and conclude our paper in Section VI and VII, respectively.

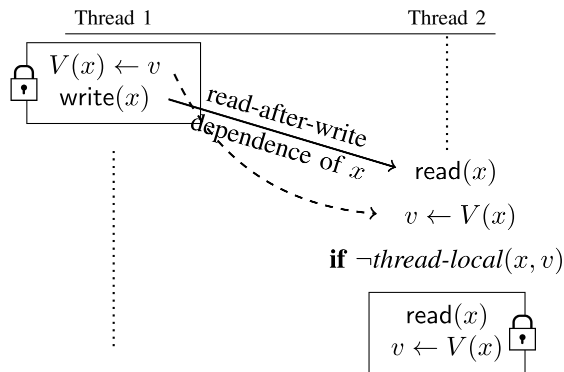

Figure 1. Capturing read-after-write dependences

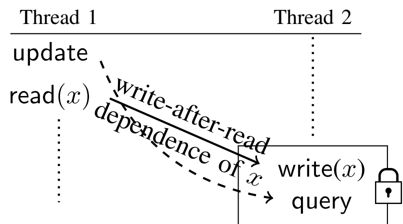

Figure 2. Capturing write-after-read dependences

II. Overview

A. Background

Behavior of concurrent systems consisting of threads and shared memory is difficult to reason, because the shared memory access order largely affects the program outcome. If two shared memory accesses are performed on the same address by two different threads and at least one of them is a write, we say that these two accesses are conflicting. Knowing the exact order of conflicting accesses is crucial to understand the system's behavior. With such ordering information, one can deterministically replay a problematic execution [1], [2], [3], or conduct further testing and analysis in search of faults and defects [4], [5], [6], [7].

B. Optimistically Tracing Shared Memory Dependences

To efficiently capture inter-thread shared memory dependences, recent research exploits the access pattern of concurrent programs, in which shared memory reads predominate writes [10], and a vast majority of accesses are not involved in an inter-thread dependence [9]. These insights facilitate shared memory dependence tracing approaches that behave differently among shared memory accesses, either in a frequent fast path or in a rare slow path. For example, Stride [10] does not synchronize for read events at all, but requires an offline inference procedure to restore shared memory dependences. Octet [9] achieved extremely efficient fast path for thread-exclusive and read-shared memory accesses, but an inter-thread dependence requires a much more costly roundtrip coordination.

We propose a novel shared memory dependence tracing recipe following an optimistic design that favors thread-local reads[1]. To achieve efficient dependence tracing, the major challenge is how to make the thread-local read fast path *wait-free* [12]. According to the types of two incorporated events, a shared memory dependence can be categorized as a read-after-write, write-after-write or write-after-read dependence. For each category, we highlight its challenge and our solution as follows. Detailed discussions are expanded in Section III.

Write-after-write dependences. We adopt the standard approach of serializing all write events to the same shared address [2], [9], [10], [11]. By maintaining the latest writer thread's identifier for each shared address, such dependences can be faithfully captured.

Read-after-write dependences. Since the thread-local reads are predominant, the major challenge is how to design an efficient fast path for them. Like [11], we *speculatively assume that a read is thread-local* and perform the read instantly followed by a *posterior test* of thread-locality, achieving a fast path when the test is passed. On the other hand, when the thread-locality test fails, we remedy the effect by a slow path, which performs the read again with lock.

However, the thread-locality test in [11] is too optimistic that may miss real dependences. We therefore adopt the memory ordering technique proposed in [10] to design a sound thread-locality test that always fails for non-thread-local reads. Figure 1 demonstrates the discovery of a read-after-write dependence between two threads accessing x (a solid line denotes a shared memory dependence, while a dashed line indicates a transitively deduced dependence). In short, we associate each address x with a version identifier $V(x)$, and perform the update of $V(x)$ *before* the write to x while queries $V(x)$ *after* performing the read from x. The designated event ordering guarantees that whenever a read event is reading an inter-thread value, the corresponding update of $V(x)$ is detected to trigger a thread-locality test failure, and hence an inter-thread dependence is captured.

Write-after-read dependences. Efficiently tracing write-after-read dependences is even more challenging because there is no synchronization between a thread-local read and a write in which a dependence is incorporated (i.e., a write-after-thread-local-read dependence). To the best of our knowledge, none of existing optimistic dependence tracing scheme can obtain the exact read events for a write-after-read dependence without a second inference pass[2].

To trace write-after-read dependences, we also enforce update is performed *before* read and query is performed *after* write, as shown in Figure 2. For each shared address, we maintain a hashed mapping to keep track of each thread's latest read event to the address. By a careful design, we make sure that any structural modification to the hash table will only happen in the slow path, and hash table update can always succeed for thread-local reads, achieving a wait-free fast path

[1]A read is thread-local if the thread had accessed the same address before, and there is no other thread writing to it in between.

[2]Octet [9], for example, enforces a RdSh $\rightarrow$ WrEx ordering by a global roundtrip coordination, but the dependence's precise reader events are lost.

that only contains an $O(1)$ hash table lookup and a shared memory write.

Finally, we offer users two alternatives in different application scenarios for practical concerns. If one collects shared memory dependences for a time-travel debugging or an offline trace analysis [2], [7], [14], read-after-write and write-after-write dependences are sufficient for restoring all write-after-read dependences by the deterministic replay algorithm described in [11]. Further analysis in Section III-E shows that in this case our read fast path is correct under the x86-TSO memory model (neither memory fence instruction nor atomic operation is needed). On the other hand, we can also trace every shared memory dependence on the fly at the cost of a few additional operations and a memory fence at each shared memory access. Such full set of dependences can be directly used to perform trace analyses [5], [15].

III. Tracing Shared Memory Dependences

We take a top-down approach in describing our shared memory dependence tracing scheme. After the notations and definitions listed in Section III-A followed by the formalized algorithm in Section III-B, we first expand the discussion on how read-after-write and write-after-write dependences are traced in Section III-C. Then, we present two approaches to tracing write-after-read dependences in Section III-D. Finally, we conduct the justification of extending our scheme to systems of relaxed memory model in Section III-E.

A. Preliminaries

1) Thread and event: We assume that a concurrent system consists of threads accessing a shared memory. Each thread t executes a single in-program-order stream of events

$$E_t = \{e_1, e_2, \dots, e_n\},$$

as defined in x86-TSO [13]. We use the unique identifier $\langle t : e_i \rangle$ to refer to event e_i performed by t. Each event e_i is of one of the following types:

1) $\mathsf{W}_t[x] = v$, for a write of value v to address x by thread t.
2) $\mathsf{R}_t[x] = v$, for a read of value v from address x by thread t.
3) $\mathsf{Acq}_t[x]$, for an acquisition of lock at address x by thread t. t is blocked until the lock is successfully acquired.
4) $\mathsf{Rel}_t[x]$, for a release of lock at address x by thread t.

These four types of events are sufficient for defining a sequentially consistent execution model. We exclude other synchronization events (e.g., thread fork/join and monitor events) for brevity since they can be easily constructed by proper Acq and Rel events. The x86-TSO memory model also defines other memory-model specific events like memory fence F_t, in-processor lock $\mathsf{L}_t/\mathsf{U}_t$ and store buffer flush τ_p. These notations do not pose any difficulty in understanding our algorithm in a sequentially consistent memory model setting, and hence we omit them for brevity. Precise semantics of these events can be found in [13].

2) Shared memory dependence: Let us first assume that our target system has a sequentially consistent memory model (i.e., shared memory accesses are not reordered and appear to be atomic). We will extend our discussions to relaxed memory model in Section III-E.

Let $E = \cup_t E_t$ be the set of all events. For a sequentially consistent system, there exists a total order $\overset{sc}{\rightarrow} \subseteq E \times E$ such that every read event in E returns the most recent written value in $\overset{sc}{\rightarrow}$ [16]. To capture the behavior of a concurrent system, one can naively obtain $\overset{sc}{\rightarrow}$ by serializing all shared memory accesses, precluding any possibility of parallelism. To best recover parallelism, we can also decompose the task of tracing $\overset{sc}{\rightarrow}$ into detecting inter-thread shared memory dependences, and recover $\overset{sc}{\rightarrow}$ later by a linear extension [8].

Specifically, to faithfully capture the behavior of a concurrent system, both program order dependences and inter-thread shared memory dependences are required. The *program order dependence* $\overset{po}{\rightarrow} \subseteq E \times E$ encodes the happens-before order between events in the same thread, i.e., $\langle t : e_i \rangle \overset{po}{\rightarrow} \langle t : e_{i+1} \rangle$. On the other hand, an *inter-thread shared memory dependence* indicates that a thread is reading or writing a value that is previously accessed by another thread. Specifically, for two shared memory access events $\langle t : e_i \rangle$ and $\langle t' : e_j \rangle$ performed on the same address, if either e_i or e_j is a write, the relative order $\langle t : e_i \rangle \rightarrow \langle t' : e_j \rangle$ is critical for understanding the program's behavior [9]. According to the type of e_i and e_j, such dependences can be categorized into three kinds: read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW), which are represented by $r \overset{raw}{\rightarrow} w$, $w \overset{war}{\rightarrow} r$ and $w \overset{waw}{\rightarrow} w'$, respectively.

Let $\overset{hb}{\rightarrow}$ be the transitive closure of all the aforementioned dependences, i.e.,

$$\overset{hb}{\rightarrow} = tr\big(\overset{po}{\rightarrow} \cup \overset{raw}{\rightarrow} \cup \overset{war}{\rightarrow} \cup \overset{waw}{\rightarrow} \big).$$

A classical result is that any linear extension of $\overset{hb}{\rightarrow}$ is a sequentially consistent total order (i.e., a valid $\overset{sc}{\rightarrow}$) [8], and hence $\overset{hb}{\rightarrow}$ is sufficient for deterministic replay. Since program order is trivially preserved by program execution, we are most interested in the inter-thread shared memory dependences which cannot be transitively deduced by program order in $\overset{raw}{\rightarrow}$, $\overset{war}{\rightarrow}$ and $\overset{waw}{\rightarrow}$, which are sufficient for reproducing and understanding a shared memory system's concurrent execution.

3) Thread Locality: If thread t's shared memory accesses $\langle t : e_{i_1} \rangle \overset{po}{\rightarrow} \langle t : e_{i_2} \rangle \overset{po}{\rightarrow} \dots \overset{po}{\rightarrow} \langle t : e_{i_k} \rangle$ are performed on the same address a, and there is no other thread writing a in between, we consider all these events *thread-local* excluding the first event. Formally, events $\{\langle t : e_{i_2} \rangle, \langle t : e_{i_3} \rangle, \dots \langle t : e_{i_k} \rangle\}$ are thread-local if there is no write event $\langle t' : e_j \rangle$ such that $t \neq t'$ and $\langle t : e_{i_1} \rangle \overset{hb}{\rightarrow} \langle t' : e_j \rangle \overset{hb}{\rightarrow} \langle t : e_{i_k} \rangle$. The first event $\langle t' : e_{i_1} \rangle$ is excluded because it may be incorporated in a inter-thread dependence. Thread-local events can be combined as a single event in understanding the behavior of a concurrent system, and are thus not have to be logged. We keep this insight in mind in designing our optimistic approach to shared memory dependence tracing.

Algorithm 1: Instrumentation for $\langle t : e_i \rangle$: $\mathsf{R}_t[a] = v$

```
1  update(M_a, ⟨t : e_i⟩);
2  [R_t[a] = v];
3  R_t[V(a)] = ⟨t' : e_j⟩;
4  result ← thread_locality_test(a, ⟨t' : e_j⟩) // discussed in
   Section III-C3;
5  if result cannot prove e_i is thread-local then
6      Acq_t[L(a)];
7      R_t[a] = v';
8      R_t[V(a)] = ⟨t'' : e_j'⟩;
9      update(M_a, ⟨t, e_i⟩);
10     Rel_t[L(a)];
11     Override read value of R_t[a] = v by v';
12     log_dependence(⟨t'' : e_j'⟩ -raw-> ⟨t : e_i⟩);
13     thread_locality_update(a, ⟨t'' : e_j'⟩);
```

Algorithm 2: Instrumentation for $\langle t : e_i \rangle$: $\mathsf{W}_t[a] = v$

```
1  Acq[L(a)];
2  R_t[V(a)] = ⟨t' : e_j⟩;
3  W_t[V(a)] = ⟨t : e_i⟩;
4  [W_t[a] = v];
5  for ⟨t_k : e_k⟩ ∈ query(M_a) do
6      log_dependence(⟨t_k : e_k⟩ -war-> ⟨t : e_i⟩);
7  Rel[L(a)];
8  if t' ≠ t then
9      log_dependence(⟨t' : e_j⟩ -waw-> ⟨t : e_i⟩);
10 thread_locality_update(a, ⟨t : e_i⟩);
```

B. Program Instrumentation

To achieve efficient shared memory dependence tracing, we associate each shared address a with a lock $L(a)$. Different shared memory addresses are allowed to share a lock[3]. For each lock $L(a)$, we also maintain its version $V(a)$ which denotes the identifier of the latest write event performed when $L(a)$ is acquired. $V(a)$ is in form of $\langle t : e_i \rangle$ indicating that the most recent writer to a is the i-th event $\mathsf{W}_t[a] = v$ performed by thread t.

In case that WAR dependences are traced, we further associate each shared address a with a key-value pair data structure M_a. We use M_a to keep each thread t's latest event reading address a, and therefore we require M_a to support the following two operations:

1) $\mathsf{update}(M_a, \langle t : e_i \rangle)$, for updating the latest read event of thread t to be e_i. In other words, update sets the key t's corresponding value to e_i.
2) $\mathsf{query}(M_a)$, for returning all $\langle t : e_i \rangle$ pairs, and remove all such pairs in M_a.

[3]This is known as the technique of *variable grouping*, and different variable grouping techniques favor different workloads. For example, LEAP [2] and Stride [10] group addresses sharing a common field name, while Order [17] and Octet [9] group addresses in the same object, and there also exist fine-grained implementations [18]. This issue is out of our scope and hence not discussed further.

In Section III-D we show that M_a can be efficiently implemented, preserving wait-freeness for our read fast path.

As briefly discussed in Section II, the key insight of capturing a shared memory dependence between $\langle t : e_i \rangle$ and $\langle t' : e_j \rangle$ (either a RAW or WAR dependence) is to instrument the program creating the event ordering of $A \overset{po}{\to} \langle t : e_i \rangle \overset{hb}{\to} \langle t' : e_j \rangle \overset{po}{\to} B$, where A updates a shared state indicating the existence of $\langle t : e_i \rangle$ and this state change is visible at B such that we can detect a dependence. This insight is realized as our shared memory dependence tracing algorithms. For a read event $\mathsf{R}_t[a] = v$ with identifier $\langle t : e_i \rangle$, our instrumentation is illustrated in Algorithm 1. Similarly, write event $\mathsf{W}_t[a] = v$ with identifier $\langle t : e_i \rangle$'s instrumentation is shown in Algorithm 2. For visual conventions, a box indicates that its contained event is the one being instrumented, and underlined code snippets are only used for tracing WAR dependences. All non-boxed shared memory access events (R_t and W_t) are performed by our instrumentation. When we are discussing the rationale of tracing RAW and WAW dependences in Section III-C, these underlined codes can be omitted.

C. Tracing RAW and WAW Dependences

1) Tracing RAW Dependences: Instrumentation for a read event (Algorithm 1) consists of two phases. We design the first phase to be a wait-free speculative read and a thread-locality test. It performs the read event $\boxed{\mathsf{R}_t[a]}$ as usual (Line 2), followed by a read of a's latest write event identifier $\mathsf{R}_t[V(a)]$ (Line 3) and finally a thread-locality test (Line 4).

We expand our discussion of thread-locality test and its correctness in Section III-C3. We design the test to exhibit only single-sided error. In other words, the thread-locality test is guaranteed to fail for any inter-thread read event, assuring that all inter-thread RAW dependences are captured.

Only when the thread locality test cannot confirm a thread-local read, a second phase is carried out to capture the exact inter-thread RAW dependence (Lines 5–13). By acquiring a's corresponding lock $L(a)$, we serialize the read event with any write event to a because all writes to a are also protected by $L(a)$ in Algorithm 2. In this lock-protected region, we once again perform the read event $\mathsf{R}_t[a]$ obtaining v' (Line 7) whose writer event's identifier is $\langle t'' : e_{j'} \rangle$ (Line 8). We override the read value v in $\boxed{\mathsf{R}_t[a] = v}$ by v', essentially "replacing" the read in Line 2 by the one in Line 7, obtaining the exact RAW dependence $\langle t'' : e_{j'} \rangle \overset{raw}{\to} \langle t : e_i \rangle$.

Our two-phase read instrumentation design gains from the fact that thread-local reads are far more frequent than the others in real-world programs [9], [11], [10]. The thread-locality test consists of only thread-local operations, and therefore our read fast path (i.e., passing thread-locality test in the first phase) is wait-free and consumes only a few machine instructions.

2) Tracing WAW Dependences: The write instrumentation (Algorithm 2) protects every write event to address a with lock $L(a)$, to ensure that all writes to the same address are

serialized. We also update $V(a)$ in the same lock-protected region to establish atomic update of both the value and its version. By studying the value stored in $V(a)$, Lines 8–9 filters out thread-local writes which can be inferred by the program order and only inter-thread WAW dependences are logged.

3) Detecting Thread Locality: The challenge of effectively and efficiently tracing RAW dependences is the design of a sound and fast thread locality test (Line 4 in Algorithm 1). The test must be sound so that no real dependence is missed, and should be extremely efficient because it is called very frequently.

To implement such test, we associate each thread t with $V_t(a)$, a thread-local shadow copy of $V(a)$. $V_t(a)$ stores t's knowledge about which write event is the latest one performed on a. We allow the thread-locality test to produce false negatives (failing the test for an actually thread-local read). Thus we allow $V_t(a)$ to hold inconsistent (i.e., outdated) writer information. $V_t(a)$ is updated only when an inter-thread RAW dependence is detected through thread_locality_update (Line 13 in Algorithm 1) or thread t itself is performing a write (Line 10 in Algorithm 2). The thread-locality test passes only if the actual writer event's identifier $\langle t' : e_j \rangle$ (fetched by Line 3 in Algorithm 1) matches the one stored in the shadow copy, i.e., $V_t(a) = \langle t' : e_j \rangle$. Note that it is neither practical nor necessary to maintain $V_t(a)$ for every a. Memory access locality indicates that the shadow mapping can be efficiently implemented by a cache-like structure of fixed size and any feasible cache replacement strategy [11].

We now explain why this thread-locality test can guarantee to recognize any inter-thread RAW dependence. Note that our write instrumentation (Algorithm 2) updates $V(a)$ *before* the actual write ($\boxed{\mathsf{W}_t[a] = v}$) is performed (Lines 3–4). Once $V(a)$ is updated, any other thread's shadow copy $V_t(a)$ (if exists) are immediately outdated, as if we had performed a global invalidate operation before v was written to the shared memory. Combined with our read instrumentation (Algorithm 1) that reads $V(a)$ *after* the actual read ($\boxed{\mathsf{R}_t[a]}$) is performed (Lines 2–3), we ensure that, if $\boxed{\mathsf{R}_t[a]}$ is reading a value written by another thread, $V_t(a)$ must be invalidated before $\mathsf{R}_t[V(a)]$ is performed, triggering a thread-locality test failure. Specifically, our instrumentation ensures that $\boxed{\mathsf{R}_t[a]} \overset{po}{\to} \mathsf{R}_t[V(a)]$ and $\mathsf{W}_{t'}[V(a)] \overset{po}{\to} \boxed{\mathsf{W}_{t'}[a]}$. If $\boxed{\mathsf{W}_{t'}[a]} \overset{raw}{\to} \boxed{\mathsf{R}_t[a]}$ is an inter-thread RAW dependence, the transitive law implies that $\mathsf{W}_{t'}[V(a)] \overset{hb}{\to} \mathsf{R}_t[V(a)]$, triggering a thread-locality test failure.

We remind our readers that this argument works only for a sequentially consistent memory model. This careful memory ordering design, however, may suffer from reordering issues in systems of relaxed memory model. We will return to this subject in Section III-E.

From another perspective, the thread-locality test can also be interpreted as a lightweight imprecise instance of the transitive reduction [19] technique for compacting the shared memory dependence log. Transitive reduction states that, for any partial order, say $\to$, we can safely remove a dependence $a \to b$ if there is some t such that $a \to t \wedge t \to b$. Repetitively removing redundant dependences yields a unique minimum representation of the partially ordered set. However, state-of-the-art transitive reduction is often performed offline [15]. Exact on-the-fly transitive reduction requires either heavy instrumentation [20], [19] or customized hardware [21]. Thread-locality test, on the other hand, only removes dependences for $a \to b$ if there is $a \overset{raw}{\to} t \wedge t \overset{po}{\to} b$ or $a \overset{waw}{\to} t \wedge t \overset{po}{\to} b$, achieving a balance of cost and effectiveness.

Algorithm 3: Algorithms for update and query

```
1  // M_a is implemented by a hash mapping from thread identifier
     t to a shared memory address addr
2  function update(M_a, ⟨t : e_i⟩)
3  begin
4      if L(a) is held then
5          // We are at Line 9 in Algorithm 1
6          addr ← new();
7          W_t[addr] = e_i;
8          hash_insert(M_a, ⟨t, addr⟩);
9      else
10         // We are at Line 2 in Algorithm 1
11         addr ← hash_lookup(M_a, t);
12         if lookup failed then
13             // do nothing, thread-locality test must fail
14         else
15             // update the write-after-read dependence
16             W_t[addr] = e_i;
17             MFENCE;
18 function query(M_a)
19 begin
20     // We are at Line 5 in Algorithm 2
21     MFENCE;
22     for all ⟨t, addr⟩ ∈ M_a do
23         R_t[addr] = e_i;
24         ret ← ret ∪ ⟨t : e_i⟩;
25     hash_reclaim(M_a); // remove all contents in M_a
26     hash_insert(M_a, ⟨t, new()⟩);
27     return ret;
```

D. Tracing WAR Dependences

We provide two independent approaches to tracing WAR dependences. The online tracing algorithm captures WAR dependences on the fly, at the cost of incurring a little more runtime overhead (retaining the wait-freeness for the thread-local read fast path). On the other hand, the offline algorithm uses only RAW and WAW dependences to infer WAR dependences by an extra replay pass.

1) Online Tracing WAR Dependences: The underlined code in Algorithm 1 and Algorithm 2 are used for tracing WAR dependences. Recall that in Section III-B, we use M_a to keep each thread's latest read event from address a. Let us first assume the atomicity of update and query. We update M_a before the read is performed, and query M_a after the write is

done. Using the similar argument presented in Section III-C3, we enforce $\mathsf{update}(M_a, \langle t : e_i \rangle) \stackrel{po}{\rightarrow} \boxed{\mathsf{R}_t[a]}$ for a read event (Lines 1–2 in Algorithm 1) and $\boxed{\mathsf{W}_{t'}[a]} \stackrel{po}{\rightarrow} \mathsf{query}(M_a)$ for a write event (Lines 4–6 in Algorithm 2). Therefore, for any $r = \mathsf{R}_t[a]$ happening before $w = \mathsf{W}_{t'}[a]$ indicating a WAR dependence, the transitive rule implies that r's update must happen before w's query, establishing the fact that no WAR dependence is missed.

There exists a subtle case when racing read and write events are present. Consider that a WAR dependence $r \stackrel{war}{\rightarrow} w$ is detected before r executes Line 2 of Algorithm 1. Later, r's thread-locality test may fail, yielding a logged RAW dependence $w' \stackrel{raw}{\rightarrow} r$ (w' is not necessarily equal to w). The cycle formed by $r \stackrel{war}{\rightarrow} w \stackrel{hb}{\rightarrow} w' \stackrel{raw}{\rightarrow} r$ clearly violates our presumption of partial order. Fortunately, this is the only case of generating a false dependence[4] that a cycle is formed. Removal of the false WAR dependence (restoring previously known WAR dependence $r' \stackrel{war}{\rightarrow} w$ where $r' \stackrel{po}{\rightarrow} r$ if exists) essentially restores the correct dependence.

The next challenge is how to efficiently implement M_a and its two operations, as a practical wait-free concurrent hash table implementation is not yet available [22]. To keep the fast path wait-free, we exploit the observation that thread t only claims itself reading a shared address. In other words, any update of M_a performed by thread t operates on key t. Accordingly, we implement M_a as an ordinary hash table, and design our algorithm to obey the invariant that key t must exist in M_a whenever thread t executes $\mathsf{update}(M_a, \langle t, e_i \rangle)$ at a thread-local read event. This invariant implies that no structural modification is ever needed for a thread-local read, as illustrated in Algorithm 3.

We enforce this invariant by a slight modification to our slow path instrumentation. For any read event r in thread t to be thread-local, there must be some event $u \stackrel{po}{\rightarrow} r$ executing thread_locality_update in Algorithm 1 or Algorithm 2, and u itself is either a write event or a non-thread-local read event capturing a RAW dependence $w \stackrel{raw}{\rightarrow} u$. In both cases, u accesses a and is protected by $L(a)$. Therefore, at this time, we insert the key t into M_a to ensure successful lookups for subsequent thread-local reads (Line 9 in Algorithm 1 and Lines 4–8, 26 in Algorithm 3).

Such modification only changes the internal implementation of M_a, and therefore our previous correctness arguments still hold. Nevertheless, several interesting cases arise. The first case is that update may fail to find the key t in the hash table (Lines 12–13 in Algorithm 3). Since hash table is only cleared at query in our write-time instrumentation, there must be another thread executed writing a value. In this situation, we have $\boxed{\mathsf{W}_t[a]} \stackrel{po}{\rightarrow} \mathsf{query} \stackrel{hb}{\rightarrow} \mathsf{update} \stackrel{po}{\rightarrow} \boxed{\mathsf{R}_{t'}[a]}$ and the read must not pass the thread-locality test. In other words, a hash table lookup failure always indicates that the subsequent thread-locality test will fail. For a similar reason, if $\mathsf{query}(M_a)$ and $\mathsf{update}(M_a, \langle t, e_i \rangle)$ happened in parallel, the read must also not be thread-local. Therefore a RAW dependence is logged, and any incorrect WAR dependence will be discarded.

[4] A WAR dependence between a write and a thread-local read is always a true dependence because a thread-local read always happens-before any writer afterwards.

Algorithm 4: An example hash table implementation

```
1  // H stores the references to the payload of the hash table, an
   array of ⟨key, value⟩ pairs
2  function hash_lookup(H, k)
3  begin
4      R_t[H] = h;
5      i ← hash_probe(h, k);
6      if key not found then
7          return FAIL;
8      R_t[h(i).value] = v;
9      return v; // read only
10 function hash_reclaim(H)
11 begin
12     W_t[H] = new();
13 function hash_insert(H, ⟨k, v⟩)
14 begin
15     R_t[H] = h;
16     i ← hash_probe(h, k); // read only
17     W_t[h(i).key] = k;
18     W_t[h(i).value] = v;
19     if hash table resize needed then
20         h' ← new();
21         copy elements in h to h';
22         W_t[H] = h';
```

We demonstrate a possible hash table implementation in Algorithm 4. Since conflicting hash_insert and hash_reclaim are protected by the same lock (Lines 8, 25–26 in Algorithm 3), we thus only need to consider the correctness of racing reads in hash_lookup. Since thread t only operates on key t, there will be no racing accesses to the same key. Moreover, as long as key t is in the hash table, lookup will always succeed no matter the old or the resized new hash table is fetched (Lines 22 in Algorithm 4). These justifications are sufficient for ensuring a wait-free fast path.

The final issue of tracing WAR dependences is memory ordering. In a concurrent system of relaxed memory model, memory accesses in the hash table operation may be reordered with the subsequent ones, violating our presumption of $\mathsf{update}(M_a, \langle t : e_i \rangle) \stackrel{po}{\rightarrow} \boxed{\mathsf{R}_t[a]}$ and $\boxed{\mathsf{W}_{t'}[a]} \stackrel{po}{\rightarrow} \mathsf{query}(M_a)$. Fences (Lines 17 and 21 in Algorithm 3) are therefore inserted to assure that (1) reads after update (Lines 2–3 in Algorithm 1) and (2) writes before query (Lines 3–4 in Algorithm 1) are not reordered with the hash table operations. More memory model issues are discussed in Section III-E.

In summary, we can precisely trace every RAW, WAW or WAR dependence. Moreover, by a careful design, we can achieve wait-freeness for our thread-local read fast path, and the amortized cost per dependence is $O(1)$.

2) Offline Deducing WAR Dependences: In case that the runtime overhead is sensitive (e.g., in a production run), or shared memory dependences are used for post-execution analysis (e.g., deterministic replay [10], [11], PTA [5], [7] or

directed random testing [14]), we offer an alternative approach to restoring WAR dependences by a one-pass replay. In this setting, underlined code in Algorithm 1 and Algorithm 2 is removed to achieve best runtime efficiency, and only RAW and WAR dependences are traced.

We now derive how to restore a WAR dependence using $\stackrel{po}{\to}$, $\stackrel{raw}{\to}$ and $\stackrel{waw}{\to}$. Consider a read event $r = \langle t : e_i \rangle$. Depending on the result of thread-locality test (Line 5 in Algorithm 1), r falls into either of the following two cases:

1) There is a logged RAW dependence $w \stackrel{raw}{\to} r$. We denote w to be the "writer" of r.
2) Otherwise, r must be thread-local, indicating that $V_t(a)$ is identical to the $\mathsf{R}_t[V(a)] = \langle t' : e_j \rangle$ in Line 2 of Algorithm 1. We take the most recent event $u \stackrel{po}{\to} r$ that updated $V_t(a)$. If $u = w$ is a write, $w \stackrel{po}{\to} r$ is r's writer. Otherwise, u must be a Case 1 read, we take $w \stackrel{raw}{\to} u \stackrel{po}{\to} r$ as the writer.

In both cases, no write event w' can exist such that $w \stackrel{hb}{\to} w' \stackrel{hb}{\to} r$ because all writes are serialized in our instrumentation. Therefore, for any $w_{pred} \stackrel{waw}{\to} w \stackrel{waw}{\to} w_{succ}$, we know that $w_{pred} \stackrel{hb}{\to} w \stackrel{hb}{\to} r \stackrel{hb}{\to} w_{succ}$. Therefore, we derive that all writes happening before w must also happen before r, and all writes happening after w must also happen after r: we essentially have restored WAR dependences for r. Based on this idea, we can adopt the replay algorithm described in [11] to replay the program execution with only $\stackrel{raw}{\to}$ and $\stackrel{waw}{\to}$, and restore any $\stackrel{war}{\to}$ if necessary.

E. Dealing with Relaxed Memory Model

1) Tracing WAW Dependences: Obviously, our algorithm can correctly trace WAW dependences because all writes to a are serialized by $L(a)$, and correct memory ordering is assured by lock semantics which is independent of memory model.

2) Tracing RAW Dependences: Let us focus on tracing RAW dependences, assuming that the underlined code used for tracing WAR dependences in Algorithm 1 and Algorithm 2 is removed. As discussed in Section III-C3, we carefully design the memory ordering to be $\mathsf{W}_t[V(a)] \stackrel{po}{\to} \mathsf{W}_t[a]$ for a write and $\mathsf{R}_t[a] \stackrel{po}{\to} \mathsf{R}_t[V(a)]$ for a read. Therefore, for any real WAR dependence $\mathsf{W}_t[a] \stackrel{hb}{\to} \mathsf{R}_t[V(a)]$, the transitivity rule implies that $\mathsf{W}_t[V(a)] \stackrel{hb}{\to} \mathsf{R}_t[V(a)]$ and the thread-locality test must fail.

However, in a system of relaxed memory model, the program order is no longer satisfied as memory accesses can be reordered by the underlying hardware. To cope with such non-sequential behaviors, memory access order can be ensured by inserting memory barriers after Line 2 of Algorithm 1 and after Line 3 of Algorithm 2.

One interesting result is that our unmodified RAW dependence tracing algorithm is correct for the most widely spread x86 multiprocessor systems. To derive the correctness, we first show that our instrumentation can capture all RAW dependences for x86-TSO memory model [13] by the following theorem.

Theorem 1. *In* x86-TSO*, if* $\mathsf{R}_t[a]$ *(Line 2 in Algorithm 1) is reading a value that is written by another thread, the thread-locality test at Line 4 must fail.*

Proof: (sketch) First, a read event $\mathsf{R}_t[a]$ reading value from the store buffer is not the case because such read is considered thread-local and we allow it to pass the test. Therefore, $\mathsf{R}_t[a] = v$ must be reading a value previously written by a write event $\mathsf{W}_{t'}[a] = v$ happened in $t' \neq t$. According to our instrumentation (Lines 3–4 of Algorithm 2), we know that $V(a)$ is updated before the actual write is performed, i.e., $\mathsf{W}_{t'}[V(a)] \stackrel{po}{\to} \mathsf{W}_{t'}[a]$, because each thread executes events in respect to the program order in x86-TSO, and both of the writes are inserted into the FIFO store buffer. Moreover, $\mathsf{R}_t[a]$ reading this value from shared memory indicates that $\mathsf{W}_{t'}[a]$ had been dequeued from the store buffer. The FIFO nature of the store buffer ensures that $\mathsf{W}_{t'}[V(a)] \stackrel{po}{\to} \mathsf{W}_{t'}[a]$ must also be dequeued, triggering a failure of our thread-locality test. ■

Note that this argument is correct even if we use a highly optimized spin-lock implementation that does not flush the store buffer at lock releases (e.g., the implementation described in [13], which is deployed in the pthread library and Linux Kernel). However, in this case, the entire system's behavior may not be sequentially consistent (SB example in [13] is a possible case). According to our previous work CARE [11], we can always reproduce each thread's local behavior with our traced RAW and WAW dependences by keeping a per-thread shadow memory at replay, and fetch values from the shadow memory instead of the shared memory for thread-local reads. This replay scheme works even if non-sequentially-consistent behavior exists because any RAW dependence is captured and a thread-local read's value is always determined by the shadow memory[5]. This is an interesting discovery that with the full set of shared memory dependences, we are able to reproduce non-sequentially-consistent behavior of a concurrent system.

To enforce a sequentially consistent memory model by instrumentation, we further require a store buffer flush each time at write-time lock release (Line 7 of Algorithm 2). With such treatment, the store buffer is always empty when read events at Lines 2–3 of Algorithm 1 are executed, and the values are read from the shared memory. Since all reads from the shared memory appear to be atomic in x86-TSO, a sequentially consistent memory model is guaranteed. This justification not only extends our CARE approach, but also applies for Stride [10] because it also serializes all shared memory writes, and hence a sequentially consistent memory model is enforced.

3) Tracing WAR Dependences: To trace WAR dependences, let us consider underlined code in Algorithm 1 and Algorithm 2. We use the similar memory ordering technique in tracing RAW dependences that places update $\stackrel{po}{\to} \mathsf{R}_t[a]$ and $\mathsf{W}_{t'}[a] \stackrel{po}{\to}$ query, to make sure that data in update reaches shared memory before query when a read happened before a write.

[5]In the SB example, both two shared memory reads are thread-local, and read values are identical to the ones in their shadow memory.

This case, however, is more complicated than that in tracing RAW dependences. When RAW and WAW dependences are traced, read instrumentation only contains reads and write instrumentation only contains writes. Since x86-TSO reorders neither a load with other loads nor a store with other stores, no memory barrier is ever needed. On the other hand, when we are tracing WAR dependence, `update` contains shared memory writes which may be reordered with the actual read $\mathsf{R}_t[a]$, causing a dependence to be missed. This is exactly why we insert MFENCE at Lines 17 and 21 of Algorithm 3.

4) Beyond Total Store Order: As described in Section III-B, the key insight of our dependence tracing scheme is the memory ordering of $A \stackrel{po}{\to} \langle t, e_i \rangle \stackrel{hb}{\to} \langle t', e_j \rangle \stackrel{po}{\to} B$, and use $A \stackrel{hb}{\to} B$ as indicator for an inter-thread dependence. Therefore, we must make sure that the $\stackrel{po}{\to}$ is not violated if we extend our algorithms to an even more relaxed memory model, and such memory ordering can be easily achieved by properly inserting memory fence instructions, while the wait-freeness is still retained.

IV. Implementation

We implemented our shared memory dependence tracing tool RWTrace for C/C++ programs compiled by LLVM [23]. Source code of RWTrace is publicly available at `http://github.com/jiangyy/rwtrace`. The RWTrace tool is implemented as an LLVM optimization pass. The optimization pass is capable of inserting arbitrary instruction sequence before and after any LLVM IR instruction, therefore our RWTrace tool is not limited to shared memory dependence tracing. We expect more dynamic analysis tools to be implemented on the RWTrace framework.

To trace shared memory dependences, we instrument `load` and `store` instructions according to the description of Algorithm 1–4. The implementation of required data structures are described as follows.

1) Every shared memory address is mapped to a lock $L(a)$. We statically allocate a total amount of 2^{20} lock objects, and map address `a` to the lock numbered `(a>>6)&0xfffff`. In other words, we divide the memory into 64-byte chunks, and let the address in the same chunk to share one same lock. We expect such address grouping strategy does not significantly impact the performance because this size is equal to the size of L1 cache line in commodity x86 systems, and false sharing of a cache line is expected to be rare.
2) The latest writer's event identifier $V(a)$ is stored within $L(a)$'s lock object because modifications to $V(a)$ are always protected by $L(a)$.
3) $V(a)$'s thread-local shadow copy $V_t(a)$ is kept in thread t's local storage, and is implemented by a direct mapping cache containing roughly 10K elements.
4) The latest read event mapping $M_a(t)$ is designed to support concurrent lookup and update. We implement it by a closed-address hash table and chaining to resolve conflict. As the linked list for each hash table bucket is monotonically increasing, we guarantee the lookups for thread-local reads are always successful. This design enabled us to use ring buffer for memory management, keeping the amortized cost of memory allocation and release to be $O(1)$ if the buffer is sufficiently large.

Other implementation details can be found in our open source project.

V. Evaluation

A. Methodology

We demonstrate that our RWTrace tool can efficiently trace shared memory dependences by evaluating RWTrace's performance on real-world benchmarks. We chose a set of subject programs that are extensively studied by the previous work (a superset of those studied in [3]). Specifically, we selected desktop applications, scientific benchmarks from PARSEC [24] and SPLASH-2 [25] as well as two HTTP Web server programs. Detailed description of these subject programs and their evaluation settings are listed in Table I. For benchmarking the server programs, HTTP requests are generated by the apache's standard benchmark tool `ab` [26].

We instrumented *all* shared memory accesses to the statically allocated data and heap in evaluation, capturing the full set of shared memory dependences (assuming that stack objects are not shared and library code is excluded). We evaluated the running time of our subject programs, as well as the amount of shared memory dependences being captured. All evaluation data were collected via averaging over 10 subject program runs.

We also implemented two lock-based shared memory dependence tracing techniques using the same infrastructure and options. The MutexLock is a C++ implementation of LEAP [2] which protects all shared memory accesses with pthread library's built-in mutex lock, and traces the dependences between lock acquisitions and releases. The second technique CREWLock enables multiple readers to execute in parallel by protecting shared memory accesses with read/write lock [27]. In this setting, only RAW and WAW dependences can be captured. It is difficult to implement Octet [9] because it requires coordination protocols to be conducted at program safe points, posing non-trivial challenge for an unmanaged runtime. Therefore, we measured Octet protocol's state transition statistics for a qualitative analysis based on our lock-based implementation.

The evaluation was conducted on a server with four 6-core Intel Xeon X7460 processors (24 cores in total) and 64GB RAM which has an NUMA architecture [28].

B. Evaluation Results

Table II shows our performance evaluation results. We collect the profiling information of the percentage of thread-local reads among all shared memory reads and all shared memory accesses, which are shown in Columns 2 and 3, respectively. Each unmodified subject program's running time is shown in Column 4 (running time is measured in seconds). Columns 5–6 show the running time and dependences captured

Table I
LIST OF BENCHMARKS AND EVALUATION SETTINGS

Subject		*Description*	*LOC*	*Evaluation Setting*
desktop	aget	parallel data fetch	2.5K	16 threads, 64MB file
	pfscan	parallel file scan	1.1K	16 threads, 640MB file
	pbzip2	parallel compression	1.9K	16 threads, 64MB file
scientific	ocean	ocean simulation	9.1K	16 threads, 1026 × 1026 grid
	water	water simulation	3.6K	16 threads, 1000 molecules, 10 steps
	fft	fast fourier transformation	1.4K	16 threads, 2^{24} data points
	radix	radix sort	1.9K	16 threads, 2^{24} elements
	fluid	fluid simulation	1.2K	16 threads, 10 frames, 100KB input
	qsort	quick sort	0.9K	16 threads, 2^{24} elements
	x264	video encoding	37K	16 threads, 128 frames, 640 × 360 resolution
server	knot	HTTP server	1.2K	16 threads, 2^{14} requests
	apache	HTTP server	339K	16 threads, 2^{14} requests

Table II
EVALUATION RESULTS

Subject	*TL/RD*	*TL/ALL*	*Time*	RWTrace (1)		RWTrace (2)		MutexLock		CREWLock		Octet		
				Time	*#Dep.*	*Time*	*#Dep.*	*Time*	*#Dep.*	*Time*	*#Dep.*	*%A/S*	*%U/F*	*%Cord.*
aget	53.06%	27.39%	4.53	4.59	3.12K	4.60	3.19K	4.89	2.84K	4.53	3.13K	54.61	22.70	22.68
pfscan	>99.99%	>99.99%	0.99	7.76	0.15K	18.45	0.21K	30.51	0.15K	28.61	0.12K	>99.99	<0.01	
pbzip2	86.86%	71.55%	1.99	2.01	0.66K	2.04	1.03K	2.07	1.32K	2.04	0.67K	86.10	2.91	10.99
ocean	99.99%	99.99%	1.61	1.87	24.7K	2.80	45.7K	54.56	49.6M	14.63	25.2K	99.98	0.01	0.01
water	99.89%	95.67%	0.50	1.10	822K	2.67	1.09M	21.86	74.0M	6.97	872K	99.16	0.19	0.64
fft	>99.99%	99.95%	3.36	4.28	2.34K	5.19	2.56K	5.52	3.09K	5.48	2.34K	>99.99	<0.01	
radix	99.96%	63.62%	0.79	0.91	2.34K	1.17	2.35K	8.46	9.17M	3.30	2.33K	>99.99	<0.01	
fluid	>99.99%	99.97%	0.53	1.46	11.5K	2.49	20.2K	111.17	119M	28.14	11.7K	>99.99	<0.01	
qsort	99.35%	61.49%	1.34	3.30	761K	5.56	1.56M	3.54	1.66M	3.35	783K	99.17	0.31	0.52
x264	99.96%	89.49%	4.26	29.41	3.75M	58.29	5.17M	69.25	84.4M	58.36	5.78M	99.98	0.01	0.01
knot	83.68%	47.09%	1.80	1.81	38.2K	1.83	79.6K	1.83	68.4K	1.83	38.6K	53.84	22.80	23.36
apache	99.93%	67.52%	3.29	3.28	3.50K	3.29	3.61K	3.28	3.22K	3.28	3.53K	99.87	0.13	<0.01

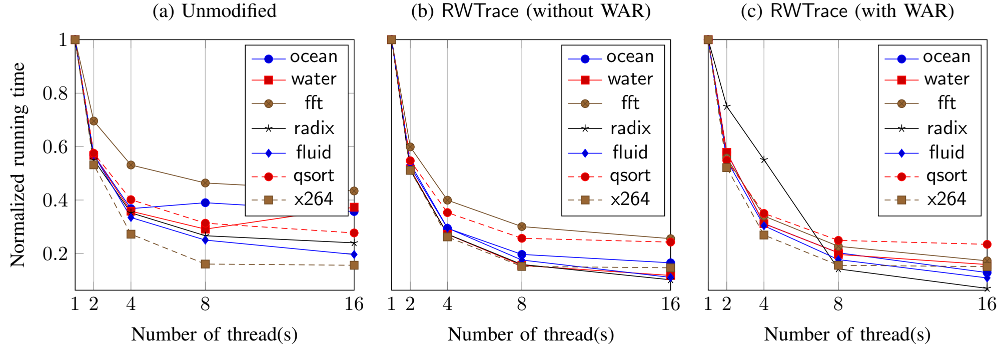

Figure 3. Evaluation of scalability

by RWTrace that only traces RAW and WAW dependences (described in Sections III-C and III-D2), while Columns 7–8 show the data when WAR dependences are also traced (described in Section III-D). Evaluation results for MutexLock and CREWLock are shown in Columns 9–10 and Columns 11–12, respectively. Columns 13–15 exhibit proportions of state transitions in lock-based Octet simulation: allocation or same state (A/S), upgrading or fence (U/F) and conflicting (Conf.).

1) Study of Efficiency: The evaluation results show that RWTrace has the best running time if only RAW and WAW dependences are traced, which is sufficient for an offline replay. In this setting, RWTrace's slow path efficiency is comparable to the lightweight lock approach, and hence outperformed CREWLock which captures exactly the same set of dependences. Compared with data reported in Chimera [3], the runtime overhead of RWTrace is smaller than that for scientific benchmarks evaluated in [3] (our average runtime overhead is 44.6% while the reported is 85.8%).

If WAR dependences are also traced, an additional slowdown of 0.3%–142% (median 39%) is paid. In this setting, RWTrace outperformed or has comparable performance with both MutexLock and CREWLock for 11 out of 12 subjects. The only exception is qsort due to large amount of shared memory writes, and our slow path is slightly less efficient than a lightweight lock. MutexLock exhibits 30–200× of slowdown for ocean, water and fluid because it prohibits concurrent shared memory reads, and our RWTrace does not suffer from such issue.

For a qualitative comparison with Octet, we refer to [29] which points out that Octet incurs large overhead for memory-intensive programs with even moderate communication (0.1–1% Cord., e.g., water and qsort). Note that this ratio is even higher for aget, pbzip2 and knot, however, these applications

are I/O-bound and therefore suffer less from performance penalty of round-trip coordination. Since RWTrace is less optimistic than Octet, it would be less efficient for benchmarks that have numerous non-conflicting writes (e.g., fluid and x264). However, this feature exactly complements Octet's heavy slow path. As suggested in [29], a hybrid implementation combining Octet and RWTrace is expected to be even more efficient as RWTrace is more efficient than locks.

2) Study of Logged Dependences: CREWLock is only capable of tracing RAW and WAW dependences, and the reported amounts of dependences are consistent with RWTrace (1) (Columns 6 and 12 in Table II). To trace WAR dependences, 0.4–108% (median 39%) more dependences are logged (Column 8). These dependences are sufficient for sound predictive trace analysis (e.g., happens-before race detector). Note that MutexLock does not use shadow memory to hold last writer's information, and hence redundant read-after-read dependences are also captured, leading to much more dependences being logged for ocean, water, fluid and x264. Also note that RWTrace captures slightly less dependences compared with CREWLock. This is because RWTrace has less probe effect, and less conflicting accesses are interleaved.

3) Study of Scalability: We also evaluated scalability of the scientific subject programs as they extensively use the shared memory. The evaluation results are shown in Figure 3. As expected, all subjects scaled well because our instrumentation does not pose any scalability bottleneck, and has a wait-free fast path to better recover parallelism. A closer inspection shows a counterintuitive observation that instrumented programs scaled slightly better than the unmodified ones. Further profiling information suggests that this is because RWTrace instrumentation does slowdown the program, but inserted code snippets are mostly scalable thread-local workloads (e.g., cache lookup), and their costs are amortized across parallel threads as processor cores increased. On the contrary, unmodified code hits the scalability bottleneck faster due to extensive shared memory accesses and NUMA architecture of the hardware.

VI. Related Work

Shared memory dependence is useful in various ways. First, it serves a basis of deterministically recording and replaying concurrent program's execution [8]. The logged dependences, after a trace simplification procedure [15], are extremely useful for programmers to debug faulty concurrent programs. Furthermore, these dependences can also be fed to predictive analysis tools for finding potential access anomalies (e.g., by causal race detection [5], [30] or random testing [14]). We are also working on such analysis tools upon the RWTrace framework.

Restoring shared memory dependences from each thread's local information is computational intractable [31], therefore such solutions are either best-effort [32] or have an exponential worst case [33]. The more straightforward approach tracing shared memory dependence is protecting each shared memory accesses with locks. Obtaining dependences is much easier in this setting as all conflicting accesses are serialized by lock. The lock can be implemented either by JVM's built-in synchronize primitives [2], hand-tuned spin lock [34], or hardware page protection [1].

Lock implementation can be optimistic towards successive acquisitions in the same thread [35]. Following an optimistic design, Octet [9] made the thread-local fast path extremely efficient, but pays heavy roundtrip coordination cost at slow paths. On the other hand, Chimera [3] uses hybrid program analysis to enforce race-freedom by optimistically grouping fine-grained locks to larger granularity ones.

Recent research work also realized that it is sometimes unnecessary to synchronize a shared memory access even if it is incorporated in an inter-thread shared memory dependence. The seminal work Stride [10] achieved eliminating all read-time synchronizations, and exact dependences can be restored by an offline replay pass. Our previous work CARE [11] is also an optimistic approach to tracing a subset of shared memory dependences for deterministic replay.

The algorithm of RWTrace combines both ideas from Stride and CARE, and extended them with several new designs such that precise dependences are traced on the fly. Compared with traditional lock-based approaches, we achieved a wait-free fast path and does not sacrifice slow path efficiency, thus the dependence tracing procedure is more efficient. Compared with Octet, our slow path (lock protected critical section) is much more efficient than its roundtrip coordination protocol. We also capture more precise write-after-read dependences than Octet because its RdSh $\rightarrow$ WrEx state transition is handled by a global roundtrip coordination, which enforces happens-before ordering but the exact reader information is lost. Compared with Stride and CARE which deduce shared memory dependences by a replay pass, we capture them in an online fashion. When a faithful replay is infeasible (e.g., missed dependences in the library code or huge amount of I/O operations which cannot be efficiently logged), we are still able to obtain shared memory dependences for further analysis.

Finally, our discussions on relaxed memory model extends the correctness of Stride and CARE to x86-TSO, as they only considered sequentially consistent memory model.

VII. Conclusion

In this paper, we present an optimistic shared memory dependence tracing scheme to effectively and efficiently tracing inter-thread shared memory dependences on the fly. Specifically, we addressed the challenge of capturing the precise write-after-read dependence between a pair of unsynchronized operations, achieving a wait-free fast path for thread-local reads. We implemented our algorithms as RWTrace tool and made it open source to enable more useful analysis tools on this platform.

Acknowledgment

This work was supported in part by National Basic Research 973 Program (Grant No. 2015CB352202), and National Natural Science Foundation (Grant Nos. 61472174, 91318301, 61321491) of China.

REFERENCES

[1] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution replay of multiprocessor virtual machines," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE, 2008, pp. 121–130.

[2] J. Huang, P. Liu, and C. Zhang, "LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, ser. FSE, 2010, pp. 207–216.

[3] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy, "Chimera: Hybrid program analysis for determinism," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, ser. PLDI, 2012, pp. 463–474.

[4] J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, 2014, pp. 337–348.

[5] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound predictive race detection in polynomial time," in *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL, 2012, pp. 387–400.

[6] C.-S. Park and K. Sen, "Randomized active atomicity violation detection in concurrent programs," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16, 2008, pp. 135–145.

[7] S. Park, S. Lu, and Y. Zhou, "Ctrigger: Exposing atomicity violation bugs from their hiding places," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV, 2009, pp. 25–36.

[8] "RecPlay: A fully integrated practical record/replay system," *ACM Transactions on Computer Systems*, vol. 17.

[9] M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. Fathi Salmi, S. Biswas, A. Sengupta, and J. Huang, "Octet: Capturing and controlling cross-thread dependences efficiently," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13, 2013, pp. 693–712.

[10] J. Zhou, X. Xiao, and C. Zhang, "Stride: Search-based deterministic replay in polynomial time via bounded linkage," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE, 2012, pp. 892–902.

[11] Y. Jiang, T. Gu, C. Xu, X. Ma, and J. Lu, "CARE: Cache guided deterministic replay for concurrent Java programs," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, 2014, pp. 457–467.

[12] M. Herlihy and N. Shavit, "On the nature of progress," in *Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*, 2011, pp. 313–328.

[13] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors," *Communications of the ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010.

[14] K. Sen, "Race directed random testing of concurrent programs," in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI, 2008, pp. 11–21.

[15] N. Jalbert and K. Sen, "A trace simplification technique for effective debugging of concurrent programs," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, ser. FSE, 2010, pp. 57–66.

[16] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[17] Z. Yang, M. Yang, L. Xu, H. Chen, and B. Zang, "ORDER: Object centric deterministic replay for java," in *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, ser. ATEC, 2011, pp. 30–43.

[18] J. Silva, J. Simao, and L. Veiga, "Ditto - deterministic execution Replayability-as-a-Service for Java VM on multiprocessors," in *Middleware 2013*, ser. Lecture Notes in Computer Science, D. Eyers and K. Schwan, Eds. Springer Berlin Heidelberg, 2013, vol. 8275, pp. 405–424.

[19] R. H. B. Netzer, "Optimal tracing and replay for debugging shared-memory parallel programs," in *Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, ser. PADD, 1993, pp. 1–11.

[20] C. Flanagan and S. N. Freund, "Fasttrack: Efficient and precise dynamic race detection," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, 2009, pp. 121–133.

[21] M. Xu, M. D. Hill, and R. Bodik, "A regulated transitive reduction (RTR) for longer memory race recording," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS, 2006, pp. 49–60.

[22] Y. Liu, K. Zhang, and M. Spear, "Dynamic-sized nonblocking hash tables," in *Proceedings of the 2014 ACM symposium on Principles of distributed computing*. ACM, 2014, pp. 242–251.

[23] The LLVM compiler infrastructure. http://llvm.org/.

[24] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[25] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, June 1995, pp. 24–36.

[26] ApacheBench. http://httpd.apache.org/docs/current/programs/ab.html.

[27] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with "readers" and "writers"," *Communications of the ACM*, vol. 14, no. 10, pp. 667–668, Oct. 1971.

[28] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, 2014.

[29] M. Cao, M. Zhang, and M. D. Bond, "Drinking from both glasses: Adaptively combining pessimistic and optimistic synchronization for efficient parallel runtime support," in *5th Workshop on Determinism and Correctness in Parallel Programming*, ser. WoDet, 2014.

[30] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems*.

[31] P. B. Gibbons and E. Korach, "Testing shared memories," *SIAM J. Comput.*, vol. 26, no. 4, pp. 1208–1244, Aug. 1997.

[32] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "PRES: probabilistic replay with execution sketching on multiprocessors," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP, 2009, pp. 177–192.

[33] J. Huang, C. Zhang, and J. Dolby, "CLAP: Recording local executions to reproduce concurrency failures," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI, 2013, pp. 141–152.

[34] Y. Chen and H. Chen, "Scalable deterministic replay in a parallel full-system emulator," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP, 2013, pp. 207–218.

[35] K. Russell and D. Detlefs, "Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06, 2006, pp. 263–272.