2020-07-31

ZipperOTF: Automatic, Precise, and Simple Data Race Detection for Task Parallel Programs with Mutual Exclusion

S. Jacob Powell
Brigham Young University

ZipperOTF: Automatic, Precise, and Simple Data Race Detection for Task Parallel Programs with Mutual Exclusion

S. Jacob Powell

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Eric Mercer, Chair
Sean Warnick
Kevin Seppi

Department of Computer Science

Brigham Young University

ABSTRACT

ZipperOTF: Automatic, Precise, and Simple Data Race Detection for
Task Parallel Programs with Mutual Exclusion

S. Jacob Powell
Department of Computer Science, BYU
Master of Science

Data race in parallel programs can be difficult to precisely detect, and doing so manually can often prove unsuccessful. Task parallel programming models can help reduce defects introduced by the programmer by restricting concurrent functionalities to fork-join operations. Typical data race detection algorithms compute the happens-before relation either by tracking the order that shared accesses happen via a vector clock counter, or by grouping events into sets that help classify which heap locations are accessed sequentially or in parallel. Access sets are simple and efficient to compute, and have been shown to have the potential to outperform vector clock approaches in certain use cases. However, they do not support arbitrary thread synchronization, are limited to fork-join or similar structures, and do not support mutual exclusion. Vector clock approaches do not scale as well to many threads with many shared interactions, rendering them inefficient in many cases. This work combines the simplicity of access sets with the generality of vector clocks by grouping heap accesses into access sets, and attaching the vector clock counter to those groupings. By combining these two approaches, access sets can be utilized more generally to support programs that contain mutual exclusion. Additionally, entire blocks can be ordered with each other rather than single accesses, producing a much more efficient algorithm for data race detection. This novel algorithm, ZipperOTF, is compared to the Computation Graph algorithm (an access set algorithm) as well as FastTrack (a vector clock algorithm) to show comparisons in empirical results and in both time and space complexity.

Keywords: data race, concurrent, detection, access sets, shadow memory, vector clocks, parallel, task parallel, model checking

ACKNOWLEDGMENTS

I express my gratitude to my advisor Eric Mercer for all of his unfailing help and feedback during the development of this thesis. I also thank my family for their continuous support and encouragement in completing this work.

Table of Contents

List of Figures **vi**

List of Tables **vii**

1 Introduction **1**

2 Background **4**

2.1 Program Executions as Histories 4

2.2 The Happens-Before Relation 6

2.3 HB and Computation Graphs 7

2.4 HB and Vector Clocks 10

2.5 Comparison 11

3 ZipperOTF **13**

3.1 Greedy-Scheduled Histories 13

3.2 Sync Maps 14

3.3 Algorithm 15

3.4 Example 19

4 Implementation **24**

5 Results **27**

5.1 Benchmarks and Experimentation 27

5.2 Complexity 27

5.3 Performance Comparison 29

6 Related Work **32**

7 Conclusion **34**

References **35**

List of Figures

2.1 Transition rules for CG. 9

3.1 Transition rules for ZipperOTF. 18

3.2 The observed history and a greedy-scheduled reordering of the observed history 20

3.3 The Computation Graph built from the Greedy-Scheduled History 21

3.4 Each point where data race checks occur; green dashed lines denote the checking node and blue dotted lines denote the nodes being checked against 22

List of Tables

5.1 Results 31

Chapter 1

Introduction

Data race occurs in a program when two or more threads access a shared memory location consecutively and at least one of the accesses is a write. A program has a data race if there is some execution of the program that contains a data race. *Task parallel* programs are a subset of concurrent programming models that help the programmer reduce the number of bugs in a program by restricting actions to be more predictable. This work focuses on the topic of dynamic data race detection in task parallel programs.

Dynamic data race detection involves detecting data race in a program by observing an execution. This is usually achieved by computing Lamport's Happens-Before (HB) relation [27] over the observed program events. The soundness guarantee of the HB relation states that any events unordered by HB are logically concurrent and can be executed consecutively in some execution of the program. Therefore, dynamic analyses can report conflicting memory access events as data races if they are unordered with respect to the HB relation.

Many state-of-the-art data race detection algorithms use vector clock counters [31] to represent the HB relation [17, 39]. A vector clock consists of an integer counter for each shared resource in the program and can be compared point-wise with other vector clocks. The updates at each program event are such that a shared variable's vector clock is less than the current thread's vector clock if and only if the previous access events to the shared variable are all HB related to the current thread's access event. Vector clocks possess the ability to detect data race in many different types of programs including those with mutual exclusion. Although many optimizations have been developed for vector clock algorithms,

the space and time overhead of the analysis can become intractable as the number of threads and shared resources in the program grow [1, 9, 17, 31, 39].

Task parallel programming models, such as OpenMP and Habanero Java, restrict how threads can synchronize, resulting in a more predictable parallel control flow that helps developers write idiomatic code. This predictable structure can also be exploited by dynamic analyses to detect data race more efficiently. Access set algorithms group sequential accesses in each concurrent task into a single block called an *access set*. If two access sets are unordered by the HB relation, they are then intersected to check for data race. A program has a data race if the intersection of two concurrent access sets is non-empty [14].

Intersecting access sets reduces much of the inefficiencies that vector clock approaches introduce immensely, but restricts the class of computable programs. The first limitation that access set approaches require is that the program follow a fork-join structure. This means that parent threads must reap child threads, and handles to threads can not be passed around. The second limitation prevents the program from containing mutual exclusion; adding in locking mechanisms introduces complexity in computing the HB relation and renders the optimizations of access set algorithms imprecise. Some algorithms such as *SP-Bags*, *ESP-Bags*, *Nondeterminator*, or *TARDIS*, for example, improve and extend access set techniques to be more optimized or to handle futures or other constructs, but each sacrifice generality, efficiency, or correctness to do so [3, 8, 14, 22, 32, 34, 37, 38, 40–44, 47].

The work in this thesis, *ZipperOTF*, utilizes a novel data structure, called a *sync map matrix*, that draws upon vector clock techniques to represent the HB relation for programs that contain mutual exclusion. It constructs a graph structure called a computation graph that follows the same patterns and rules as the Computation Graph (CG) algorithm introduced by Nakade *et. al.* [34]. The construction of the graph groups accesses into sets that are contained in nodes in the graph and each node represents a series of accesses and/or a synchronization event. Similar to the vector clock counters utilized by FastTrack, ZipperOTF's sync maps contain information that is used to check whether two points in the

program execute concurrently; the difference however is that sync maps are placed on nodes in the computation graph rather than accesses directly, and paths between nodes are used to determine which access sets must be intersected to check for data race.

This utilization of the computation graph and vector clock techniques both simplifies the data structures required to track synchronization points while alleviating the negative symptoms found with each approach separately. Access sets allow ZipperOTF to coalesce accesses into groups and therefore reduce the number of sync maps down to the number of threads in the program rather than having multiple vector clocks for each access. By borrowing the synchronization techniques utilized in vector clocks in the form of sync maps, calculating the order of two access sets is reduced to a simple lookup. ZipperOTF is able to detect or prove the absence of data race in task parallel programs precisely and more efficiently than many state-of-the-art access set and vector clock approaches, including programs with mutual exclusion. ZipperOTF is shown to be comparable in empirical experiments to state of the art data race detection as well as possessing improvements in space and time complexity for certain types of programs.

Chapter 2

Background

2.1 Program Executions as Histories

ZipperOTF reasons over a *history*, which is simply a set of events paired with a total order. A history can be built from any program execution by recording each program action in the order that it occurs during execution. ZipperOTF specifically focuses on thread and memory events, which contain the needed information for accurate data race detection. The formalism used here is an extension of the program history definition given in the presentation on the weaker partial order presented by Ogles *et. al.*[36].

Each thread can perform either a *thread* action or a *memory* action. A thread action is any member of the finite set $P = \{frk, jn\}$, where *frk* and *jn* are symbols for *fork* and *join* respectively. Thread actions operate on the finite set T, which contains all thread IDs found in the history; a fork action creates new threads, and the join action reaps threads once they have completed. Memory actions are any member of the finite set $L = \{rd, wrt, acq, rel\}$.

Any given history consists of thread events of the form $e_t \in T \times P \times T$, and memory events of the form $e_m \in T \times L \times M$, where M is the finite set of memory locations. For an event $e = \langle t, l, m \rangle$, *read (rd)* and *write (wrt)* are *access* actions l that correspond to reading or updating the value of a certain memory location m on the thread t. Similarly, *acquire (acq)* and *release (rel)* are *lock* actions l that correspond to locking and unlocking operations for mutual exclusion on a memory location m. Both access and lock actions are memory actions because they operate on any member of the finite set of memory locations, M. Therefore the possible actions in any event in a history are any member of the set $A = P \cup L$.

For a given thread event $e = (t, p, t')$, the thread can not operate on itself, and therefore $t \neq t'$. Additionally it is said that t is the *forking* thread and t' is the thread that is *forked* if $p = \mathit{frk}$, and likewise t is the *joining* thread and t' the *joined* thread if $p = \mathit{jn}$.

The total order for the observed events in a history H is given by $<_H$. The notation $H|_t$ is a set of events obtained by projecting events in the history H that belong to the target thread t. Two events are *thread ordered* in a history, $e_i <_{TO} e_j$, if and only if they occur in order on the same thread: $\exists t \in T\ (e_i \in H|_t \wedge e_j \in H|_t \wedge e_i <_H e_j)$.

For convenience, the function $\mathit{tid}(e)$ returns the thread to which the event e belongs ($e \in H|_t$, where H is a history). The function $\mathit{frk}(t', H)$ gives the thread event in H that creates t' and is undefined if no such event is in H or more than one event creates t' in H. The function $\mathit{jn}(t', H)$ is defined similarly for the event that joins with t' in the history. Every history $H = \{e_0, \ldots\}$ has a *master thread*, $t_o = \mathit{tid}(e_0)$, that starts the history, for which $\mathit{frk}(t_o, H)$ and $\mathit{jn}(t_o, H)$ are undefined. The functions $\mathit{isFork}(e)$, $\mathit{isJoin}(e)$, $\mathit{isRead}(e)$, $\mathit{isWrite}(e)$, $\mathit{isAcq}(e)$, and $\mathit{isRel}(e)$ each return true if the event is of action type frk, jn, rd, wrt, acq, or rel, respectively. Additionally, the function $\mathit{rel}(e_i, H)$ for an acquire event $e_i \in H \wedge \mathit{isAcq}(e_i)$, gives the subsequent release event, $e_j \in H$, from the same thread $e_i <_H e_j \wedge \mathit{isRel}(e_j) \wedge \mathit{tid}(e_i) = \mathit{tid}(e_j)$. If $\neg \mathit{isAcq}(e_i)$ or no such event exists, $\mathit{rel}(e_i, H)$ is undefined.

A history is *well-formed* if and only if its threads are *thread consistent* and *lock consistent*. A thread t is thread consistent if and only if $e_f = \mathit{frk}(t, H)$ and $e_j = \mathit{jn}(t, H)$ are defined, the thread is joined by the forking thread $\mathit{tid}(e_f) = \mathit{tid}(e_j)$, and all of its events in the history happen after it is forked and before it is joined: $\forall e_i \in H|_t : e_f <_H e_i <_H e_j$. A history H is said to be thread consistent if and only if $\forall t \in T$: t is thread consistent, excluding the master thread t_o.

A history is said to be lock consistent if and only if $\mathit{rel}(e_i)$ is defined for all acquire events $\forall e_i \in H : \mathit{isAcq}(e_i)$ and there are no interleaving acquire or release events: $\forall e_i, e_k \in H : \mathit{isAcq}(e_i) \wedge \neg \mathit{isAcq}(e_k) \wedge \neg \mathit{isRel}(e_k) \wedge e_i <_H e_k <_H \mathit{rel}(e_i)$. Additionally, there are no

release events without a matching acquire: $\forall e_j, \exists e_i : isRel(e_j) \wedge rel(e_i) = e_j$ and all lock actions apply to a single memory location m.

Two access events e_i, e_j on the same memory location are said to conflict, $e_i \asymp e_j$, if they originate from different threads and at least one is a write action: $tid(e_i) \neq tid(e_j) \wedge (isWrite(e_i) \vee isWrite(e_j))$. A history H is said to *witness* a data race if two events e_i and e_j are concurrent in H and conflict $e_i \asymp e_j$. Two events are *concurrent* in a history H, if they are on different threads and there exists some feasible history H' generated from the same program where the events appear in a different order. Being concurrent in the history guarantees that there are no intervening lock actions that would order the events; methods of determining if two events are concurrent are discussed in the following sections. A program exhibits data race if there is a history generated from that program that witnesses a data race.

2.2 The Happens-Before Relation

The Happens-Before (HB) relation is a partial order that can be used to determine which events in history execute sequentially, even if they are on different threads, and therefore not be checked for data race. For example, an acquire event that happens after a previous release event is HB ordered with the release event, since the acquire can only execute after the release event. Similarly, any events on a thread created by a fork event are ordered with that event. Two reads on different threads, however, may be processed in certain histories in either order, and are therefore unordered by the HB relation.

The HB relation $<_{HB}$ is the least strict partial order on events in a history that includes $<_{TO}$ and satisfies the following: $\forall e_i, e_j \in H$:

$isRel(e_i) \wedge isAcq(e_j) \wedge \nexists e_k \in H : (isAcq(e_k) \vee isRel(e_k)) \wedge e_i <_H e_k <_H e_j$, then $e_i <_{HB} e_j$

Like other partial orders, the HB relation is transitive, reflexive, and asymmetric.

The HB relation is an effective way to compute order between two events in a history and therefore determine which events are concurrent. Concurrent events can then be compared

to check for conflict; if a conflict is present, a data race is reported. There are many ways to compute the HB for task parallel programs; included in this thesis are two methods that help demonstrate how ZipperOTF computes the HB relation.

2.3 HB and Computation Graphs

One way to compute the HB relation is with a computation graph as demonstrated by the CG algorithm [34]. Access set algorithms collect sequential events on each thread into sets of reads and writes to shared resources which are then intersected to detect concurrency. The CG algorithm is an example of an access set algorithm that builds a Directed Acyclic Graph (DAG) with each node containing the sequentially grouped events and the edges representing any HB ordered relationships between those sets. The computation graph definition here is identical to the CG algorithm; $\mathcal{P}(M)$ is the power set of M.

Definition 1 *A computation graph is a directed acyclic graph (DAG),* $G = \langle N, E, R, W \rangle$*, where*

- N *is a set of nodes;*
- $E \subseteq N \times N$ *is a set of directed edges;*
- $R : (N \mapsto \mathcal{P}(M))$ *maps* N *to the unique identifiers for the shared locations* M*, read by the tasks; and*
- $W : (N \mapsto \mathcal{P}(M))$ *maps* N *to the unique identifiers for the shared locations* M*, written by the tasks.*

A computation graph can be built from a history H by adding events into the current access set in the node until a synchronization event such as fork, join, acquire, or release separates the sets into new nodes. Synchronization events are not inserted into any node, but are represented as edges in the graph and separations between nodes based on the HB order of the events; therefore new node(s) are created upon each occurrence of a synchronization

event. Any two nodes' access sets can be intersected to find common shared resource accesses, and if they are unordered, a data race is reported.

The construction of a computation graph from a history follows a set of state transition rules, continuing to follow any matching state transitions until none match. The state used in the CG transition rules is represented as the tuple $\langle G, \mathit{last}, C, H \rangle$. G is a computation graph being constructed that is itself a tuple, as defined above: $\langle N, E, R, W \rangle$. The variable *last* represents the node at the time of the most recent release in the history. The function C maps thread IDs to the latest node on that thread: $C : T \times N$, where T is the set of all threads in the history H.

The functions and elements in the state such as G, R, W, and C, can be updated in-place using the $\mapsto$ and $\overset{\cup}{\mapsto}$ operators. The statement $C = C\,[t \mapsto n]$ updates the function C to be exactly the same as it was before, but with the thread $t \in T$ now mapped to $n \in N$, regardless of what it was mapped to previously. The members of the computation graph tuple G can be updated using the following notation; $G\,[E \mapsto E \cup \{\langle a, b \rangle\}]$ for example unions the edge $\langle a, b \rangle$ into the edge set within G, leaving the rest of G unchanged. The $\overset{\cup}{\mapsto}$ operator can be used as shorthand to update the map to reference the same set as before, unioned with a new set of values. For example, $R \mapsto R\left[C(t) \overset{\cup}{\mapsto} \{m\}\right]$ updates the R function's mapping from $C(t) \in N$ to map to the same set as before unioned with m.

The graph is built from a history according to the state transitions in Figure 2.1. In the transition rules the history is represented as a sequence such as $\langle e_0\ e_1\ e_2\ \ldots \rangle$, where e_0 is the first event in the sequence, e_1 is the second, and so on; this sequence is derived from the total order on the history $<_H$ and its set of events, H, as defined previously. Ellipses are used to denote zero or more events: $\langle e_0\ \ldots \rangle$. The helper function *fresh*() creates new nodes, and $\mathit{tid}(n)$ retrieves the thread that resulted in the creation of the node n (similar to $\mathit{tid}(e)$ on an event e). The set of nodes N is totally ordered, $<_N$, in the order in which they are created by *fresh*(). The edges in the computation graph keep track of fork and join operations as well as inter-thread order via lock actions. Each transition is triggered by an

observed event in the history where the left side of $\rightarrow$ is the current state, and the right side is the new state. Events in the history H are removed once they are processed by the current state transition rule, allowing the subsequent rule to match on the next event.

READ ACCESS

$$\langle G, last, C, \langle\langle t, rd, m\rangle\ e_0\ \ldots\rangle\rangle \rightarrow \quad \langle G\left[R \mapsto R\left[C(t) \overset{\cup}{\mapsto} \{m\}\right]\right], last, C, \langle e_0\ \ldots\rangle\rangle$$

WRITE ACCESS

$$\langle G, last, C, \langle\langle t, wrt, m\rangle\ e_0\ \ldots\rangle\rangle \rightarrow \quad \langle G\left[W \mapsto W\left[C(t) \overset{\cup}{\mapsto} \{m\}\right]\right], last, C, \langle e_0\ \ldots\rangle\rangle$$

FORK

$$\frac{n_l, n_r = fresh() \qquad E' = E \cup \{\langle C(t), n_l\rangle, \langle C(t), n_r\rangle\} \qquad C' = C\left[t \mapsto n_l\right]\left[t' \mapsto n_r\right]}{\langle G, last, C, \langle\langle t, frk, t'\rangle\ e_0\ \ldots\rangle\rangle \rightarrow \quad \langle G\left[E \mapsto E'\right], last, C', \langle e_0\ \ldots\rangle\rangle}$$

JOIN

$$\frac{n_j = fresh() \qquad E' = E \cup \{\langle C(t), n_j\rangle, \langle C(t'), n_j\rangle\} \qquad C' = C\left[t \mapsto n_j\right]}{\langle G, last, C, \langle\langle t, jn, t'\rangle\ e_0\ \ldots\rangle\rangle \rightarrow \quad \langle G\left[E \mapsto E'\right], last, C', \langle e_0\ \ldots\rangle\rangle}$$

ACQUIRE

$$\frac{n_a = fresh() \qquad E' = E \cup \{\langle C(t), n_a\rangle, \langle last, n_a\rangle\} \qquad C' = C\left[t \mapsto n_a\right]}{\langle G, last, C, \langle\langle t, acq, m\rangle\ e_0\ \ldots\rangle\rangle \rightarrow \quad \langle G\left[E \mapsto E'\right], last, C', \langle e_0\ \ldots\rangle\rangle}$$

RELEASE

$$\frac{n_r = fresh() \qquad E' = E \cup \{\langle C(t), n_r\rangle\} \qquad last' = C(t) \qquad C' = C\left[t \mapsto n_r\right]}{\langle G, last, C, \langle\langle t, rel, m\rangle\ e_0\ \ldots\rangle\rangle \rightarrow \quad \langle G\left[E \mapsto E'\right], last', C', \langle e_0\ \ldots\rangle\rangle}$$

Figure 2.1: Transition rules for CG.

By applying these rewrite rules, the CG algorithm builds a graph from an execution while expanding the state via these transitions. When no more transitions can be taken, the graph is complete and ready for analysis.

The conflict operator $\asymp$ is redefined to work with a pair of nodes (rather than events), a and b, and is defined across the R and W sets of a graph G as such: $a \asymp b \leftrightarrow (R(a) \cap W(b)) \cup (W(a) \cap R(b)) \cup (W(a) \cap W(b)) \neq \emptyset$

The CG algorithm utilizes a naïve method for computing whether two nodes are concurrent. Each node is compared with any unordered node that comes later in a topologically sorted list of all nodes. The nodes are sorted topologically so that it is guaranteed that subsequent nodes occur either concurrently or after the current node in the list. A reachability check is then performed on each node with each of the nodes following it in the sorted list of nodes. If they are found to be unreachable from each other, then they are concurrent and unordered by the HB relation. If two nodes a and b are unordered and conflict $a \asymp b$, then a data race is reported.

2.4 HB and Vector Clocks

Vector clocks are another common approach to computing the Happens-Before (HB) relation on a program history. Each shared variable and thread is assigned a vector clock for each type of update or communication. By updating the necessary vector clock for each event, the HB order is computed and the clocks can then be used to check for data race.

A vector clock is a function $VC : T \times \mathbb{N}$ that maps threads to counter values. Multiple vector clocks are maintained in order to track concurrent interactions between resources. The HB partial order is tracked by a clock B_t for each thread t; reads and writes to each memory location x are tracked by the clocks R_x and W_x respectively; and finally L_x^r and L_x^w track reads and writes on each variable x respectively that occur while covered by a lock. Therefore each thread has a vector clock, and each memory location has four clocks.

The integer value for a thread t' in the HB clock for thread t is obtained as follows: $B_t(t') \in \mathbb{N}$. A vector clock B_x is ordered with another clock B_y if it is point-wise comparable: $B_x \sqsubseteq B_y \longleftrightarrow \forall t \in T(B_x(t) \leq B_y(t)) \wedge \exists t \in T(B_x(t) < B_y(t))$. Two vector clocks *merge*, $\sqcup$, by performing a point-wise comparison of their integer counters to produce a clock with

each index containing the greater value of the two. For example: $B_t \leftarrow B_t \sqcup L_x^w$ updates the vector clock function B_t by merging it with L_x^w, and $R_x(t') \leftarrow B_t(t')$ updates the specific value at t' in R_x.

On each read and write event, a data race check occurs. On each read event $\langle t, rd, x\rangle$ in a history, if the write clock for x is unordered with the HB clock for the current thread $W_x \not\sqsubseteq B_t$, then a data race is reported. Likewise for write events $\langle t, wrt, x\rangle$, if the access event is concurrent with previous accesses $W_x \not\sqsubseteq B_t \vee R_x \not\sqsubseteq B_t$, then a race has occurred.

For each read event $\langle t, rd, x\rangle$ or write event $\langle t, wrt, x\rangle$, two clocks are updated. The read and write clocks are updated in each case if there is no race: $R_x(t) \leftarrow B_t(t)$ for reads and $W_x(t) \leftarrow B_t(t)$ for writes. On $\langle t, frk, t'\rangle$ events, B_t is copied to the forked thread $B_{t'} \leftarrow B_t$ and both clocks' entries for themselves are incremented $B_t(t) \leftarrow B_t(t)+1$ and $B_{t'}(t') \leftarrow B_{t'}(t')+1$. The $B_t(t)$ entry is also incremented on $\langle t, rel, x\rangle$ events $B_t(t) \leftarrow B_t(t) + 1$.

Vector clocks merge at specific synchronization points in the history. If a $\langle t, rd, x\rangle$ or $\langle t, wrt, x\rangle$ event was done in mutual exclusion, B_t is updated: $B_t \leftarrow B_t \sqcup L_x^w$ for reads and $B_t \leftarrow B_t \sqcup L_x^w \sqcup L_x^r$ for writes. On $\langle t, jn, t'\rangle$ events, the threads' HB clocks merge: $B_t \leftarrow B_t \sqcup B_{t'}$.

And finally, a set of variables accessed in mutual exclusion is maintained: $V_r \leftarrow V_r \cup \{x\}$ on $\langle t, rd, x\rangle$ events and $V_w \leftarrow V_w \cup \{x\}$ for $\langle t, wrt, x\rangle$ events. These sets are then used to update L_x^r and L_x^w on each $\langle t, rel, m\rangle$ event: $\forall x \in V_r : L_x^r \leftarrow L_x^r \sqcup B_t$ and $\forall x \in V_w : L_x^w \leftarrow L_x^w \sqcup B_t$, and are then cleared after they are used: $V_r \leftarrow V_w \leftarrow \emptyset$.

2.5 Comparison

The two approaches described here have different advantages and disadvantages when it comes to detecting data race in different types of parallel programs. Access set algorithms typically focus more on specialization to boost efficiency, while vector clock algorithms boast a wide generality. In all cases there are specific cases where these different approaches struggle to perform well.

The CG algorithm and access set algorithms typically focus on specific types of parallel programs such as task parallel programs. They focus on generalizing to a wider set of task parallel programs by handling programs that have mutual exclusion. While it is able to handle a wider class of programs without sacrificing correctness, the CG algorithm struggles with time complexity due to its need to perform a topological sort and reachability check to check for concurrency. Because of this, it has a time complexity of $O(N^3 + N^2V)$, where N is the number of nodes in the graph and V is the number of shared variables, and its space complexity is $O(T^2 + NV)$.

Vector clocks provide many advantages when detecting data race in a concurrent program. They are not limited to task parallel programs and boast a time complexity of $O(MT^2)$ where M is the number of shared variable accesses and T is the number of threads. The comparison of two vector clocks is also efficient, but as the number of threads and shared memory locations grow, the space complexity becomes infeasible with a space complexity of $O(T^2 + VT)$.

ZipperOTF gains inspiration from vector clocks in the way that it represents its *sync map matrix*, which is ZipperOTF's method of tracking concurrency between threads in a computation graph. As will be discussed in further detail below, the number of sync maps can be significantly less than the vector clock's requirement, requiring a single sync map for each thread in the program due to its ability to be computed during the execution of a program, or *on-the-fly*, and reuse the space used by sync maps.

Chapter 3

ZipperOTF

3.1 Greedy-Scheduled Histories

ZipperOTF requires the history it processes to be *greedy-scheduled*, which ensures that all events on a thread will be visited in order with no interleaving actions until that thread is blocked and must switch context. This requirement guarantees that no needed information is missed at each point where a data race check occurs.

Greedy-scheduled histories are feasible executions of a program that preserve the Read-Preserving HB order (RP). The RP relation $<_{RP}$ is the smallest partial order over events in a history H that includes $<_{HB}$ and for all write/read pairs that are ordered in $<_H$, the same pairs are ordered in RP:

$$let\ w = \langle t, wrt, m\rangle,\ r = \langle t', rd, m\rangle,\ o = \langle t'', wrt, m\rangle$$

$$\forall w, r \in H : (w <_H r\ \wedge t \neq t' \wedge\ \nexists o \in H : w <_H o <_H r) \implies w <_{RP} r$$

A *write/read pair* as shown above is defined as a pair of events $w, r \in H$ such that they are a write and read on the same variable on different threads with no interleaving accesses. Reordering an observed history into a greedy-scheduled history must preserve the RP order, otherwise it is possible that read actions of a shared variable could read a different value than originally observed which consequently can change the structure of the history from that point forward.

Subsequently, a *greedy-scheduled* history is defined as a total order $<_{GS}$ on $<_{RP}$ such that for any two unordered events, the threads only switch in between two adjacent events e_i and e_j in the history if the next event is an acquire or join: $e_i, e_j \in H : e_i <_H e_j \wedge (tid(e_i) =$

$tid(e_j) \vee isAcq(e_j) \vee isJoin(e_j)$). This constraint ensures that all events on each thread have been visited before processing each *acq* or *jn* event, which guarantees that no accesses have been missed that are needed to perform a data check.

It is guaranteed that any history has at least one feasible execution that is a greedy-scheduled history, as long as the history is well-formed. This guarantee is due to the fact that there are no control flow differences between an observed history and its greedy-scheduled history because $<_{GS}$ contains $<_{RP}$, therefore they both contain all of the same events and maintain order for all write/read pairs. ZipperOTF considers only those schedules that are greedy-scheduled to guarantee that all actions on a thread will be visited in order with no interleaving actions until that thread is blocked. By observing events in this way, it is guaranteed that the check for data race at synchronization points can be performed with correctness because all of the relevant accesses have been observed and can therefore be intersected with parallel events without missing possible data races.

3.2 Sync Maps

ZipperOTF combines the advantages of the explicit graph representation of the HB relation in the CG algorithm and the implicit representation of the HB relation in vector clocks. Instead of performing a reachability check like the CG analysis, ZipperOTF treats nodes as if they themselves are vector clock counters that indicate where the threads last synchronized. This novel data structure is called a *sync map matrix* in ZipperOTF, and represents the main contribution introduced in this work to simplify and optimize data race detection for task parallel programs.

ZipperOTF maintains a sync map matrix S that contains a sync map for each thread. S is a matrix $S : T \times T \times N$ that maps a pair of threads to nodes. In other words, each thread has a sub-matrix $S(t) = T \times N$ that is the sync map for a thread t. Furthermore, $S(t)(t')$ returns the earliest created node on thread t' that is concurrent with the thread t. In this presentation, the sync map matrix S is referenced and updated using typical indexing

and assignment; for example, $S(t)(t') = n$, where S is the same as before, with the sync map for thread t now relating ("mapping") the thread t' to the node n.

The general use of sync maps by the ZipperOTF algorithm consists of creating and updating the sync map matrix at specific points (these points are more specifically defined in the Algorithm section). Sync maps are created at each fork; the forking thread t has an entry in the sync map matrix $S(t)(t')$ that points to the node created on the forked thread t'; likewise, there is an entry $S(t')(t)$ that points to the node created on the forking thread t. Then as ZipperOTF continues, when it checks for data race it guarantees that nodes in t are concurrent with the node $S(t)(t')$ and all of its children. Whenever an *acq* event is observed on the current thread, the sync map from the thread that the previous rel event occurred on is merged into the current thread's sync map. This synchronizes the current thread with any new information gathered since the sync map was created.

Like vector clocks, sync maps provide a simple compressed form of storing synchronization information that allows for a quick way to check for concurrency. The sync maps are updated in all of the same places as vector clocks, with some minor differences. Unlike vector clocks, sync maps map a thread to a single node that contains an access set made up of many reads and writes on many variables, and so are not dependent on the number of accesses.

ZipperOTF builds a graph data structure as is done by the CG algorithm, but because it only processes greedy-scheduled traces, it is able to check a partially constructed graph for data race rather than waiting for the entire graph to be constructed. Ultimately, sync maps help ZipperOTF identify concurrent nodes more quickly because it does not need to perform a reachability check like the CG algorithm.

3.3 Algorithm

The ZipperOTF algorithm is expressed below in the form of state transition rules. These transition rules extend the CG transition rules, denoted by $\rightarrow_{CG}$, and so this symbol is used in the definitions below. The computation graph is built in the same manner as CG, and the

method for detecting data race draws on vector clock techniques, as shown in the transition rules.

The state of the ZipperOTF algorithm is defined as the tuple $\langle G, C, H, S \rangle$. The symbols G, C, and H are the same as defined by the CG transition rules, and are updated in the same way as defined by $\rightarrow_{CG}$. S is the novel sync map matrix that tracks concurrent nodes.

Nodes in the graph $G(N)$ are totally ordered $<_N$, just as they are in CG; therefore comparison $<$ and $max(n, n')$ are used. The operator $S(t) \sqcup S(t')$ (redefined here from a vector clock merge) performs a merge operation on two sync maps, $S(t)$ and $S(t')$, where S is the sync map matrix, and t and t' are threads. Because a sync map is a sub-matrix with two columns $S(t) : T \times N$, it can be treated as a set where each row in the matrix is a tuple $(t \in T, n \in N)$ in the set. Therefore the merge of two sync maps $S(t)$ and $S(t')$ is defined as $S(t) \cup S(t')$, unless $\exists (t'', n) \in S(t) \wedge \exists (t'', n') \in S(t')$, then instead of including both entries in the union operation, only $(t, max(n, n'))$ is included.

The procedure *CheckRace()* receives a node n_a, the thread t that is calling CheckRace for n_a, and parts of the state (G,C, and S) as input, and returns a boolean that represents whether there is a conflict in any of the concurrent nodes, as tracked and identified by the use of the sync map matrix S. The helper function $path(n_1, n_n)$ returns a finite set of nodes that are members of the vertex sequence that make up the directed path from n_a to n_b in G: $\forall n \in N : n \in path(n_1, n_m) \leftrightarrow n \in (n_1, n_2, \ldots, n_m)$ such that $\forall i \in (1, 2, \ldots m-1) : (n_i, n_{i+1}) \in E$. Due to the order and nature of how the graph is built, n_1 and n_n are guaranteed to exist and n_n is guaranteed to be reachable from n_1.

1: **procedure** CheckRace(n_a, t, G, C, S)
2: $\quad I_a = S(t)$
3: $\quad P = \{n_p \mid \forall t' \in T \wedge \forall n_p \in path(I_a(t'), C(t'))\}$
4: $\quad return\ \exists n_b \in P\ s.t.\ n_a \asymp n_b$
5: **end procedure**

Similar to the CG state transition rules, all state transitions are followed until there are no more matching rules. If there are no more matching transitions and the history is not empty, then a data race has been observed; likewise if there are no more events left in the history, then the program is data race free.

READ ACCESS

$$\frac{\langle G, last, C, \langle\langle t, rd, m\rangle\ e_0\ \ldots\rangle\rangle \rightarrow_{CG} \langle G', last', C', \langle e_0\ \ldots\rangle\rangle}{\langle G, last, C, \langle\langle t, rd, m\rangle\ e_0\ \ldots\rangle, S\rangle \rightarrow \quad \langle G', last', C', \langle e_0\ \ldots\rangle, S\rangle}$$

WRITE ACCESS

$$\frac{\langle G, last, C, \langle\langle t, wrt, m\rangle\ e_0\ \ldots\rangle\rangle \rightarrow_{CG} \langle G', last', C', \langle e_0\ \ldots\rangle\rangle}{\langle G, last, C, \langle\langle t, wrt, m\rangle\ e_0\ \ldots\rangle, S\rangle \rightarrow \quad \langle G', last', C', \langle e_0\ \ldots\rangle, S\rangle}$$

FORK

$$\frac{\begin{array}{l}\langle G, last, C, \langle\langle t, frk, t'\rangle\ e_0\ \ldots\rangle\rangle \rightarrow_{CG} \langle G', last', C', \langle e_0\ \ldots\rangle\rangle \\ CheckRace(C(t), t, G, C, S) = false \\ S' = S \qquad S'(t)(t') = C'(t') \qquad S'(t')(t) = C'(t)\end{array}}{\langle G, last, C, \langle\langle t, frk, t'\rangle\ e_0\ \ldots\rangle, S\rangle \rightarrow \quad \langle G', last', C', \langle e_0\ \ldots\rangle, S'\rangle}$$

JOIN

$$\frac{\begin{array}{l}\langle G, last, C, \langle\langle t, jn, t'\rangle\ e_0\ \ldots\rangle\rangle \rightarrow_{CG} \langle G', last', C', \langle e_0\ \ldots\rangle\rangle \\ CheckRace(C(t), t, G, C, S) \vee CheckRace(C(t'), t, G, C, S) = false \\ S' = S \qquad S'(t)(t') = \bot\end{array}}{\langle G, last, C, \langle\langle t, jn, t'\rangle\ e_0\ \ldots\rangle, S\rangle \rightarrow \quad \langle G', last', C', \langle e_0\ \ldots\rangle, S'\rangle}$$

ACQUIRE

$$\frac{\begin{array}{l}\langle G, last, C, \langle\langle t, acq, m\rangle\ e_0\ \ldots\rangle\rangle \rightarrow_{CG} \langle G', last', C', \langle e_0\ \ldots\rangle\rangle \\ CheckRace(C(t), t, G, C, S) = false \qquad S' = S \qquad S'(t) = S(t)\ \sqcup\ S(tid(last))\end{array}}{\langle G, last, C, \langle\langle t, acq, m\rangle\ e_0\ \ldots\rangle, S\rangle \rightarrow \quad \langle G', last', C', \langle e_0\ \ldots\rangle, S'\rangle}$$

RELEASE

$$\frac{\begin{array}{l}\langle G, last, C, \langle\langle t, rel, m\rangle\ e_0\ \ldots\rangle\rangle \rightarrow_{CG} \langle G', last', C', \langle e_0\ \ldots\rangle\rangle \\ CheckRace(C(t), t, G, C, S) = false\end{array}}{\langle G, last, C, \langle\langle t, rel, m\rangle\ e_0\ \ldots\rangle, S\rangle \rightarrow \quad \langle G', last', C', \langle e_0\ \ldots\rangle, S\rangle}$$

Figure 3.1: Transition rules for ZipperOTF.

3.4 Example

The input to the ZipperOTF algorithm is a greedy-scheduled history; this can be obtained through reordering an observed history, or, as in the implementation, enforced. The RP partial order is preserved between histories, but is unused in the actual algorithm. The ZipperOTF algorithm then captures the HB relation by building a computation graph as described previously, checking for data race as it processes each event in the history. An example observed history and its corresponding greedy-scheduled history are shown here, along with the computation graph and data race checks that are performed by the ZipperOTF algorithm.

The observed history and a greedy-scheduled history are shown in Figure 3.2. The histories are shown in graph form, with time flowing from top to bottom, and each column representing a different thread as indicated by the event tuples in each node in the graph. Additionally the HB and RP orders are shown with red arrows and dashed blue edges respectively; solid edges indicate thread and fork/join edges for ease of viewing. In the observed history, events are shown in the order they occurred during execution, which is somewhat interleaved and switches context arbitrarily. The events to note are the two mutual exclusive blocks which are ordered by the HB relation, and events e and k, which serve to illustrate a write/read pair that must be preserved and is therefore ordered by the RP relation.

On the right of Figure 3.2, one of possibly many greedy-scheduled histories for the observed history is also shown. Both the HB order and the RP order are also shown in the greedy-scheduled history preserved as they were in the observed history. The figure shows a simple example, but in more complex case these orders are still preserved, choosing one thread over another in order to do so. Aside from preserving the RP order, the second property of the greedy-schedule is also apparent in the figure; the only time that a context switch occurs is before an *acq* or a *jn*. The resulting history has as many events grouped

together as possible, ordering as many events first on the current thread as possible before switching, following the properties defined in the greedy-scheduled history previously.

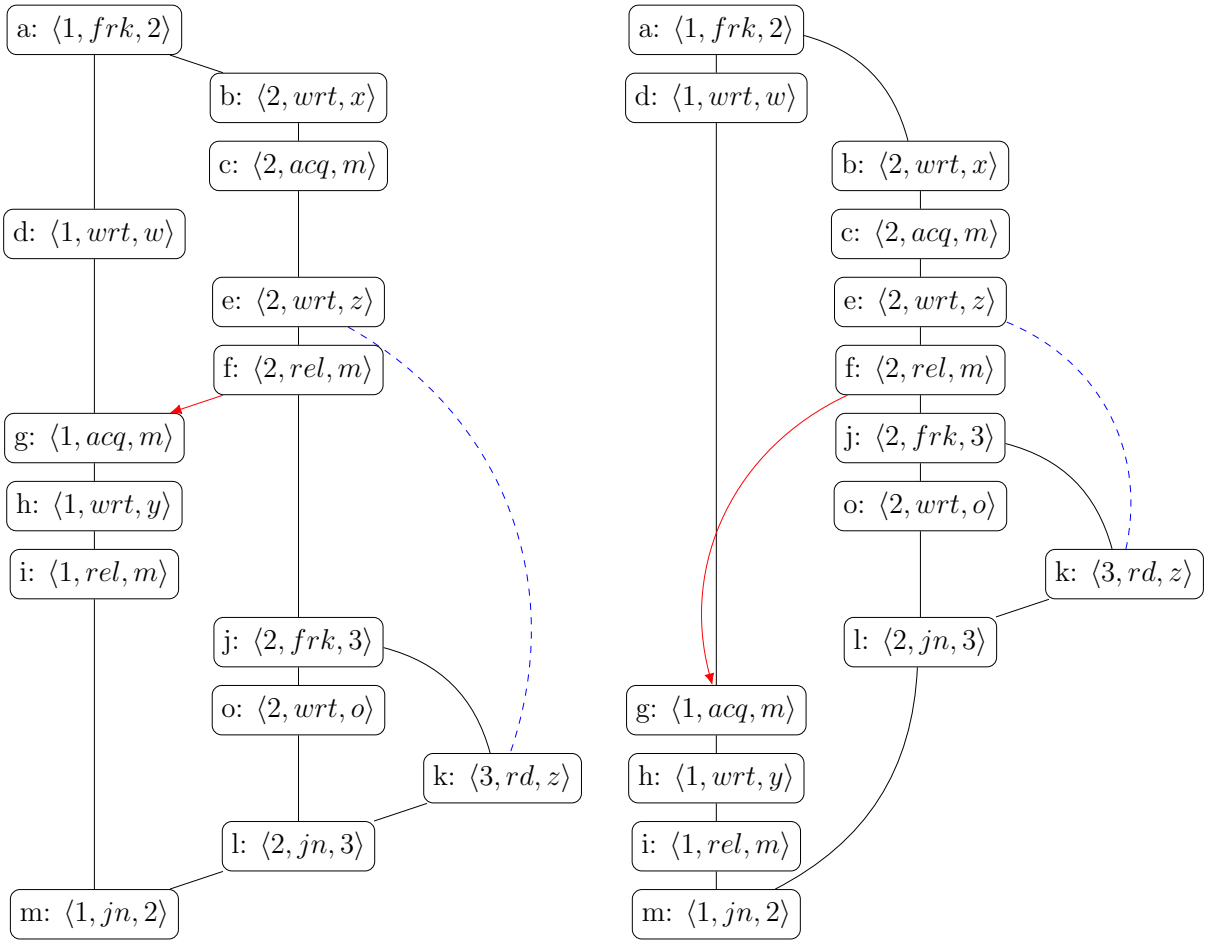

Figure 3.2: The observed history and a greedy-scheduled reordering of the observed history

From a greedy-scheduled history, the computation graph is built as shown previously in the ZipperOTF rewrite rules. The computation graph that is built from the greedy-scheduled history shown in Figure 3.2 is shown in Figure 3.3. The red arrow indicates an edge between mutual exclusive nodes, for ease in correlating it with the original history. Each node is given a number, and the events are also listed in the node to represent the access sets that are recorded by ZipperOTF.

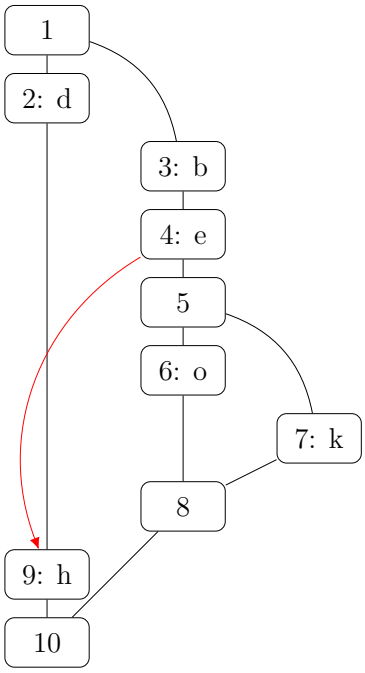

Figure 3.3: The Computation Graph built from the Greedy-Scheduled History

In order to illustrate each step that ZipperOTF takes as it processes the greedy-scheduled history, the partial graphs that exist during the algorithm are shown at key points where data race checks occur. Edges without nodes indicate an observed *acq*, *rel*, *frk*, or *jn* event where a data race check occurs. Dotted blue nodes indicate nodes that are being checked, and dashed green nodes indicate the source node that is checking for race. These checking nodes reference the sync map entries for each other thread in order to find which nodes to check.

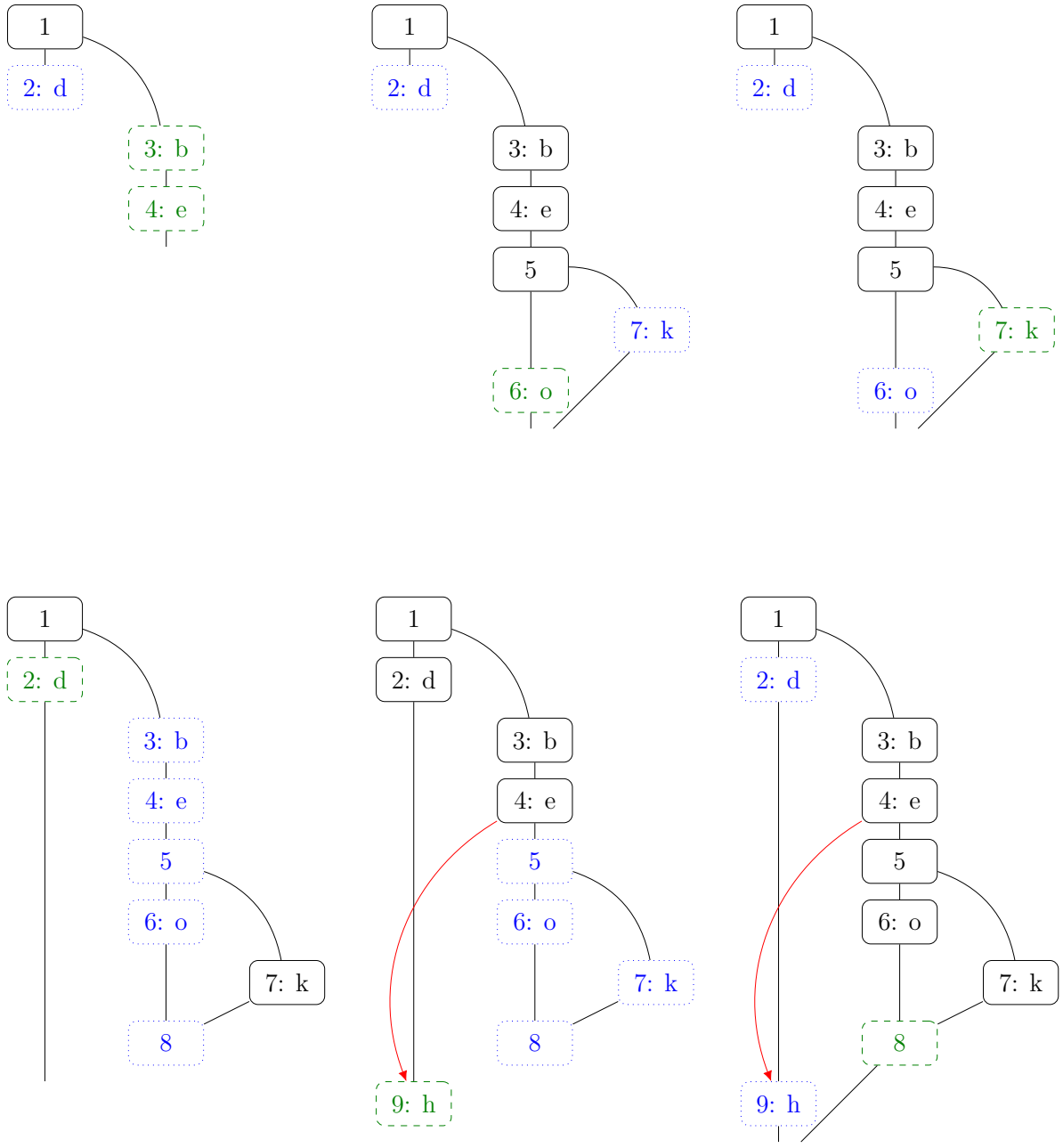

Figure 3.4: Each point where data race checks occur; green dashed lines denote the checking node and blue dotted lines denote the nodes being checked against

ZipperOTF can be performed during the observation of a history, if that history is guaranteed to be greedy-scheduled; but the examples shown here are post-mortem to illustrate how it detects data race. Given a greedy-scheduled history, ZipperOTF can build a graph that represents the HB order, track needed information in the form of sync maps, and

check for data race at synchronization points all on-the-fly while building the graph. As will be discussed in the next chapter, greedy-schedules are obtained through means of how the algorithm is implemented, and so the ZipperOTF analysis can be performed during execution with the guarantee that the history is greedy-scheduled.

Chapter 4

Implementation

The implementation of ZipperOTF utilizes the Java Path Finder (JPF) framework and the Habanero Java task parallel programming library. JPF is a suite of tools that allow for parallel programs to execute in a controlled JVM. Because the JVM is completely under programmer control, this makes it easy for a model checker to execute, backtrack, and state-match program executions in order to enumerate all possible executions. Additionally, this allows all executions to be greedy-scheduled histories, which allows ZipperOTF to process them. Utilizing the model checking capabilities of JPF allows for each interleaving of mutual exclusive blocks to be executed and analyzed by ZipperOTF to detect data race in each execution. By running ZipperOTF on each unique execution order, the absence of data race is proven if the execution reaches then end; this is because ZipperOTF reports a data race as soon as one is detected. Additional debugging tools are used as well to visualize and pinpoint exactly where a data race first occurs, and optimizations are also utilized in the tool to make it more user-friendly.

JPF provides specific hooks into each command that is executed by the Habanero Java library. These hooks correspond to the state transitions rules described previously, and the implementation follows the same pattern of state transformations. As each action is executed the corresponding hook into the ZipperOTF algorithm is called which constructs the graph and records the same information described in the transition rules. The graph is built using node objects with a set of variable accesses and references to nodes for edges. When a data race check is performed, instead of consulting a sync map matrix, the sync map

attached to the current node is indexed to find the nodes to be checked and each edge is followed until there is no next node, intersecting the access sets along the way.

Running the program once proves that there is no data race in that execution (if the execution terminates with no events left to process). In order to prove the absence of data race in the program (not just a single execution), each unique order of mutual exclusive blocks needs to be checked. JPF does this by executing as many actions as possible on the first thread until it reaches a mutual exclusion block where a context switch may occur; it then executes all other threads in the same manner in a predictable order. It then chooses one of the mutual exclusive blocks to execute, and continues to execute in the same manner until nothing is enabled again. Then, it backtracks to a previous state before an order was chosen and tries every combination; the ZipperOTF algorithm consequently has different instances and states for each unique execution.

This pattern of execution and backtracking to check each execution does two important things: first, it ensures completeness by proving the absence of data race in all orders of execution, and second, it enforces that each order is performed such that the history is a greedy-scheduled history. Because the order in which the mutual exclusion nodes are visited in the execution is always trivially the same order that they execute in, the need to process them in order of execution is no longer an obstacle as it is in the CG algorithm.

Additional tools are also included that allow detailed debugging and visualization. On command, the tool will output graph visualizations that show which variable and where in the execution the data race was detected. Different options can also be set to turn different optimizations and data on and off to make the tool more user-friendly. It also provides built in processes for reporting individual or collective benchmark times and running on different programs to compare different tools, as shown in the Results section. It is also possible to change when ZipperOTF halts execution when finding a data race, including the ability to stop just the current execution or the entire model checking analysis. Currently, since the objective is to report any data race or prove its absence, execution of the model checker is

stopped completely as soon as a data race is observed, and it is reported immediately. If no data race is present in the program, then execution continues until every enumeration is exhausted, and a message indicating no data race found is reported.

The implementation includes an optimization that is not present in the algorithm described previously. First, it doesn't always place an edge between mutual exclusive blocks; instead, it checks to see if they access the same variable, then place an edge if so. By not placing the edge, more graphs can be checked in a single execution without having to backtrack for a different order, and therefore the entire analysis requires less time (without sacrificing correctness). The partial order that the graph represents is subsequently not the HB order, but instead the WCP relation. The WCP has been shown to be sound and complete and allows for predicting a data race in another schedule, even if a data race is not observed in the current execution [24]. Reducing the model checking portion of the analysis yields a significant time improvement when using the tool.

Chapter 5

Results

5.1 Benchmarks and Experimentation

ZipperOTF focuses on both proving data race freedom with fewer operations as well as detecting data race earlier. Because it operates on-the-fly, as soon as a data race is observed, it is reported and the execution halts. This improvement in speed can be seen in many of the benchmarks that contain a data race as indicated by *true* in the "Race" column in Table 5.1. The execution time in milliseconds is given for three tools: FastTrack (denoted as FT in the table), the Computation Graph algorithm (CG), and ZipperOTF (ZOTF). There are an additional two columns that compare the algorithms to the FastTrack results; first the CG algorithm, then the ZipperOTF algorithm, as noted by the column headers. This metric is calculated using the formula given in the headers. FastTrack is typically seen to be the fastest, but, while both of the other algorithms are comparable in speed, ZipperOTF improves the time and space complexity of the naïve CG algorithm.

5.2 Complexity

The required reachability check in the CG analysis results in a time complexity of $O(N^3+N^2V)$, where N is the number of nodes in the graph and V is the number of shared variables. This large inefficiency is present because the CG algorithm does not take into account the additional synchronization information held in the history and graph that can provide more optimized approaches, and therefore it must accommodate with its inefficient reachability check.

While the space overhead is similar to that of vector clocks at $O(T^2 + NV)$, where T is the number of threads, this method of checking for data race becomes unfeasible for a large number of nodes and threads. ZipperOTF doesn't differ in space complexity from the CG analysis, but it does improve upon the time complexity by checking only concurrent nodes for data race. By intelligently tracking nodes that need to be checked, it reduces the time complexity to $O(N^2 + N^2V)$, cutting out the inefficient reachability check required by the CG analysis.

ZipperOTF uses the same computation graph built by the CG algorithm to represent the Happens-Before (HB) partial order. Sync maps are constructed while creating the computation graph during the traversal of the input history, and data race checks are made at mutual exclusion and join points in the graph. The only exception to this is ZipperOTF will not place an edge between two mutually exclusive nodes that do not contain conflicting accesses; this allows the algorithm to soundly predict data race in other executions of the program, even if those data races are not observed in the current history. As will be discussed previously, ensuring that the input histories are Greedy-Scheduled guarantees that the graph will be built and traversed in an order that allows the optimizations that ZipperOTF leverages to be used successfully without sacrificing correctness.

The time complexity of basic vector clock approaches is reported as $O(MT^2)$ where M is the number of shared variable accesses and T is the number of threads. FastTrack specifically can reach time complexities near $O(M)$ by utilizing its epoch based technique to limit the number of operations for a vector clock lookup. The space complexity of FastTrack is $O(T^2+VT)$, where V is the number of shared variables. FastTrack specifically is extremely fast in data race detection, but is a vector clock implementation, whereas ZipperOTF is an access set technique. TARDIS has shown that access sets can possibly exhibit higher performance under certain conditions; for example, in the absence of mutual exclusion TARDIS is able to be parallelized and minimize the number of set intersections drastically [22].

5.3 Performance Comparison

In many of the results, FastTrack outperforms ZipperOTF when it comes to empirical speed. In most cases this speed difference is fairly close, but can show a noticeable gap in speed. ZipperOTF, like CG, has the overhead of building a graph which requires more object creation than FastTrack. The construction of the graph enables ZipperOTF to accurately perform its data race detection, including grouping the accesses into the sets that are intersected.

While ZipperOTF does not outperform FastTrack in general, it does in most cases outperform its previous iterations, such as CG and Zipper. Because ZipperOTF is on-the-fly, it has a lower memory overhead than Zipper since it relies on the order that is currently being observed in the history. ZipperOTF reduces the complexity of the CG algorithm to $O(N^2 + N^2V)$ by intelligently identifying which nodes are concurrent and should be checked for data race. This space complexity is comparable to FastTrack since it only requires a single sync map per thread, whereas FastTrack must place a vector clock (similar to the sync map) on each shared variable. ZipperOTF also has many opportunities for improvement and optimization by utilizing techniques that are available from other task parallel tools, such as TARDIS. TARDIS provides techniques for parallelization that would allow ZipperOTF to reduce its complexity, but only in cases where mutual exclusion is absent. These techniques are either impossible or non-trivial to use with FastTrack since it operates on a wider class of parallel programs; therefore ZipperOTF can exploit the guarantees that task parallel programs provide and be optimized even further.

ZipperOTF's simplicity compared to Zipper is also an advantage that makes it easier to implement and use, since it requires much fewer data structures. It also ensures to only check between nodes that are concurrent and does not have the overhead of the reachability check that CG must perform post-mortem. FastTrack's worst case can be described as a large number of accesses to shared variables across a few threads; in this case ZipperOTF would coalesce these accesses and perform and intersection to see if there is a conflict, which theoretically is much more efficient.

ZipperOTF has potential to outperform FastTrack with additional optimizations that would utilize parallelization techniques from TARDIS, as well as incorporating epoch-based analyses like FastTrack. Because it is access-set-based, it would be able to perform well in certain cases of many reads and writes in a row on a thread of a shared variable, with lots of mutual exclusion. Reducing the object-creation overhead would also yield better empirical results, since the building of the graph can be intensive when the size of the program is large. Overall ZipperOTF shows in the empirical results that it can now compete with state of the art algorithms while simplifying the operations required to do so. It also has the opportunity to utilize weaker partial orders to yield more optimized proofs of data race absence in a program by reducing the number of graphs that must be explored in the context of a model checker. All experiments were run on an Intel(R) Xeon(R) Gold 5120 CPU with 8GB RAM, and the times are in milliseconds, except the far right two columns, which are ratios.

Benchmark	Race	FT	CG	ZOTF	CG/FT	ZOTF/FT
DataRaceIsolateSimple1	true	114	248	172	2.18	1.51
FuturesIso	true	127	283	314	2.23	2.47
PrimeNumCounter	false	666	702	186	1.05	0.28
DoubleBranchExample	false	125	218	223	1.74	1.78
JGFCryptBenchSizeC	false	5245	6267	6181	1.19	1.18
JGFCryptBenchSizeB	false	1980	1831	3075	0.92	1.55
JGFCryptBenchSizeA	false	6767	7105	9312	1.05	1.38
JGFSeriesBenchSizeC	false	855	891	1273	1.04	1.49
JGFSeriesBenchSizeB	false	2649	2146	2803	0.81	1.06
JGFSeriesBenchSizeA	false	350	292	448	0.83	1.28
strassen.Main	false	30985	31374	39859	1.01	1.29
DataRaceIsolateSimple	true	168	256	168	1.52	1.0
DisjointIsolated	false	21551	23596	29591	1.09	1.37
IsolatedLoop	false	163	226	234	1.39	1.44
ConfigurableSimpleNo	false	121	242	158	2.0	1.31
Template	false	108	230	174	2.13	1.61
Indirectaccess2OrigYes	true	1607	1593	1706	0.99	1.06
SimpleSimpleSimple	true	185	244	251	1.32	1.36
Antidep1VarYes	true	289	437	427	1.51	1.48
TrueDepLinearOrigYes	true	310	691	580	2.23	1.87
PrimitiveArrayRace	true	118	261	163	2.21	1.38
Antidep2OrigYes	true	225	483	469	2.15	2.08
FuncArgOrigYes	false	249	253	209	1.02	0.84
Indirectaccess4OrigYes	true	986	1653	1741	1.68	1.77
PrimitiveArrayNoRace	false	173	312	170	1.8	0.98
AddNo	false	175	340	326	1.94	1.86
Indirectaccess3OrigYes	true	1251	1566	1769	1.25	1.41
MinusMinusOrigYes	true	338	566	622	1.67	1.84
ConfigurableSimpleYes	true	114	197	173	1.73	1.52
TrueDepSingleElementOY	true	846	6476	6976	7.65	8.25
SimpleSimpleSimple2	true	175	241	186	1.38	1.06
ConfigurableManyAccessesNo	false	46444	42133	52866	0.91	1.14
DoAll1OrigNo	false	440	587	636	1.33	1.45
Antidep1OrigYes	true	1123	6353	9172	5.66	8.17
VectorAdd	false	173	209	181	1.21	1.05
TrueDep1VarYes	true	383	567	634	1.48	1.66
DoAll2OrigNo	false	2090	2097	2742	1.0	1.31
TrueDepFirstDimOY	true	1110	2989	3614	2.69	3.26
ForallWithIterable	false	179	219	192	1.22	1.07
Indirectaccess1OrigYes	true	1027	1665	1999	1.62	1.95
TrueDep1OrigYes	true	971	7961	6718	8.2	6.92
ReciprocalArraySum	false	2323	2358	2822	1.02	1.21
TwoDimArrays	false	173	381	337	2.2	1.95
MatrixMultiplyOrigNo	false	173113	175501	209188	1.01	1.21
LinearMissingOrigYes	true	352	624	688	1.77	1.95
LastPrivateMissingOrigYes	true	356	571	586	1.6	1.65
ThreeMMParallelNo	false	599	702	729	1.17	1.22
TrueDepLinearVarYes	true	377	530	533	1.41	1.41
Simd1OrigNo	false	492	653	822	1.33	1.67
ConfigurableManyThreadsNo	false	49383	43788	59872	0.89	1.21
Antidep2VarYes	true	945	6400	6936	6.77	7.34
ReciprocalArraySumFutures	false	2645	2492	2853	0.94	1.08
ScalarMultiply	false	196	313	315	1.6	1.61
ManyNetworkRequests	false	4780	5963	24251	1.25	5.07
TrueDepSingleElementVarYes	true	1111	6283	7425	5.66	6.68
IsolatedBlockNo	false	372	392	412	1.05	1.11

Table 5.1: Results

Chapter 6

Related Work

FastTrack [17] is a state of the art dynamic data race detection algorithm that uses vector clocks to encode the HB relation over the events in a program execution. FastTrack introduces optimizations that effectively compress the size of each vector clock and reduce the number of memory accesses that need to be checked for data race. However, the size of the entry is still proportional to the number of threads, which can grow large in task parallel programs. ThreadSanitizer [39] is a practical dynamic data race detection tool that utilizes many FastTrack optimizations and has been shown to scale to large industrial applications. However, ThreadSanitizer sacrifices soundness for performance and can miss data races.

Goldilocks is an example of a data race detection tool specifically for Java [12]. Tools such as Goldilocks focus instead on accessibility and usability, as it is written to be lightweight to specifically be used as a debugging tool. While instead of analyzing the program and identifying data races, it throws an error when a data race is about to occur, so that it can be handled or fixed. This solves a different problem than the dynamic data race detectors written here, and is specifically for debugging production code.

Feng *et. al.* [14] introduce the SP-Bags algorithm that uses a union find data structure to group tasks together that are logically concurrent with the executing task. Raman *et. al.* [38] extend SP-Bags to reason about async-finish parallelism. The algorithms presented in [43] and [42] also extend SP-Bags to analyze programs that use futures. Mellor-Crummey [32] and Utterback *et. al.* [44] use a thread labeling scheme to determine if two tasks are logically concurrent. Parallel data race detection algorithms have also been developed [44, 47]. All

of these algorithms give efficient representations of the HB relation that scale to programs with many threads but none of them fully support mutual exclusion without introducing the possibility of false data race reports.

The computation graph analysis by Nakade *et. al.* [34] supports mutual exclusion but uses an inefficient algorithm to determine when memory accesses are concurrent. The Zipper algorithm improves upon the computation graph analysis but is a solely post-mortem analysis making it impractical for many programs [40]. GT-Race, presented in [48] is another graph traversal based data race detection algorithm that supports mutual exclusion. However, GT-Race performs reachability queries on the graph after every access rather than coalescing accesses into sets and intersecting them periodically as is done in this work. In addition, some of the optimizations of GT-Race may be complementary to ZipperOTF.

Chapter 7

Conclusion

ZipperOTF represents a significant improvement upon its predecessor, Zipper, by combining the merits of the CG algorithm and vector clock techniques in order to bring task parallel program data race detection into the larger class of mutual exclusion. ZipperOTF simplifies the data structures required to ensure precise data race detection and proof of data race freedom while also expanding the capabilities of task parallel detectors to handle mutual exclusion as well as more types of programs that produce differently shaped computation graphs. It introduces a novel hybrid data structure using the advantages of both access set and vector clock techniques. In general it can perform empirically in a way that is comparable to other race detectors while reducing or matching the space required to precisely detect or prove freedom from data race. By executing on-the-fly, only the information that is required up to a certain point is kept, and any observed data race is promptly reported. Executing inside of a model checker will provide optimizations as well since the information gained from a single history can be used to influence the model checker to enumerate fewer graphs without losing precision. This is one of the future works that will be done with the ZipperOTF algorithm. It is also possible to utilize techniques from other algorithms such as TARDIS to parallelize and combine parts of the graph to greatly reduce the number of intersections required. This requires some work to be able to handle mutual exclusion while performing these optimizations, but provides a great platform to bring ZipperOTF to its full potential in detecting data race in task parallel programs.

References

[1] Koenraad Audenaert. Clock trees: Logical clocks for programs with nested parallelism. *IEEE Transactions on Software Engineering*, 23(10):646–658, 1997.

[2] George Balatsouras and Yannis Smaragdakis. Structure-sensitive points-to analysis for c and c++. In *International Static Analysis Symposium*, pages 84–104. Springer, 2016.

[3] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 133–144, New York, NY, USA, 2004. ACM.

[4] Ahmed Bouajjani and Michael Emmi. Analysis of recursively parallel programs. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 203–214, New York, NY, USA, 2012. ACM.

[5] Ahmed Bouajjani and Michael Emmi. Analysis of recursively parallel programs. *ACM SIGPLAN Notices*, 47(1):203–214, 2012.

[6] G. Brat and W. Visser. Combining static analysis and model checking for software analysis. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE Comput. Soc.

[7] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: The new adventures of old x10. 08 2011.

[8] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in cilk programs that use locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 298–309, New York, NY, USA, 1998. ACM.

[9] Mark Christiaens and Koen De Bosschere. Accordion clocks: Logical clocks for data race detection. In Rizos Sakellariou, John Gurd, Len Freeman, and John Keane, editors, *Euro-Par 2001 Parallel Processing*, pages 494–503, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[10] Jack B Dennis, Guang R Gao, and Vivek Sarkar. Determinacy and repeatability of parallel program schemata. In *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2012*, pages 1–9, IEEE, 2012. IEEE, IEEE.

[11] Dimitar Dimitrov, Veselin Raychev, Martin Vechev, and Eric Koskinen. Commutativity race detection. *SIGPLAN Notices*, 49(6):305–315, 2014.

[12] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A race and transaction-aware java runtime. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 245–255, New York, NY, USA, 2007. ACM.

[13] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[14] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 1–11, New York, NY, USA, 1997. ACM.

[15] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, PADD '88, pages 183–194, New York, NY, USA, 1988. ACM.

[16] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[17] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. *Commun. ACM*, 53(11):93–101, November 2010.

[18] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121, New York, NY, USA, 2005. ACM.

[19] M. Gligoric, P. C. Mehlitz, and D. Marinov. X10x: Model checking a new programming language with an "old" model checker. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 11–20, April 2012.

[20] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[21] Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. Thread-modular shape analysis. *SIGPLAN Notices*, 42(6):266–277, 2007.

[22] Weixing Ji, Li Lu, and Michael L Scott. Tardis: Task-level access race detection by intersecting sets. In *Workshop on Determinism and Correctness in Parallel Programming (WoDet), Houston, TX*, 2013.

[23] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 13–22, 2009.

[24] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. Dynamic race prediction in linear time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 157–170, New York, NY, USA, 2017. ACM.

[25] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. Dynamic race prediction in linear time. In *ACM SIGPLAN Notices*, volume 52, pages 157–170. ACM, 2017.

[26] Sergey Kulikov, Nastaran Shafiei, Franck Van Breugel, and Willem Visser. Detecting data races with java pathfinder, 2010.

[27] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[28] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[29] Li Lu, Weixing Ji, and Michael L. Scott. Dynamic enforcement of determinism in a parallel scripting language. *SIGPLAN Not.*, 49(6):519–529, June 2014.

[30] Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko. Precise thread-modular verification. In *Static Analysis*, pages 218–232. 2007.

[31] Friedemann Mattern et al. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

[32] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 24–33. IEEE, 1991.

[33] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 24–33, New York, NY, USA, 1991. ACM.

[34] Radha Nakade, Eric Mercer, Peter Aldous, and Jay McCarthy. Model checking task parallel programs for data-race. In *NASA Formal Methods*. Springer International Publishing, 2018.

[35] Radha Nakade, Eric Mercer, Peter Aldous, and Jay McCarthy. Model-checking task parallel programs for data-race. In Aaron Dutle, César Muñoz, and Anthony Narkawicz, editors, *NASA Formal Methods*, pages 367–382, Cham, 2018. Springer International Publishing.

[36] Peter; Ogles, Benjamin; Aldous and Eric. Mercer. Proving data race freedom in task parallel programs with a weaker partial order. *FMCAD*, 2019.

[37] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification*, pages 368–383, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[38] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. In *International Conference on Runtime Verification*, pages 368–383. Springer, 2010.

[39] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic race detection with llvm compiler. In *International Conference on Runtime Verification*, pages 110–114. Springer, 2011.

[40] Kyle Storey, S. Jacob Powell, Benjamin Ogles, Joshua Hooker, Peter Aldous, and Eric Mercer. Optimized sound and complete data race detection in structured parallel programs. Salt Lake City, UT, USA, 2018.

[41] Rishi Surendran and Vivek Sarkar. Dynamic determinacy race detection for task parallelism with futures. In Yliès Falcone and César Sánchez, editors, *Runtime Verification*, pages 368–385, Cham, 2016. Springer International Publishing.

[42] Rishi Surendran and Vivek Sarkar. Dynamic determinacy race detection for task parallelism with futures. In *International Conference on Runtime Verification*, pages 368–385. Springer, 2016.

[43] Robert Utterback, Kunal Agrawal, Jeremy Fineman, I Lee, and Ting Angelina. Efficient race detection with futures. *arXiv preprint arXiv:1901.00622*, 2019.

[44] Robert Utterback, Kunal Agrawal, Jeremy T Fineman, I Lee, et al. Provably good and practically efficient parallel race detection for fork-join programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 83–94. ACM, 2016.

[45] Martin Vechev, Eran Yahav, Raghavan Raman, and Vivek Sarkar. Automatic verification of determinism for structured parallel programs. In *Proceedings of the 17th International Conference on Static Analysis*, pages 455–471, 2010.

[46] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 205–214, New York, NY, USA, 2007. ACM.

[47] Adarsh Yoga, Santosh Nagarakatte, and Aarti Gupta. Parallel data race detection for task parallel programs with locks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 833–845. ACM, 2016.

[48] Lechen Yu and Vivek Sarkar. Gt-race: graph traversal based data race detection for asynchronous many-task parallelism. In *European Conference on Parallel Processing*, pages 59–73. Springer, 2018.

[49] Timothy K. Zirkel, Stephen F. Siegel, and Timothy McClory. Automated verification of chapel programs using model checking and symbolic execution. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods*, pages 198–212, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.