RICE UNIVERSITY

# Data Race Detection for Event-Driven Parallel Runtime Systems

Thesis by
**Lechen Yu**

RICE
Thesis for the Degree of Master of Science
Department of Computer Science
Rice University (Houston, Texas) July, 2017

RICE UNIVERSITY

# Data Race Detection for Event-Driven Parallel Runtime Systems

by

**Lechen Yu**

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

**Master of Science**

Approved, Thesis Committee:

Vivek Sarkar

Vivek Sarkar, Chair
Professor of Computer Science
E.D. Butcher Chair in Engineering

Keith D Cooper

Keith D. Cooper
L. John and Ann H. Doerr Chair in
Computational Engineering
Professor of Computer Science

Robert Cartwright

Robert (Corky) Cartwright
Professor of Computer Science

Houston, Texas

July, 2017

ABSTRACT

Data Race Detection for Event-Driven Parallel Runtime Systems

by

Lechen Yu

Event-Driven Parallel (EDP) runtime systems (or more simply, EDP runtimes) are growing in popularity in the high-performance computing area because they provide a promising foundation for new programming systems that can support heterogeneous architectures and ever-increasing hardware complexity. EDP runtimes allow the programmer to focus on program logic, such as control and data dependences, thereby enabling portability across a wide range of platforms and system configurations. However, the applications written on top of EDP runtimes remain vulnerable to data races. Existing data race detection tools either do not support the primitives in EDP runtimes, or incur intractable large overheads by failing to utilize the structural information available in event-driven programs. In this dissertation, we propose a graph-traversal based data race detection method for EDP runtimes. It introduces a reachability graph (encodes the dependences in a program), to check the happens-before relation between memory accesses. In order to reduce the time complexity for race detection, we propose a few optimizations, such as *reachability cache* and *reversed reachability graph* to avoid unnecessary graph traversals and *path compression* to reduce the number of steps performed for graph traversal. Based on our race detection technique, we have developed a prototype implementation for the Open Community Runtime (OCR). Our evaluation on a set of open source OCR

benchmarks shows that our tool handles all OCR constructs, and that the time overhead for race detection is comparable to that of past work on race detection for more constrained (e.g., fork-join) runimes.

# Acknowledgments

I would first like to thank my thesis advisor, Prof. Vivek Sarkar, for providing me with precious advice on my research. He also spent substantial time revising my poor writing. Thank you very much for your guidance.

I would also like to thank the rest of my thesis committee: Prof. Keith Cooper and Prof. Robert Cartwright. Thank you for your comments, feedback and challenging questions. I also enjoyed their courses: Introduction to Compiler(Comp 412) and Programming Language(Comp 511), which showed me promising directions in the programming language area.

I would also like to acknowledge Arghya Chatterjee, for his help on this thesis and my daily life. He helped me adapt to this country and patiently taught me all required skills.

My sincere thanks goes to my best friends, Dingming and Betty. Thank you for being by my side during my highs and lows.

Finally, I would like to express my very profound gratitude to my parents for providing me with unwavering support and continuous encouragement throughout my years of education. Thank you for letting me chase my dreams in a foreign country without reservation. This accomplishment would not have been possible without their love and affection.

# Contents

# Illustrations

# Chapter 1

# Introduction

## 1.1 Motivation

With the ever-increasing complexity of modern computing architectures (e.g., heterogeneous processing units and hierarchical memories), applications on these machines must leverage the architectural complexity to perform efficiently. While developing parallel applications, the program must repeatedly be tuned to obtain the proper work partition for load balancing, thereby reducing the application execution time on a given platform. In order to avoid low-level tuning, Task Parallel (TP) runtimes have been proposed, which contains a group of concise constructs and a well-designed runtime. Programmers only need to divide the program logic into tasks and specify the dependences among tasks, and the underlying runtime will be responsible for task creation, task scheduling, memory allocation and data migration after launching the program. In general, the runtime has a complete knowledge of the program and machine so it can schedule task and data dynamically according to the machine status. Usually, TP runtimes can achieve better performance than manually tuning [1, 2]. Among TP runtimes, Event-Driven Parallel (EDP) runtimes [1] are a new trend. They support a graph model that allows the expression of dependences more naturally. Compared with other TP runtimes, EDP runtimes are much more general since they can express computation graphs that are more general than those supported by fork-join TP runtimes.

Although EDP runtimes alleviate the difficulty of writing efficient and portable parallel programs, event-driven applications are still prone to data races, a notorious error in parallel programs. A data race occurs when the program issues two unordered memory accesses to the same location where at least one of the accesses is a write. Since the order of memory accesses can be reversed in some executions compared with others, the program behavior is dependent on the thread interleaving. Since a data race may only occur on some particular interleavings, detecting and reproducing data races can be hard and time-consuming. It may take multiple weeks or months to fix a data race in certain cases [3].

There has been a lot of past work on detecting data race automatically at runtime. Some algorithms are very general [4, 3], but they do not take the feature of EDP runtimes into consideration, which causes additional overhead because each task has to be treated as a separate thread to apply these approaches on event-driven programs. Other works are only applicable to a specific type of parallel runtimes. For instance, SP-bag [5] can only detect data race for spawn-sync parallel runtimes and *fully strict* computation graphs, and ESP-bag [6] can only detect data race for async-finish parallel runtimes and *terminally strict* computation graphs. These algorithms make use of the structural information to report data race precisely with low overhead, but they rely on runtime-specific constraints on the computation graph structures, which are not satisfied in EDP runtimes. Currently, there does not exist any data race detection algorithm with tractable overhead that can support EDP runtimes.

In this dissertation, we introduce a *reachability graph* for EDP runtimes, a novel representation of the happens-before relation in an EDP program*. The reachability graph expresses the happens-before relation as directed paths, taking into account

*A program utilizing EDP runtimes to achieve parallelism

the unique dependence properties of EDP runtimes. It enables the use of a graph traversal based data race detection algorithm. After one execution, the race detection algorithm can detect data races in all possible thread interleavings for the same input.

## 1.2 Thesis Statement

*The reachability graph can represent the happens-before relation in an EDP program, and can be used to enable precise dynamic data race detection algorithms that can detect data races in an EDP program with less overhead than existing techniques.*

## 1.3 Contributions

This thesis makes the following contributions:

- *Reachability graph*, a graph based representation of the happens-before relation in an EDP program.
- A graph traversal based data race detection algorithm for EDP runtimes.
- Three optimizations to the race detection algorithm, which reduce the time overhead without loss in precision.
- A data race detection tool that supports all the constructs in the Open Community Runtime, an exemplar of EDP runtimes that is the focus of the implementation work in this dissertation.
- Experimental performance evaluations of the race detection algorithm and the three optimizations.

## 1.4 Organization

This thesis is organized as follows:

- Chapter 2 contains background on EDP runtime and data race detection. This chapter introduces a classification on parallel runtimes and gives a comprehensive description of the Open Community Runtime as an example of EDP runtimes. For data race detection, this chapter introduces vector clock [7], a widely used abstraction of data race detection and illustrates how to apply vector clocks to an EDP program.

- Chapter 3 discusses our graph traversal based data race detection algorithm. This chapter focuses on the reachability graph and the race detection algorithm against the graph.

- Chapter 4 discusses our data race detection tool. This chapter shows how we implement the proposed race detection algorithm on top of Intel Pin [8] and how we extract the runtime information through binary instrumentation.

- Chapter 5 discusses the intuitions to optimize the race detection algorithm. This chapter introduces three different optimizations.

- Chapter 6 evaluates our race detection algorithm on a group of Open Community Runtime benchmarks. This chapter compares the performance between the original race detection algorithm and the three optimizations.

- Chapter 7 discusses related work for data race detection.

- Chapter 8 wraps up by summarizing the thesis and potential areas for future research.

# Chapter 2

# Background

## 2.1 Task Parallel Runtimes

A Task Parallel (TP) runtime treats tasks as first-class citizens. When a parallel program is mapped on to a TP runtime, it is decomposed into a group of tasks that are independent by default, so that any dependences must be specified explicitly [9]. A task is a dynamic instance of a code segment that executes asynchronously, and is the basic execution unit of a TP runtime. A single task may have control or data dependences with other tasks. For instance, one task may send its result to another task as an input. A task cannot start execution until all tasks that it depends on have completed. Compared to a system-level execution unit, such as a thread, task is both more general and high-level constructs. TP runtimes hide low-level details of execution, such as context switch and processor affinity, from the higher levels of the application. The dependences in a program can be represented by a computation graph, in which tasks are represented by nodes and dependences by edges.

Figure 2.1 illustrates how a TP runtime tackles a parallel program. The program is represented as a dynamic computation graph. The TP runtime consists of a task scheduler (which may have a centralized or distributed implementation), and multiple processing units which can execute one task at a time. In many TP runtime implementations (including work-stealing runtimes), each processing unit has a task queue to store runnable tasks. The task scheduler is responsible for managing

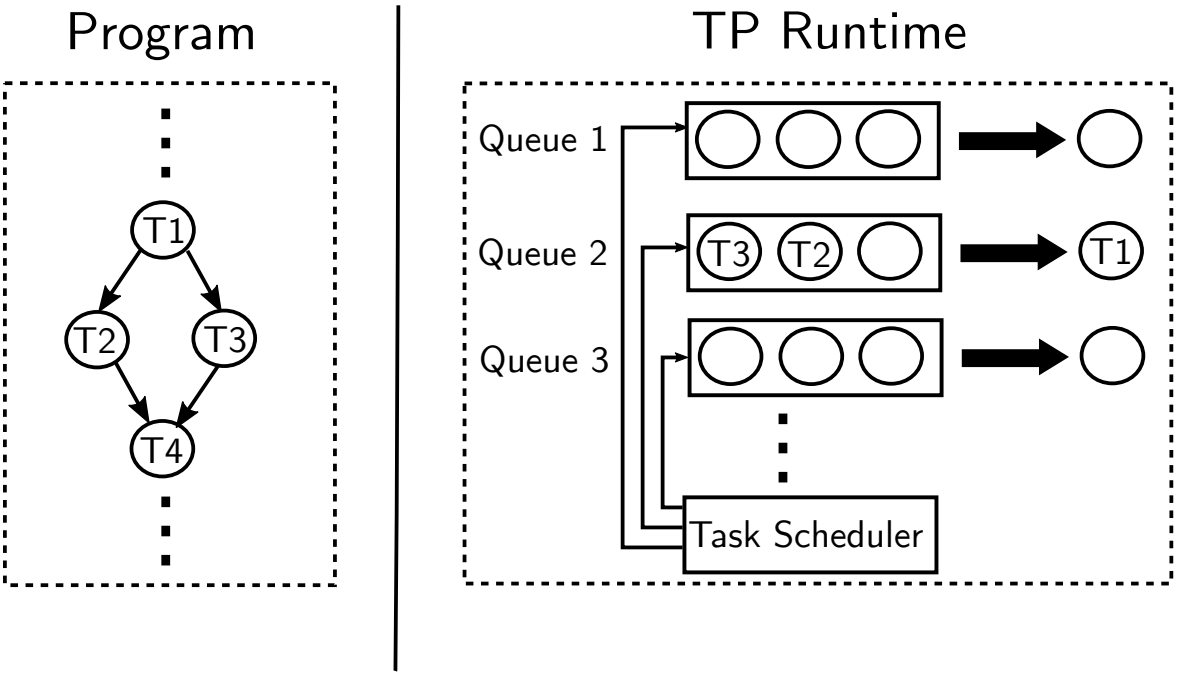


Figure 2.1 : Task Parallel Program and Runtime

the creation and scheduling of task, including keeping track of the dependences to determine when tasks become runnable (ready). When a processing unit becomes available, the decision of which runnable task it should execute next is determined by the task scheduler's policy. Thus, the TP runtime provides a high-level abstraction of the underlying machine and hides low-level features from the application. By only exposing high-level abstractions, such as tasks, to the application, TP runtimes make programs more portable to different machines. Furthermore, TP runtimes free programmers from the low-level burden of performance tuning for load balance; instead, programmers only need to specify tasks and their dependences, and can leave the scheduling details to the TP runtime.

To help structure our discussion of background related to EDP runtimes, we summarize a categorization of parallel runtimes in Table 2.1. Two widely used parallel

runtimes, MPI and Pthreads, are categorized as System-level Parallel (SP) runtimes as their constructs only support system-level parallel programming. Programmers must manually create threads and bind logical tasks to the threads. Further programmers are also responsible for inserting necessary synchronization operations, such as mutexes and barriers, to ensure the correctness of thread execution and shared memory accesses.

TP runtimes are divided into three subclasses according to how dependences are specified in each subclass:

- **Spawn-Sync Parallel Runtimes.** In spawn-sync parallel runtimes, such as Cilk, a task can only wait for its immediate child tasks. Cilk includes two constructs, *cilk_spawn* and *cilk_sync*, to write programs that use spawn_sync task parallelism. *cilk_spawn* specifies that a function call is treated as a child task and executes asynchronously. *cilk_sync* specifies that all spawned child tasks must complete before the parent task can continue.

- **Async-Finish Parallel Runtimes.** Async-finish parallel runtimes relax the "only wait for immediate child" restriction by supporting *finish* as a more general task termination construct. In X10 and HJ, *async* is equivalent to *cilk_spawn* that specifies an asynchronous child task. *finish* specifies a scope in which the invoking task must wait until all directly and transitively spawned tasks within the scope complete. *finish* enables a task to depend on a set of nested parallel tasks. TBB, OpenMP Task and HPX define a similar construct, *task_group*. A task can synchronize with a *task_group* to wait for the termination of all tasks that belong to the *task_group*.

- **Event-Driven Parallel Runtimes.** Unlike spawn-sync and async-finish task

| | | |
|---|---|---|
| SP Runtimes | | MPI [10], Pthreads [11] |
| TP Runtimes | spawn-sync | Cilk [12], |
| | async-finish | X10 [13], HJ [14], TBB [15], OpenMP Task [16], HPX [17] |
| | event-driven | CnC [18], OCR [1], Realm [19], StarPU [20], HC [21] |

Table 2.1 : Parallel Runtime Categorization

parallel runtimes, in which programmers must wrap tasks into a scope to synchronize with other tasks, Event Driven Parallel (EDP) runtimes allow direct specification of dependences. EDP runtimes use graph-based models to specify dependences. For instance, in Realm, *spawn* accepts a task waiting list when spawning a new task. Another example is OCR, which provides a function *ocrAddDependence* to specify the dependence between any two tasks.

## 2.2 Open Community Runtime

In this dissertation, we focus on Open Community Runtime (OCR), an open-source EDP runtime developed by Intel, Rice, and others. The basic four objects in OCR are event driven tasks (EDTs), data blocks (DBs), events and EDT templates, all of which are referred to as OCR objects. Each OCR object has a globally unique ID (GUID) used to identify the object during its life cycle. Further, OCR objects may have dependences on other objects. The definitions of the four OCR objects are listed below.

- **EDT** represents a task. The code snippet associated with an EDT will execute asynchronously after all its dependences are satisfied. The process of EDT exe-

cution is nonblocking, which implies no synchronization operation is permitted within an EDT. For convenience, there is a special EDT called "**finish EDT**" that mimics the semantics of *finish* from async-finish parallel runtimes. A finish EDT will terminate after all directly and transitively spawned EDTs in its scope terminate.

- **DB** represents a chunk of consecutive memory that an EDT can access. An EDT can only access DBs that are specified as data dependences or created during its execution. At the start of execution, each EDT internally records the start pointers for accessible DBs. The start pointer is only guaranteed to be valid during the execution of the acquiring EDT because OCR may migrate DBs to other locations or duplicate a DB to transparently make two copies for two acquiring EDTs.

- **Event** represents synchronization among EDTs. The semantics is similar to that of a semaphore or latch. An event may have multiple directed edges linked to multiple EDTs. EDTs linking to an event through its outgoing edges must wait for termination of EDTs linking through incoming edges.

- **EDT template** represents meta-data from which an EDT is created. It records the code to execute and the number of dependences. As with objects and classes in object-oriented programming, multiple EDTs can be created from the same EDT template.

Control and data dependence are mapped to different kinds of edges in an OCR program.

- **Control Dependence.** A directed edge from an event to an EDT that does not involve a DB.

- **Data Dependence.** A directed edge from a DB to an EDT or event.

As an EDP runtime, OCR provides a graph-based model to spawn OCR objects and specify dependences. The key data structures and functions are listed below (Only selected parameters are listed, for more detail, please refer to the OCR specification [1]):

- ***ocrGuid_t.*** Type of GUID used to reference all OCR objects.

- ***ocrEdtDep_t.*** Type of dependence. It has two fields, *guid* referring to a DB and *ptr* pointing to the DB's start address. These fields are only valid when the dependence is a data dependence.

- ***ocrEdtCreate(edt_guid, template_guid, output_guid).*** Function used to create an EDT. *edt_guid* is an output parameter for the spawned EDT's GUID. *template_id* refers to the associated EDT template. *output_guid* is an output parameter for the associated output event that indicates its termination.

- ***ocrDbCreate(db_guid, addr, size).*** Function used to create a DB. The first two parameters are output parameters, *db_guid* for the spawned DB's GUID and *addr* for the start address. The third parameter, *size*, is an input parameter that specifies the size of memory in bytes.

- ***ocrEventCreate(event_guid).*** Function used to create an event. It only tasks one output parameter *event_guid* for the spawned event's GUID.

- ***ocrEdtTemplateCreate(template_guid, func_ptr).*** Function used to create an EDT template. The first parameter *template_guid* is an output parameter for the spawned EDT template's GUID. The second and third parameter are

input parameters, *func_ptr* pointing to the function that EDT will execute and *depc* indicating the number of dependences.

- ***ocrAddDependence(src, dst).*** Function used to specify a dependence between two EDTs. It tasks two input parameters, *src* referring to the source OCR object and *dst* referring to the destination OCR object.

- ***ocrShutdown()*:** Function used to terminate OCR program execution.

Listing 2.1 shows an OCR implementation of a parallel array sum computation. Note that this code is very verbose, since the OCR APIs are designed to represent an intermediate runtime interface, rather than a programming interface. We would expect a higher-level programming model to automatically generate code such as that in Listing 2.1. The program first divides the workload evenly into two EDTs which calculate the partial sum asynchronously. Then the two partial sums are transferred to the third EDT to calculate the total sum. Lines 1-16 define a function responsible for calculating a partial sum. It acquires two DBs as data dependences that record the assigned array and its size. After sequentially adding up the elements in its subarray, it outputs the DB storing the partial sum. Lines 17-25 define a function responsible for outputting the array sum. It adds up the two partial sums from the preceding two EDTs and prints the final result. Lines 26-54 define *mainEdt*, which is the entry point of an OCR program. It sets up the parallel array sum through OCR constructs. Lines 46-52 specify dependences among OCR objects.

The corresponding dynamic computation graph is shown in Figure 2.2 in which an EDT, DB, or event are denoted by an ellipse, rectangular, or rhombus. We observe that the function calls in *mainEdt* specify key dependences for this program.

```
ocrGuid_t partial_sum(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    // get array size
    u32 len = * (int*) depv[0].ptr;
    // get array
    int* array = (int*) depv[1].ptr;
    // allocate variable for partial sum
    int* k
    ocrGuid_t db_guid;
    ocrDbCreate(&db_guid, (void**) &k, sizeof(int));
    k[0] = 0;
    for (u32 i = 0; i < len; i++) {
        k[0] += array[i];
    }
    // return partial sum
    return db_guid;
}
ocrGuid_t output(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    // get partial sum
    int* data1 = (int*) depv[0].ptr;
    int* data2 = (int*) depv[1].ptr;
    // output array sum
    printf("Array sum is %d\n", * data1 + * data2);
    ocrShutdown(); // shutdown the program
    return NULL_GUID;
}
ocrGuid_t mainEdt ( u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    u32 len = paramv[0];
    int* array = malloc(sizeof(int) * len);
    // define GUID
    ocrGuid_t partial_sum_template, output_template;
    ocrGuid_t edt1, edt2, edt3, sync_event, db1, db2, db3;
    // define EDT template
    ocrEdtTemplateCreate(&partial_sum_template, partial_sum, 1);
    ocrEdtTemplateCreate(&output_template, output, 2);
    // define EDT
    ocrEdtCreate(&edt1, partial_sum_template, NULL);
    ocrEdtCreate(&edt2, partial_sum_template, NULL);
    ocrEdtCreate(&edt3, output_template, NULL);
```

```
    // define event
    ocrEventCreate(&sync_event);
    // define DB
    ocrDbCreate(&db1, (void**) array, sizeof(int) * len / 2);
    ocrDbCreate(&db2, (void**) array + len / 2, sizeof(int) * len / 2);
    ocrDbCreate(&db3, (void**) len, sizeof(u32));
    // specify dependence
    ocrAddDependence(db1, edt1);
    ocrAddDependence(db3, edt1);
    ocrAddDependence(db2, edt2);
    ocrAddDependence(db3, edt2);
    ocrAddDependence(edt1, sync_event);
    ocrAddDependence(edt2, sync_event);
    ocrAddDependence(sync_event, edt3);
    return NULL_GUID;
}
```

Listing 2.1: OCR Implementation of Array Sum

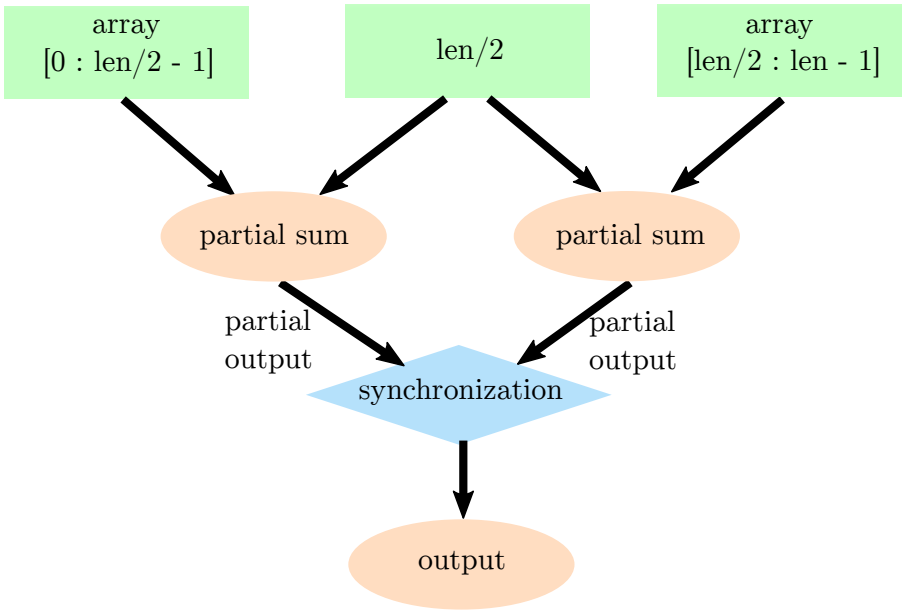


Figure 2.2 : Dynamic Computation Graph

## 2.3 Data Race Detection

A data race occurs when two memory operations access the same memory location without any ordering constraints and at least one of them is a write operation. Because the two operations are not ordered, the final value in the accessed memory location is nondeterministic. Data race causes the program to generate nondeterministic results, thereby making data races hard to detect, reproduce and fix. To alleviate the difficulty of detecting data races, a wide range of approaches for automatic detection of data races have been proposed in past work. Some methods utilize static analysis to detect suspicious memory accesses [22]. While many static approaches can guarantee soundness, they are usually prone to large numbers of false positives. In contrast, dynamic approaches to data race detection instrument the program to record all memory accesses to shared variables at runtime, and check the happens-before relations among these operations dynamically [4, 23]. Compared with static data race detection, dynamic data race detection is more precise, but its scope is limited to a single input. Further, it also incurs higher overhead due to memory instrumentation and other runtime book-keeping. In this thesis, we focus on dynamic data race detection.

## 2.4 Vector Clocks

Vector clocks were proposed by Leslie Lamport and Friedemann Mattern to solve event ordering problem in distributed system [24, 7]. It models each process in the system as an event sequence which communicates with other processes through messaging passing. The happens-before relation $\rightarrow$ between two events is defined by the following three conditions.

- If events $a$ and $b$ are in the same process and $a$ occurs before $b$, then $a \rightarrow b$.
- If event $a$ represents sending a message and event $b$ is the associated receipt of the same message, then $a \rightarrow b$.
- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

With the abovementioned three rules, if there is an event sequence $t_i$ from event $a$ to event $b$ and any two adjacent events $t_i$, $t_{i+1}$ have $t_i \rightarrow t_{i+1}$, then we can conclude that $a \rightarrow b$, otherwise $a$ and $b$ may happen in parallel. Figure 2.3 shows an example of happens-before relation. *e2* happens before *e8* as there exists an event sequence $\{$*e2*, *e4*, *e5*, *e8*$\}$.

A vector clock $VC : ProcessID \rightarrow Nat$ records a clock for each process in the system. Suppose there are $n$ processes in the system, each process will acquire a $n$-length vector $vc$ in which the $i-th$ element represents the latest preceding epoch of process $i$. There are several operators defined on vector clocks.

$$vc_1 \leq vc_2 \quad = \quad \text{if } \forall i.\, vc_1[i] \leq vc_2[i] \text{ then } true \text{ else } false.$$

$$vc_1 \bigcup vc_2 \quad = \quad \forall i.\, max(vc_1[i], vc_2[i]).$$

$$inc_j(vc) \quad = \quad \forall i.\ \text{if } i == j \text{ then } vc[i] = vc[i] + 1.$$

$$\perp (vc) \quad = \quad \forall i.\, vc[i] = 0.$$

Each vector clock updates locally according to local events and received messages, without acquiring a consistent view of global state among all processes. The rules of updating a vector clock are listed below.

- **Initialize process *i*.** $vc = \perp (vc)$, then $inc_i(vc)$.

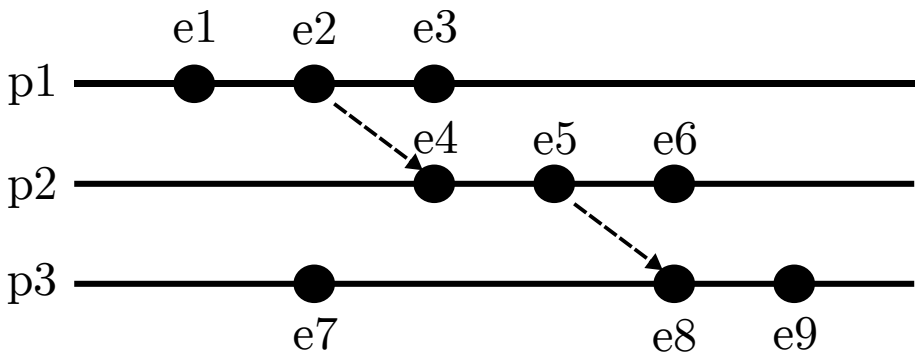


Figure 2.3 : Example of Event Ordering

- **Send a message to process *i*.** Send out $vc$ through a message, then $inc_i(vc)$.
- **Receive a message from process *i*.** Extract $vc'$ from the received message, then $vc = vc \bigcup vc'$.

When an event occurs, its clock is set to the vector clock of the affiliated process. By comparing assigned clock of two events locally, we can tell whether one happens before the other or they run in parallel. For any two events $a$ and $b$, $a$ happens before $b$ iff $vc_a \leq vc_b$.

Vector clocks provide an approach to encoding happens-before relations in a point-wise manner. It can detect casual violations in any system whose order relation is isomorphic to the happens-before relation. For an OCR program, if all operations in an EDT are mapped to a "process" and inter-EDT operations, such as spawning EDTs and dependences, are mapped to "message passing", then vector clocks can be applied to verify whether all accesses to shared memory are ordered correctly.

First, we give formal definitions of possible operations in an OCR program.

- ***read(x)*:** read a value from variable $x$.
- ***write(x)*:** write a value to variable $x$.

- ***spawn(t, u)*:** EDT $t$ spawns EDT $u$.
- ***add_dependence(t, u)*:** EDT $u$ depends on EDT $t$.

The transformed rules of updating vector clocks for operations are listed below. $vc\_final_t$ denotes the final vector clock after EDT $t$ terminating.

- ***spawn(t, u)*:** $vc_u = vc_t, vc_t = inc_t(vc_t)$.
- ***add_dependence(t, u)*:** $vc_u = vc_u \bigcup vc\_final_t$.

Listing 2.2 shows an OCR program containing a data race, and Figure 2.4 describes the transition of vector clocks during the execution of that program. *mainEdt* spawns *edt1* at $\langle 1,0,0\rangle$ and *edt2* at $\langle 2,0,0\rangle$. All of three EDTs write to the same variable $x$. The write in *mainEdt* happens before the write in child EDTs as $\langle 1,0,0\rangle \leq \langle 1,1,0\rangle$ and $\langle 1,0,0\rangle \leq \langle 2,0,1\rangle$. However, $\langle 1,1,0\rangle \nleq \langle 2,0,1\rangle$, so there exists a data race between the write by *edt1* and *edt2*. The execution order of EDTs has no effect on race detection as vector clocks can detect potential data races in all possible interleavings for a given input. Even if in one execution the three EDTs execute sequentially (on a single processor, say), and the actual data race does not occur, the vector clock approach can still report data races after comparing the clocks of different operations.

```
1 ocrGuid_t child(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
2     int* data = (int*) depv[0].ptr;
3     *data = 1;
4     return NULL_GUID;
5 }
6 ocrGuid_t mainEdt ( u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
7     int* data;
8     ocrGuid_t edt1, edt2, db, template;
9     ocrEdtTemplateCreate(&template, child, 1);
```

```
10      ocrDbCreate(&db, (void**)data, sizeof(int));
11      *data = 0;
12      ocrEdtCreate(&edt1, template, NULL);
13      ocrEdtCreate(&edt2, template, NULL);
14      ocrAddDependence(db, edt1);
15      ocrAddDependence(db, edt2);
16      return NULL_GUID;
17  }
```

Listing 2.2: OCR Program with Data Race

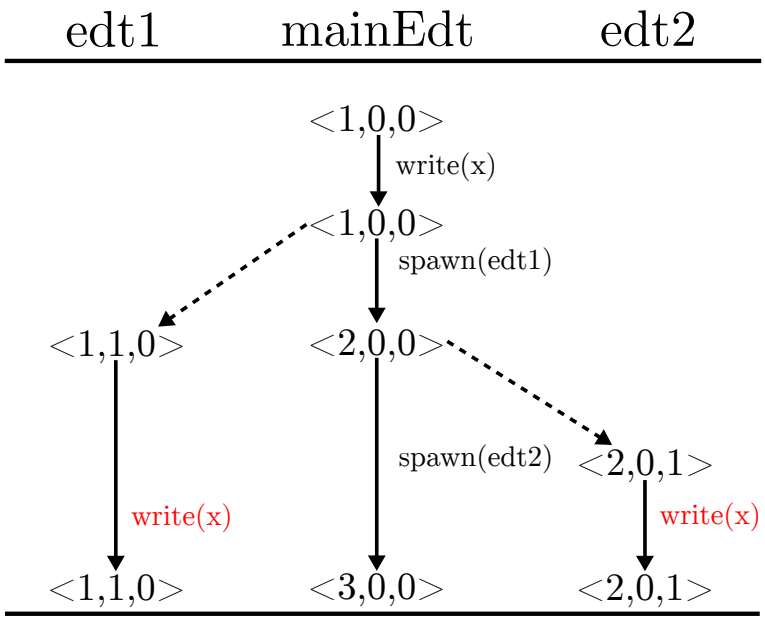


Figure 2.4 : Vector Clock Transition

The main disadvantage of the vector clock approach lies in its time and space overhead. For an OCR program containing $n$ simultaneously live EDTs, it requires $O(n^2)$ space as each EDT acquires an $n$-length vector clock. The time complexity of comparing two operations is $O(n)$ as it loops through every element of the vector clock to check the happens-before relation. Considering that in some real-world applications there exists thousands of simultaneously live EDTs, vector clocks can easily exhaust

available memory. Although there are some revised implementations of vector clocks that reduce the time complexity to $O(1)$ [4, 25], the memory overhead of storing a vector clock in each thread still restricts its usage.

# Chapter 3

# Graph Traversal based Data Race Detection Algorithm

## 3.1 Reachability Graph

A Reachability Graph (also called a computation graph) [26, 5, 27, 6, 28] is a directed acyclic graph which encodes the happens-before relation between operations. A node denotes operation (including spawning an EDT, reading / writing a memory location, synchronization by an event and returning an DB), and an edge denotes an ordering constraint. There are three different kinds of edges in a reachability graph.

- **Continue Edge** represents the sequential execution order within an EDT. All operations belonging to the same EDT are connected by continue edges.
- **Spawn Edge** represents the parent-child relationship. The first operation in child EDT executes after the spawn operation in parent EDT, so the corresponding two nodes are linked by a spawn edge. All spawn edges constitute the spawn tree which encodes ancestor-descendant relationship among all EDTs.
- **Join Edge** represents dependences between EDTs. Since the first operation of an EDT executes after the last operation of all dependent EDTs, join edges link the EDT to all dependent EDTs.

For the array sum program in Listing 2.1, the associated reachability graph is displayed in Figure 3.1. All operations inside an EDT are linked by continue edges.

*main_edt* spawns three child EDTs iteratively, so there are spawn edges linking spawn operation to the beginning of child EDTs. *edt3* depends on the partial sum from preceding EDTs, so there are join edges linking *sync_event* to *edt3* (An event is also considered to be an operation).

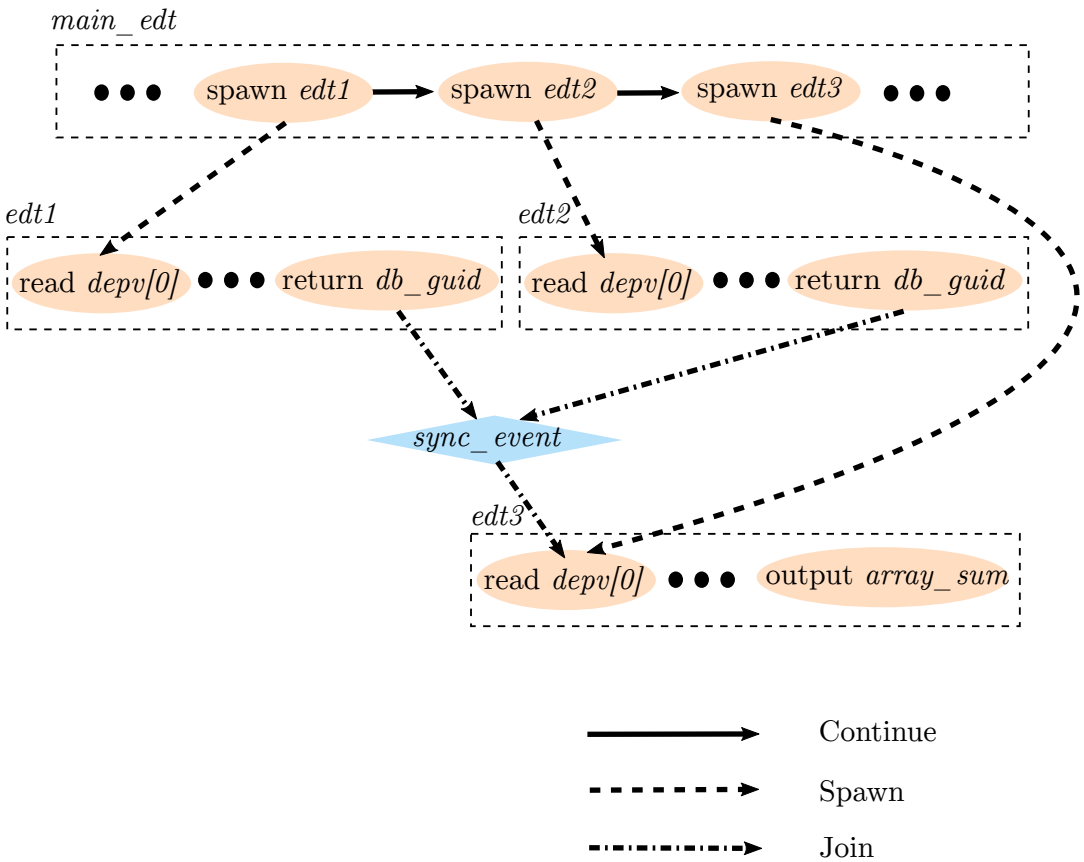


Figure 3.1 : Reachability Graph

The happens-before relation between two operations can be defined using the reachability graph. For two operations $a$, $b$ and their corresponding nodes $na$, $nb$, if there exists a directed path from $na$ to $nb$ such that any two contiguous nodes on the path are connected by an edge, then $a$ happens before $b$. If there is no such ordering from $a$ to $b$ or $b$ to $a$, they may happen in parallel.

## 3.2 Race Detection

Based on the reachability graph, we propose an on-the-fly data race detection algorithm for OCR programs. It defines two fundamental data structures: reachability graph and shadow memory. The reachability graph is dynamically constructed along with the execution of OCR programs, and the race detection algorithm leverages the reachability graph to check whether memory accesses are ordered correctly. In the original reachability graph shown in Figure 3.1 every single operation has a unique node, which incurs high memory overhead. To reduce memory usage, the race detection algorithm makes use of a revised reachability graph. As on the example displayed in Figure 3.2, each node represents a straight-line operation sequence ending with spawn or return operation (epoch node). The operation sequence is defined as an *epoch* (The definition of epoch is different from that in vector clocks [4]) and continue edges inside an epoch node are omitted. Each epoch node is identified by a unique *epoch* ID. In contrast with the original graph, the memory space occupied by revised reachability graph is within the same order of magnitude as the number of EDTs.

The shadow memory (SM) is responsible for recording previous memory accesses. Every byte of shared memory used by the program is associated with an SM instance which stores meta data for race detection [29]. According to [30], all concurrent reads and the latest write to a memory byte should be stored in its SM instance, in order not to miss any data race. Considering that in OCR, the only approach to sharing data among EDTs is through DBs and DBs can be moved between the execution of EDTs, we use the GUID of a DB and the offset from the starting address to uniquely locate an SM instance regardless of whether the start address is changed.

The pseudocode of the race detection algorithm is shown in Algorithm 3.1-Algorithm 3.5.

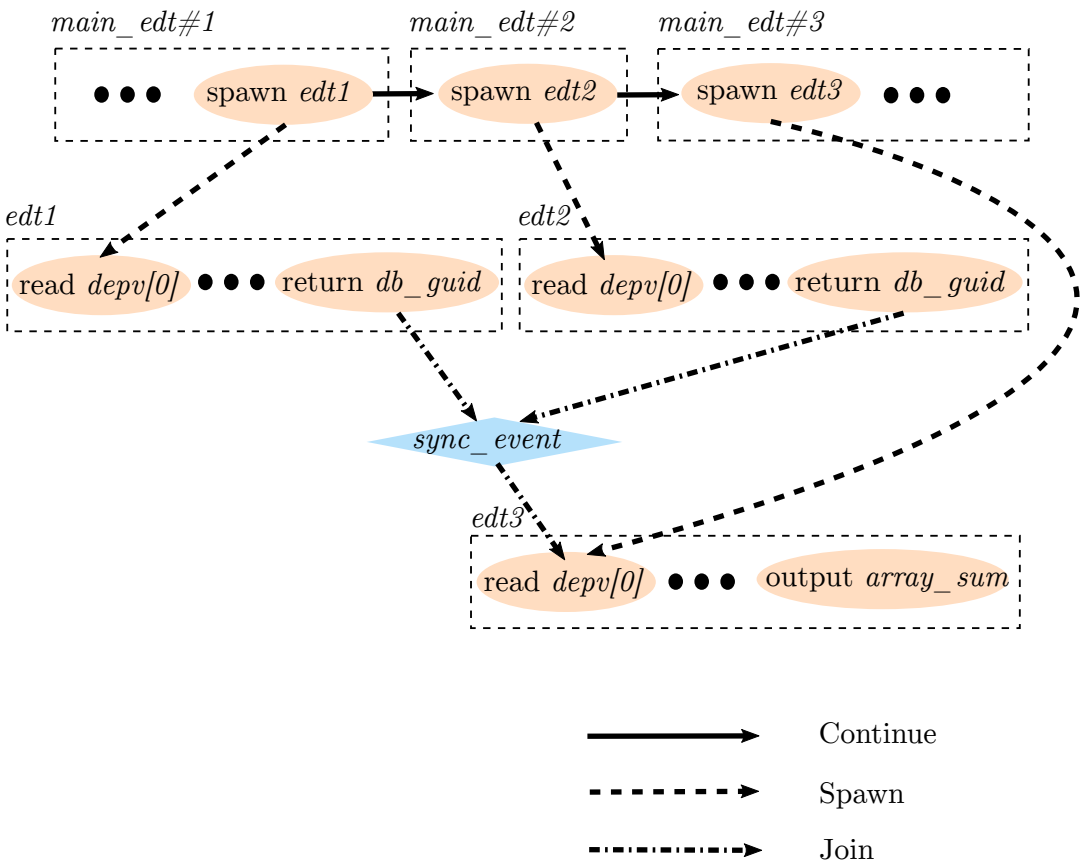


Figure 3.2 : Revised Reachability Graph

There are two global data structure, *reachability_graph* and *db_map*. *reachability_graph* is an adjacency list representation of reachability graph. *db_map* is an index for SM instances which binds a DB's GUID to its SM instance array. The input parameter *op* represents the monitored operation from which the algorithm extracts runtime information for analysis. The race detection algorithm consists of four modules:

- ***race_detector*** is the central module of the whole algorithm, which is shown in Algorithm 3.1. It monitors program execution and selects a proper auxiliary module to tackle encountered operation. All OCR library calls and DB accesses have a corresponding handler. For instance, *race_detector* will redirect to *initialize_shadow_memory* for calls to *ocrDbCreate*, to initialize SM instances for

the created DB.

**Algorithm 3.1** Race Detector

```
1:  procedure RACE_DETECTOR
2:      while program issues an operation op do
3:          if IS_OCR_LIBRARY_CALL(op) then
4:              if IS_DB_CREATE(op) then
5:                  INITIALIZE_SHADOW_MEMORY(op)
6:              else
7:                  UPDATE_GRAPH(op)
8:              end if
9:          else if IS_DB_ACCESS(op) then
10:             CHECK_DATA_RACE(op)
11:         end if
12:     end while
13: end procedure
```

- ***initialize_shadow_memory*** is responsible for initializing an SM instance for each byte of shared memory. As shown in Algorithm 3.2, this procedure is called upon DB creation, which creates SM instances according to DB size. These instances are stored in *db_map* and DB's GUID is used to retrieve them.

- ***update_graph*** keeps the reachability graph up-to-date. As shown in Algorithm 3.3, it performs a different action for each kind of OCR library call. If the library call is to spawn an EDT, a corresponding node is inserted into *reach-*

**Algorithm 3.2** Initialize Shadow Memory

```
1: procedure INITIALIZE_SHADOW_MEMORY(op)
2:     id = op.db_id, size = op.db_size
3:     sm = new SM[size]
4:     db_map[id] = sm
5: end procedure
```

*ability_graph* and the node for parent EDT links to it through a spawn edge. As spawning EDT means the end of parent EDT's current epoch, a node for next epoch is also inserted and parent EDT links to it through an continue edge. If the library call is to spawn an event, only a corresponding node is inserted. If the library call is to specify dependence, a join edge is inserted between the two involved nodes. Currently we does not remove terminated epochs or events from the graph.

- ***check_data_race*** carries out reachability check to detect data race. It is shown is Algorithm 3.4. When the program issues a shared memory access, it first calls $findSM$ to retrieve the associated SM instance from *db_map*. For a read to DB, *check_data_race* calls *check_reachability* once to see whether the read and latest write are ordered. *check_reachability* is a breadth-first search on the reachability graph. It returns true if the corresponding nodes of the two operations are reachable. The detail of *check_reachability* is shown in Algorithm 3.5. If *check_reachability* returns true, *check_data_race* adds the read into the associated SM instance for future data race detection, otherwise reports write-read race. For a write to DB, *check_data_race* compares it with all recorded reads and write to detect read-write race and write-write race. If the write is reachable

from all recorded memory accesses, *check_data_race* clears out the SM instance and sets the write as latest write since it happens after all previous memory accesses to the same memory location.

**Algorithm 3.3** Update Reachability Graph

```
1:  procedure UPDATE_GRAPH(op)
2:      if IS_EDT_CREATE(op) then
3:          id = op.epoch_id
4:          n = new EpochNode(id)
5:          reachability_graph[id] = n
6:          parent = reachability_graph[op.parent_id]
7:          parent.add_spawn_edge(n)
8:          new_epoch = new EpochNode(parent)        ▷ create a node for next epoch
9:          parent.add_continue_edge(new_epoch)
10:     else if IS_EVENT_CREATE(op) then
11:         id = op.event_id
12:         reachability_graph[id] = new EventNode(id)
13:     else if IS_ADD_DEPENDENCE(op) then
14:         src = reachability_graph[op.src_id]
15:         dst = reachability_graph[op.dst_id]
16:         src.add_join_edge(dst)
17:     end if
18: end procedure
```

**Algorithm 3.4** Check Data Race

```
1: procedure CHECK_DATA_RACE(op)
2:     sm = findSM(op.db_id, op.addr)
3:     if IS_READ(op) then
4:         is_ordered = CHECK_REACHABILITY(sm.write, op.epoch_id)
5:         if !is_ordered then
6:             report write-read race
7:         end if
8:         sm.read.add(op.epoch_id)
9:     else if IS_WRITE(op) then
10:        is_ordered = CHECK_REACHABILITY(sm.write, op.epoch_id)
11:        if !is_ordered then
12:            report write-write race
13:        end if
14:        for all r in sm.read do
15:            is_ordered = CHECK_REACHABILITY(r, op.epoch_id)
16:            if !is_ordered then
17:                report write-write race
18:            end if
19:        end for
20:        sm.write = op.epoch_id
21:        sm.read = ∅
22:    end if
23: end procedure
```

**Algorithm 3.5** Check Reachability

```
procedure CHECK_REACHABILITY(src_id, dst_id)
    src = reachability_graph[src_id]
    dst = reachability_graph[dst_id]
    reached_nodes = ∅
    queue = src
    while queue ≠ ∅ do
        next = queue.pop
        if reached_nodes.contain(next) then
            continue
        else
            reached_nodes.add(next)
        end if
        if next == dst then
            return true
        end if
        queue.add_all(next.edges)
    end while
    return false
end procedure
```

## 3.3 Complexity Analysis

Let us assume an OCR program allocates $\alpha$ EDTs, $\beta$ DBs and $\gamma$ events during the execution, and it calls *ocrAddDependence* $\delta$ times. The space complexity of the reachability graph is:

- **Epoch Node.** Every time the program spawns an EDT, the race detection algorithm allocates an epoch node for the EDT, and another node for its parent, so in total there are $2\alpha$ epoch nodes.
- **Event Node.** Every time the program spawns an event, the race detection algorithm allocates an event node for it, so in total there are $\gamma$ event nodes.
- **Continue Edge.** Every time the program spawns an EDT, the race detection algorithm inserts a continue edge between the node for parent's current epoch and the node for the next epoch, so in total there are $\alpha$ continue edges.
- **Spawn Edge.** Every time the program spawns an EDT, the race detection algorithm inserts a spawn edge between the parent and child, so in total there are $\alpha$ spawn edges.
- **Join Edge.** Every time the program calls *ocrAddDependence*, the race detection algorithm inserts a join edge between the source and destination, so in total there are $\delta$ join edges.

The size of reachability graph is $O(4\alpha + \gamma + \delta)$.

The SM index is a map binding each DB to its SM instance array, so the size of SM index is $O(\beta)$. For each byte in DB, the SM instance requires O($\alpha + 1$) space in the worst case when all EDTs read it concurrently.

The time complexity of checking data race after a read operation is different from that after a write operation in term of the number of reachability check.

- **Read.** For a read operation to a DB, the race detection algorithm carries out reachability check once between the read and the latest write to the same location. We use a breadth-first search on the reachability graph to check whether the operation is ordered correctly. In the worst case the checking process iterates over all nodes and edges, so the time complexity of reachability check is $O(4\alpha+\gamma+\delta)$. The time complexity of data race detection after a read is equal to the complexity of reachability check.

- **Write.** For a write operation to a DB, the race detection algorithm checks reachability with the latest write and all previous concurrent reads. In the worst case all EDTs reads the memory location, there is $\alpha$ concurrent reads, so the time complexity of data race detection after a write is $O((\alpha+1)(4\alpha+\gamma+\delta))$.

Compared with the complexity analysis for vector clock in Section 2.4, the space complexity of our algorithm is in the same order of magnitude as vector clock. The time complexity is larger than vector clock. However, in common case, our algorithm only incurs a comparable time overhead to vector clock. We explain this conclusion in Chapter 6. Furthermore, we propose several optimizations to mitigate the time overhead without increasing the space complexity. We introduce the detail of these optimizations in Chapter 5.

# Chapter 4

# Prototype Implementation

## 4.1 Prototype Design

We introduced a prototype race detection tool based on the race detection algorithm in Chapter 3. Figure 4.1 shows the architecture of our prototype. The prototype leverages Intel's Pin [8] to instrument and analyze the OCR program. Pin is a binary instrumentation framework for executables on the IA-32, Intel(R) 64 and Intel(R) Many Integrated Core architectures. It works like a virtual machine that interpreting the executable. The just-in-time(JIT) compiler generates new codes for the program until a branch statement is reached. Pin transfers control to the dispatcher to execute the generated sequence. Upon exiting the branch statement the JIT compiler regains control of the program and generates more codes for the branch target. In order to reuse the generated code efficiently, Pin stores them into a code cache.

To facilitate the development of program analysis tool, Pin provides instrumentation APIs to give access to the runtime information of the generated codes. In addition, Pin also allows attaching a user-defined handler with the generated code through instrumentation APIs. The program analysis tool executing on top of Pin is referred as PinTool [31]. It leverages instrumentation APIs to register handlers for the interested code.

The prototype is implemented as a Pintool. It registers handlers for OCR library calls and memory accesses to fulfill data race detection.

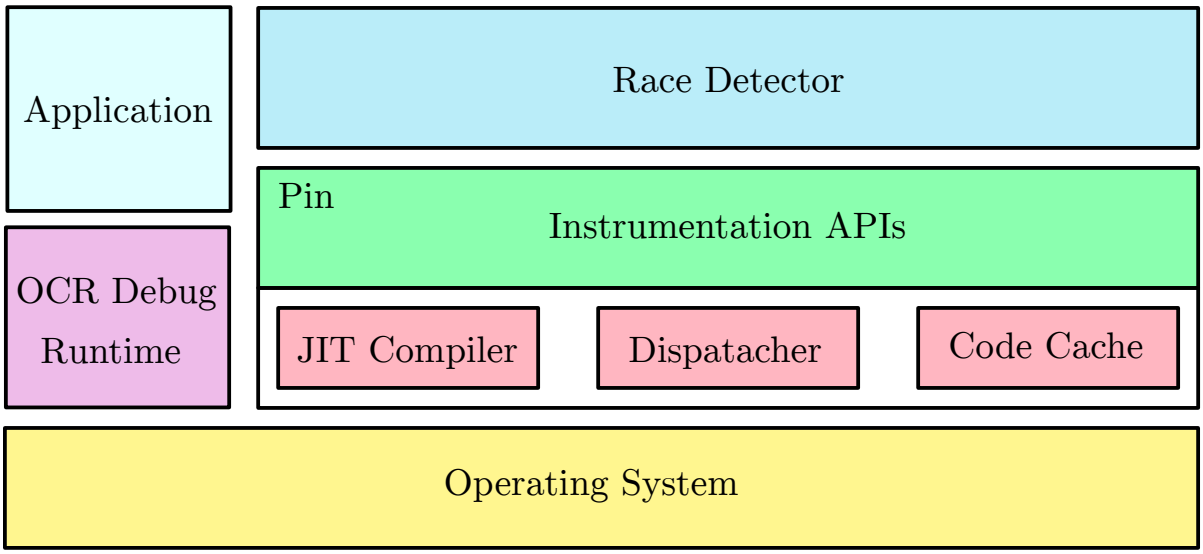


Figure 4.1 : Prototype Architecture

- **Library Call.** The corresponding handler takes cases of updating reachability graph and allocating shadow memory. Because Pin does not support registering a handler for a specific function call, The prototype does the registration in an indirect way with the help of an OCR debug runtime. The detail is introduced in Section 4.2.

- **Memory Access.** The corresponding handler takes care of detecting data race and recording the access in shadow memory. Since the OCR program executes in the same address space with Pin, the handler is registered directly by memory access instrumentation.

## 4.2 Register Handler for OCR Library Call

Because Pin does not support registering a handler for a specific function call, the prototype has to resort to the OCR debug runtime to implement the registration indirectly. For each OCR function, the OCR debug runtime have a corresponding "placeholder" function which executes upon the OCR function. The "placeholder" function takes same input parameters as the OCR function and its function body is blank. In instrumentation APIs, *RTN_ReplaceSignature* allows Pintools to replace the body of a specific function with another implementation before program execution. The prototype makes use of *RTN_ReplaceSignature* to replace the "placeholder" function with the handler.

Listing 4.1 shows the code snippet of registering a handler for *ocrDbCreate*. *notifyDbCreate* is the "placeholder" function corresponding to *ocrDbCreate*, and *afterDbCreate* is the handler. *RTN_ReplaceSignature* takes the "placeholder" function, the function signature and the handler as input. The prototype calls *RTN_FindByName* to locate *notifyDbCreate* in the executable, and *PTOTO_Allocate* to define the function signature. All parameters in the function signature are described by Pin's PARG macros [31]. After acquiring the "placeholder" function and its signature, the prototype replaces *notifyDbCreate* with *afterDbCreate*. In the later execution, *afterDbCreate* will be called automatically after *ocrDbCreate* to initialize shadow memory.

```
1 // locate function by name
2 RTN rtn = RTN_FindByName(img, "notifyDbCreate");
3 if (RTN_Valid(rtn)) {
4     // define function signature
5     PROTO proto_notifyDbCreate = PROTO_Allocate(
6         PIN_PARG(void), CALLINGSTD_DEFAULT, "notifyDbCreate",
7         PIN_PARG_AGGREGATE(ocrGuid_t), PIN_PARG(void*), PIN_PARG(u64),
8         PIN_PARG(u16), PIN_PARG_ENUM(ocrInDbAllocator_t),
9         PIN_PARG_END());
```

```
    // replace placeholder function with handler
    RTN_ReplaceSignature(
        rtn, AFUNPTR(afterDbCreate), IARG_PROTOTYPE,
        proto_notifyDbCreate, IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
        IARG_FUNCARG_ENTRYPOINT_VALUE, 1, IARG_FUNCARG_ENTRYPOINT_VALUE,
        2, IARG_FUNCARG_ENTRYPOINT_VALUE, 3, IARG_FUNCARG_ENTRYPOINT_VALUE, 4, ↩
            IARG_END);
    PROTO_Free(proto_notifyDbCreate);
}
```

Listing 4.1: Register Library Call Handler

## 4.3 Register Handler for Memory Access

The prototype leverages the memory access instrumentation features in Pin to register handlers for memory access. Listing 4.2 shows the code snippet for registration. *tackleMemRead* and *tackleMemWrite* are two handlers for memory accesses. *InsertPredicatedCall* is the instrumentation API that attaches a handler to an instruction. Because our algorithm only concentrates on memory accesses when detecting race, all unrelated instructions are filtered out. The prototype registers handlers for all instructions if their operands are related to memory access. The handler will be called automatically upon completion of these instructions.

```
// filtering out unrelated instructions
if (isIgnorableIns(ins)) return;
if (INS_IsAtomicUpdate(ins)) return;
uint32_t memOperands = INS_MemoryOperandCount(ins);
// iterate over each memory operand of the instruction.
for (uint32_t memOp = 0; memOp < memOperands; memOp++) {
    if (INS_MemoryOperandIsRead(ins, memOp)) {
        INS_InsertPredicatedCall(ins, IPOINT_BEFORE, (AFUNPTR)tackleMemRead,
            IARG_MEMORYOP_EA, memOp, IARG_MEMORYREAD_SIZE, IARG_REG_VALUE,
            REG_STACK_PTR, IARG_INST_PTR, IARG_END);
    }
    if (INS_MemoryOperandIsWritten(ins, memOp)) {
```

```
13          INS_InsertPredicatedCall(ins, IPOINT_BEFORE, (AFUNPTR)tackleMemWrite,
14              IARG_MEMORYOP_EA, memOp, IARG_MEMORYWRITE_SIZE, IARG_REG_VALUE,
15              REG_STACK_PTR, IARG_INST_PTR, IARG_END);
16      }
17  }
```

Listing 4.2: Register Memory Access Handler

# Chapter 5

# Optimization

## 5.1 Ideas to Optimize Data Race Detection

According to the complexity analysis in Section 3.3, the time complexity of our algorithm is not competitive with vector clock. In order to optimize the original race detection algorithm, we first analyze the OCR benchmarks with respect to *a* ) read / write operations, *b* ) number of EDTs, events, and edges. Table 5.1 shows the analysis of the OCR benchmarks (see Section 6.2 for the description of OCR benchmarks). Most of the benchmarks issue more read operations than write operations, but we observe that the number of read operations and write operations are in the same order of magnitude. *Smith-Waterman* and *XSBench* are two extreme cases in which read operations are dominant. Since the race detection after a write operation is much more expensive, the overall time overhead can be reduced if we apply certain optimization for the write. Also, we observe that the number of nodes and edges in the reachability graph are close, which means the graph is sparse. We should insert some shortcuts in the reachability graph to reduce the execution time of breadth-first search.

Apart from the optimization on the algorithm, we can also use cache to avoid unnecessary reachability check. Because EDTs acquiring the same DB usually access the DB repeatedly, and the accessed regions usually overlap, it is highly possible that the reachability between the same pair of EDTs is rechecked. We can implement

a cache to reuse the reachability check result. Table 5.2 shows the statistics on reachability check. In most cases more than 99% of reachability checks are redundant, so the cache is indispensable to reduce the time overhead.

| Benchmark | Memory Access | | Reachability Graph | | |
|---|---|---|---|---|---|
| | Read | Write | Epoch | Event | Edge |
| Cholesky | 3098400 | 2666400 | 222 | 605 | 1101 |
| FFT | 106496 | 172036 | 5 | 4 | 11 |
| Fibonacci | 1768 | 704 | 267 | 177 | 620 |
| Quicksort | 99160 | 71544 | 399 | 795 | 1986 |
| Smith-Waterman | 128908 | 100 | 26 | 108 | 196 |
| Task-Priorities | 0 | 0 | 13 | 5 | 38 |
| NQueens | 0 | 0 | 2285652 | 2285651 | 6856952 |
| UTS | 18377244 | 5744938 | 302014 | 111116 | 826233 |
| RSBench | 108521980 | 597928 | 30033 | 50 | 60136 |
| XSBench | 26063580 | 72 | 36835 | 52 | 73742 |

Table 5.1 : OCR Benchmark Statistics

## 5.2 Reachability Cache

We add a reachability cache to reuse the result of previous reachability checks when implementing the prototype. It is a global map that the record of reachability can be retrieved by the IDs of two nodes. Since the number of unique checks in OCR bench-

| Benchmark | Unique Check | Redundant Check |
|---|---:|---:|
| Cholesky | 550 | 5764250 |
| FFT | 3 | 278529 |
| Fibonacci | 465 | 2359 |
| Quicksort | 1767 | 231313 |
| Smith-Waterman | 81 | 128927 |
| Task-Priorities | 0 | 0 |
| NQueens | 0 | 0 |
| UTS | 549178 | 24014844 |
| RSBench | 30029 | 109089879 |
| XSBench | 73632 | 25990020 |

Table 5.2 : Reachability Check Statistics

marks are not large, currently we just store all known reachability. In future works, we may fix the cache size and manage the cache through certain cache replacement policies [32].

## 5.3 Reversed Reachability Graph

In order to reduce the times of reachability check after a write operation, we make use of *reversed reachability graph* which is the equivalent representation of the original graph. Reversed reachability graph reverses the direction of edges in the corresponding reachability graph.

- **Reversed Continue Edge.** A directed edge from an epoch node to the preceding epoch.

- **Reversed Spawn Edge.** A directed edge from the child EDT to the parent EDT.
- **Reversed Join Edge.** A directed edge from the destination EDT to the source EDT.

Figure 5.1 shows an example transformed from Figure 3.2. The definition of happens-before relation is the same as the original reachability graph. Based on the reversed reachability graph, we change the strategy of data race detection that the breadth-first search starts at the unchecked write operation and try to reach all previous accesses. The algorithm outputs all unreachable accesses after the search.

The pseudocode of the updated data race detection algorithm is listed in Algorithm 5.1 and Algorithm 5.2. For both read and write operation *check_data_race* only calls *check_reachability* once. *check_reachability* checks the reachability between *dst* and *src_set*. It returns unreachable accesses, namely data races, after the breadth-first search.

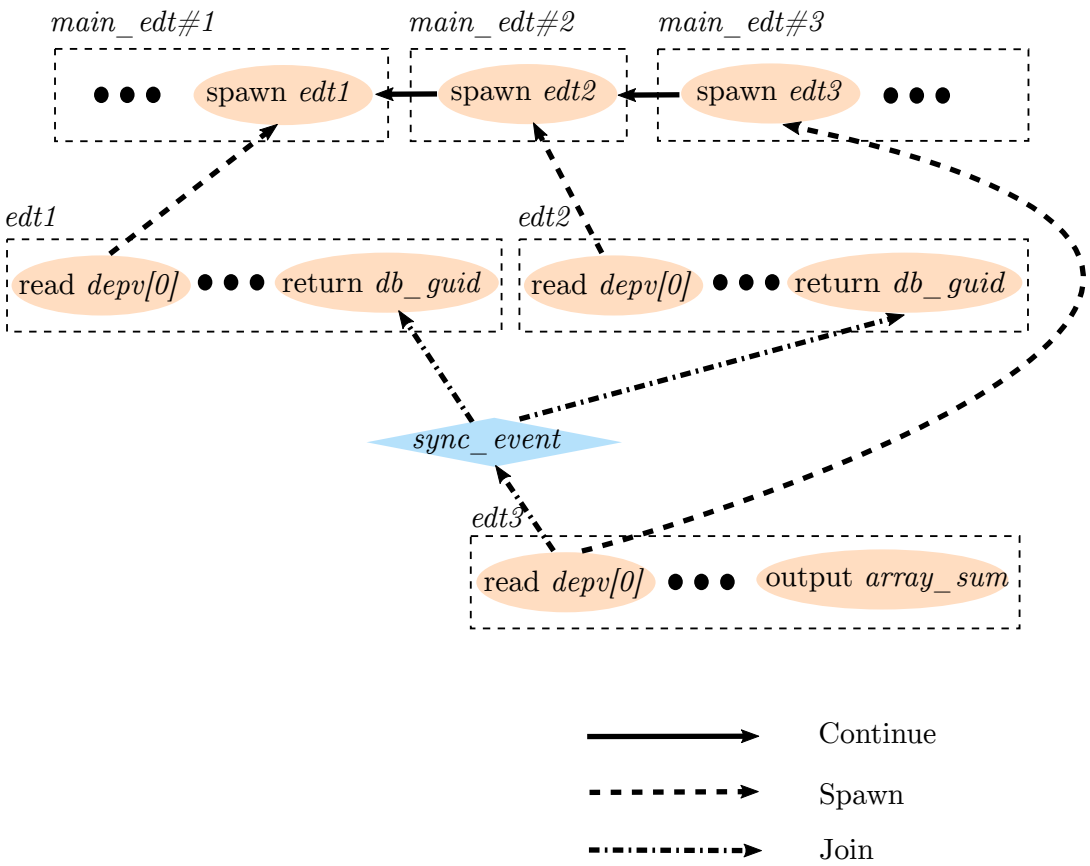


Figure 5.1 : Reversed Reachability Graph

**Algorithm 5.1** Check Data Race

```
1: procedure CHECK_DATA_RACE(op)
2:     sm = findSM(op.db_id, op.addr)
3:     if IS_READ(op) then
4:         unreached_nodes = CHECK_REACHABILITY(sm.write, op.epoch_id)
5:         if !unreached_nodes.empty() then
6:             report write-read race
7:         end if
8:         sm.read.add(op.epoch_id)
9:     else if IS_WRITE(op) then
10:        preceding =
11:        preceding.add_all(sm.read)
12:        preceding.add(sm.write
13:        unreached_nodes = CHECK_REACHABILITY(preceding, op.epoch_id)
14:        for all uinunreached_nodes do
15:            if IS_READ(u) then
16:                report read-write race
17:            else
18:                report write-write race
19:            end if
20:        end for
21:        sm.write = op.epoch_id
22:        sm.read = ∅
23:    end if
24: end procedure
```

**Algorithm 5.2** Check Reachability on Reversed Reachability Graph

```
1:  procedure CHECK_REACHABILITY(src_id_list, dst_id)
2:      src_set = ∅
3:      for all id in src_id_list do
4:          src_set.add(reachability_graph[id])
5:      end for
6:      dst = reachability_graph[dst_id]
7:      reached_nodes = ∅
8:      queue = dst
9:      while queue ≠ ∅ do
10:         next = queue.pop
11:         if reached_nodes.contain(next) then
12:             continue
13:         else
14:             reached_nodes.add(next)
15:         end if
16:         if src_set.contain(next) then
17:             src_set.remove(next)
18:         end if
19:         queue.add_all(next.edges)
20:     end while
21:     return src_set
22: end procedure
```

## 5.4 Path Compression

The time overhead in the reachability check is related to the depth of breadth-first search. Since the search does not move to next level unless it has iterated over all nodes in the current level, reachability checks on a large graph are always slow.

In some cases, the reachability cache can help reduce the depth of breadth-first search. For instance, Figure 5.2 shows an implementation of task-loop in OCR. *edt_init* sets up the configuration in *db_config* and all consequent EDTs in the task loop read the configuration. For each reachability check from *edt_init* to *edt_i*, the bread-first search only takes one step in depth since the reachability cache already records that *edt_init* is reachable from *edt_i-1*.

However, in mosts cases that the program contains events, the reachability cache is not helpful for mitigating the depth. Figure 5.3 shows an implementation of two-dimensional task loop. Similar to Figure 5.2, *db_config* stores the global configuration. EDTs in the same inner loop run in parallel and a synchronization is placed between two continuous iterations of the outer loop. For any reachability check between *edt_init* and *edt_ij*, the reachability cache keeps missing until the search reaches a task spawned in the previous iteration of the outer loop.

To reduce the depth of search in Figure 5.3, we can add additional edges to the event nodes to compress the frequently accessed paths between *edt_init* and event nodes. After the reachability check from *edt_init* to *edt_10*, we already know that *sync_1* is reachable from *edt_init*. We can link an edge from *edt_init* to *sync_1*. Later when we check the reachability between *edt_init* to any *edt_1i*, the bread-first search will jump to *sync_1* instantly without reaching any *edt_0i*.

Inspired by the observation in Figure 5.3, we propose a path compression algorithm. The pseudocode is listed in Algorithm 5.3. The algorithm analyzes the found

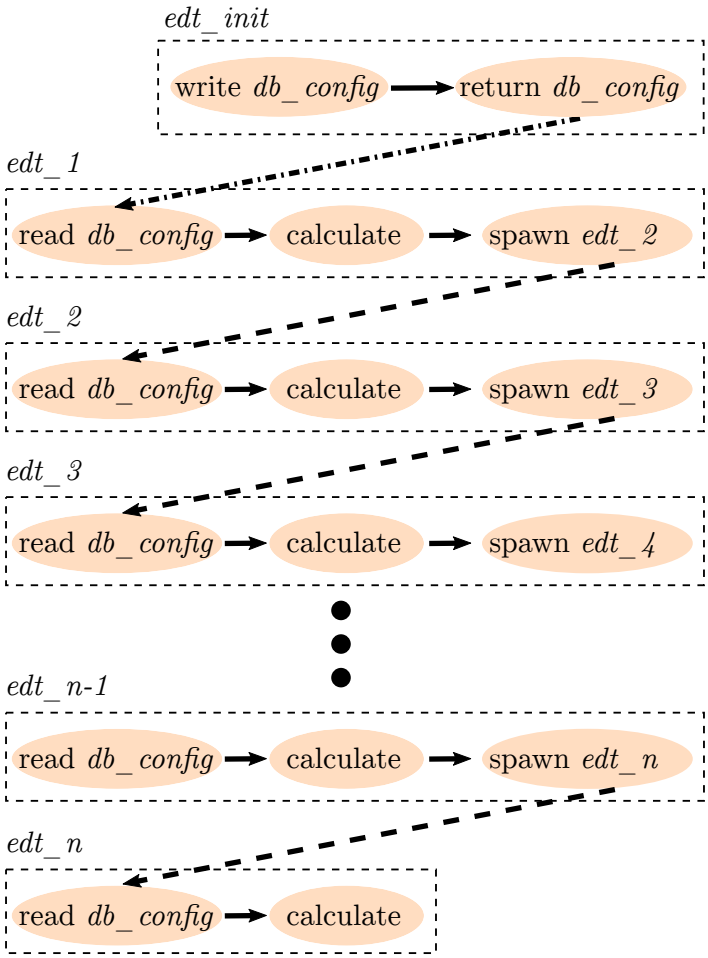


Figure 5.2 : Reachability Graph for Task Loop

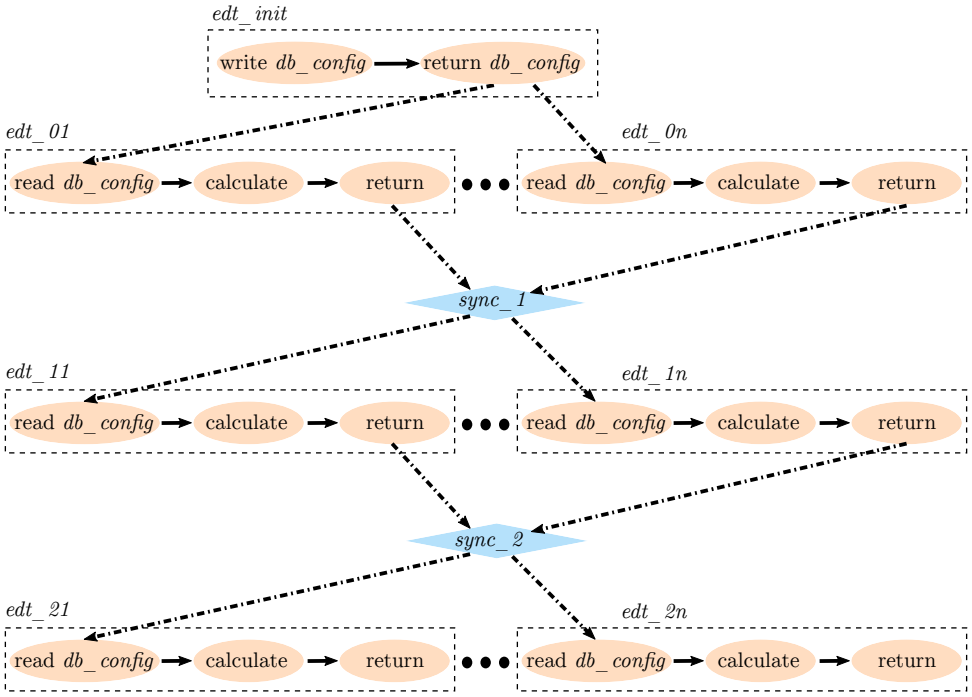


Figure 5.3 : Reachability Graph for Two Dimension Task Loop

path after a breadth-first search. If the path length is larger than a threshold, the algorithm will iterate over all nodes on the path to find out the node with largest in-degree which is referred as *key_node* in the pseudocode. The algorithm compresses the two subpaths * *a* ) from the source to *key_node* *b* ) from the *key_node* to the destination by adding additional edges between these nodes.

Since a node with large in-degree is the intersection of multiple paths, it is more likely to appear on the final path. Compressing the path starting and ending at the node is helpful for reducing the depth of future breadth-first searches.

*A subpath is a section of a larger path

**Algorithm 5.3** Path Compression

```
procedure PATH_COMPRESSION(path)
    key_node = ∅, in_degree = 0
    if path.length > threshold then
        for node in path do
            if node.in_degree >= in_degree then
                key_node = node, in_degree = node.in_degree
            end if
        end for
        add an edge from path.src to key_node
        add an edge from key_node tp path.des
    end if
end procedure
```

# Chapter 6

# Evaluation

## 6.1 Environment

To evaluate the performance of the graph traversal based data race detection algorithm, we carry out several experiments using the OCR benchmarks. All experiments are conducted on an Intel workstation. The hardware configuration is listed below.

- **CPU:** 24-core Intel(R) Xeon(R) CPU E5-2667, 2.90GHz
- **Memory:** 125 GB
- **OS:** Ubuntu 15.04

All selected OCR benchmarks execute on top of a customized OCR v1.1 runtime. The race detection tool executes on top of Pin 3.2. OCR benchmarks, the OCR runtime and the race detection tool are all compiled by GCC 4.9.2 with -O3 optimization. Because of the limitation in our race detection tool, all OCR benchmarks execute sequentially.

## 6.2 Benchmark

We select 10 OCR benchmarks from the *ocr app* repository [33] to evaluate the race detection algorithm. These benchmarks are either scientific computing program or mini app from real world applications.

- **Cholesky:** A tiled cholesky decomposition.
- **FFT:** A fast fourier transform implementation.
- **Fibonacci:** A Fibonacci number calculation program for the given index.
- **Quicksort:** A quick sort implementation for a randomly generated integer sequence.
- **Smith-Waterman:** A Smith-Waterman algorithm implementation.
- **Task-Priorities:** A test for EDT priority.
- **NQueens:** A bitwise recursive algorithm that computes the number of solutions to the N-queens problem.
- **UTS:** An exhaustive search on an unbalanced tree.
- **RSBench:** A mini-app to represent the multipole resonance representation look up cross section algorithm.
- **XSBench:** A mini-app to represent a key computational kernel of the Monte Carlo neutronics application OpenMC.

According to the statistics in Table 5.1 and Table 5.2, we observe that *Cholesky*, *FFT*, *Fibonacci* , *Quicksort*, *Smith-Waterman*, *Task-Priorities* are small-scale benchmarks and *NQueens*, *UTS*, *RSBench*, *XSBench* are large-scale benchmarks. Besides, *Task-Priorities* and *NQueens* do not have any access to DB during the execution.

## 6.3 Result & Analysis

We execute benchmarks in different execution modes to evaluate the time overhead incurred by different modules of our race detection tool. Table 6.1 shows the execution

time of the benchmarks. The first column lists the benchmark name. The second - fifth columns list the execution time in a corresponding mode.

- **Binary.** Execute a benchmark on the underlying operating system.
- **Pin.** Execute a benchmark on top of Pin.
- **Instrumentation.** Execute a benchmark on top of Pin and instrument all memory accesses.
- **Graph Traversal.** Execute a benchmark on top of Pin and detect data race with the original graph traversal based data race detection algorithm.

We can make a number of observations from the data in Table 6.1.

- The slowdown incurred by Pin varies significantly on different benchmarks (5.576x-177.333x), and it decreases with the increase of execution time. The difference of slowdown is due to code cache in Pin. Code cache can avoid redundant code generation for the statements executed repeatedly. In OCR programs EDT templates store the common code of spawned EDTs, which can be stored in code cache to accelerate the code generation of all spawned EDTs. Since *NQueens*, *UTS*, *RSBench*, and *XSBench* create much more EDTs than other benchmarks, they benefit more on performance from the code cache.
- The time overhead of instrumentation depends on the number of shared memory accesses. Since large-scale benchmarks issue more memory accesses, the instrumentation incurs larger slowdown. In all benchmarks except *RSBench*, the slowdown to Pin mode is less than 5.3X.
- The unoptimized race detection algorithm is time-consuming. The time overhead of race detection is related to the number of reachability checks. Since

| Benchmark | Binary | Pin | Instrumentation | Graph Traversal |
|---|---|---|---|---|
| Cholesky | 0.018 | 1.300 | 1.952 | 10.280 |
| FFT | 0.008 | 1.086 | 1.156 | 1.300 |
| Fibonacci | 0.007 | 1.088 | 1.058 | 1.070 |
| Quicksort | 0.012 | 1.046 | 1.210 | 2.170 |
| Smith-Waterman | 0.006 | 1.064 | 1.116 | 1.170 |
| Task-Priorities | 0.006 | 1.034 | 1.032 | 1.040 |
| NQueens | 3.525 | 31.434 | 32.220 | 31.970 |
| UTS | 1.088 | 7.380 | 13.900 | >1 hour |
| RSBench | 0.717 | 3.998 | 60.514 | >1 hour |
| XSBench | 0.664 | 3.768 | 20.014 | >1 hour |

Table 6.1 : Execution Time of Race Detection in Second

the time of reachability check is proportional to the number of nodes and edges in the reachability graph, the race detection algorithm incurs huge overhead to large-scale benchmarks. For the three large benchmarks *UTS*, *RSBench*, and *XSBench*, it cannot finish the race detection in one hour. The thousands of times of slowdown is unacceptable.

We also carry out experiments to evaluate the three optimization strategies. The result is shown in Table 6.2. The first column lists the benchmark name, and the following three columns list the execution time under different combinations of optimizations. According to the data in Table 6.2 we can draw several conclusions.

- Reachability cache is indispensable for data race detection. From Table 5.2 we know more than 99% of reachability checks are unnecessary. With reachability

cache, our race detection tool can accomplish race detection on all benchmarks in one hour. For all small-scale benchmarks except *Cholesky*, the execution time is close to the time in Pin mode. For *UTS*, *RSBench*, and *XSBench*, the slowdown to Pin mode is 40.122X-536.687X. Since the slowdown becomes significant when executing large-scale benchmarks, the race detection algorithm should apply other optimizations to tackle real world applications.

- Reversed reachability graph does not has a significant effect on the selected OCR benchmarks. The execution time is similar to the time of only applying reachability cache. Since the race detection algorithm executes few reachability checks for small-scale benchmarks, the reduction of execution time is not obvious. For *UTS*, *RSBench*, and *XSBench*, most of the calculation is looking up the data in a DB after the DB is initialized. They issue few write operations after concurrent read operations, so the reversed reachability graph does not have a significant effect.

- Path compression significantly reduces the time overhead of race detection on large-scale benchmarks. The slowdown to Pin mode is 5.283X-89.891X, which is around one-eighth of the slowdown without path compression. Since path compression can reduce the depth of bread-first search during a reachability check, it is quite effective for large-scale benchmarks. For small-scale benchmarks, path compression does not have a significant effect since most of the found paths are shorter than the threshold. Table 6.3 shows the statistics on edges, the number of additional edges is less than one-third of the number of original edges, which means path compression incurs acceptable space overhead and the space complexity is similar to the original algorithm.

| Benchmark | Cache | Cache + Reversed Reachability Graph | Cache + Reversed Reachability Graph + Path Compression |
|---|---|---|---|
| Cholesky | 4.354 | 4.182 | 4.199 |
| FFT | 1.252 | 1.242 | 1.242 |
| Fibonacci | 1.060 | 1.032 | 1.060 |
| Quicksort | 1.368 | 1.344 | 1.337 |
| Smith-Waterman | 1.156 | 1.150 | 1.155 |
| Task-Priorities | 1.030 | 1.032 | 1.033 |
| NQueens | 32.178 | 32.524 | 32.066 |
| UTS | 296.104 | 293.812 | 38.995 |
| RSBench | 1138.532 | 1140.332 | 145.951 |
| XSBench | 2022.240 | 2034.518 | 338.713 |

Table 6.2 : Execution Time of Optimized Race Detection in Second

| Benchmark | Original Edge | Additional Edge |
|---|---:|---:|
| Cholesky | 1101 | 0 |
| FFT | 11 | 0 |
| Fibonacci | 620 | 0 |
| Quicksort | 1986 | 244 |
| Smith-Waterman | 196 | 0 |
| Task-Priorities | 38 | 0 |
| NQueens | 6856952 | 0 |
| UTS | 826233 | 236674 |
| RSBench | 60136 | 20019 |
| XSBench | 73742 | 29912 |

Table 6.3 : Statistics on Edge

# Chapter 7

# Related Work

## 7.1 Vector Clock

As described in Section 2.4, vector clock based data race detection algorithms [4, 25] check happens-before relations in the program by comparing the vector clocks of two memory accesses. Each thread / task holds a vector to record the latest preceding timestamps of other units. For an $n$-threaded program, vector clock requires $O(n)$ space for each thread and each memory location, and takes $O(n)$ time for each comparison between memory accesses. There are some other works that reduce the space overhead for a memory location, and time overhead for the comparison. For example, Flanagan proposed FastTrack [4] that replaces the heavyweight vector clock with an adaptive lightweight representation *epoch* when the operation is ordered. FastTrack improves the time complexity with no loss in precision, but it does not change the memory usage of vector clocks inside each thread.

According to the complexity analysis in Section 2.4, vector clock is not applicable to large-scale OCR programs since the size of vector clock is proportional to the maximum number of simultaneously live tasks, which may deplete available memory space.

## 7.2 Lockset

Lockset is a lightweight data race detection algorithm for lock-based multithreaded programs. It was first exemplified by Eraser [3]. Unlike vector clock that is based on happens-before relation, lockset detects data race according to a consistent locking discipline that every variable shared between threads should be protected by a lock during its life cycle. Lockset maintains a set for each shared variable which records the common locks protecting the variable. Every time the program issues a memory access, lockset intersects the set with holding locks of the access. If the set becomes empty, lockset reports a data race. Since the intersection is the only workload for detecting data race, lockset only takes O(1) time for each memory access. To avoid unnecessary checkings on local and read-only variables, lockset utilizes a state machine to keep track of the variable state. It carries out data race detection only after the variable transits to shared-write state.

Since lock is not the only way to synchronize the program, lockset reports false positives for the memory locations which are protected by other synchronization methods such as barrier. Because OCR uses dependence to synchronize EDTs, we cannot apply lockset to OCR programs.

## 7.3 SP-bag / ESP-bag

SP-bag [5] is an efficient determinacy race detection algorithm for Cilk. It makes use of the structural information in a Cilk program to report exact race with low time and space overhead. In Cilk a task can only synchronize with sibling tasks spawned by the parent. Two tasks logically run in parallel unless they have common ancestors and a *sync* statement executes between them. SP-bag detects race based on this

observation.

SP-bag is a serial algorithm. It executes the program in a depth-first fashion. During the program execution, it maintains a S-bag and a P-bag for every task. The S-bag records preceding tasks and the P-bag records concurrent tasks. S-bag and P-bag are updated by the following rules.

- **Spawn task $A$.** $S_A = A, P_A = \emptyset$
- **Task $A$ returns to task $B$.** $P_B = P_B \bigcup S_A \bigcup P_A, S_A = \emptyset, P_A = \emptyset$
- **Task $A$ issues a synchronization.** $S_A = S_A \bigcup P_A, P_A = \emptyset$

SP-bag reports race according to the affiliation of the previous read / write operation.

- **Read Operation.** If the previous write to the same location belongs to a P-bag, then report race.
- **Write Operation.** If the previous read or write to the same location belongs to a P-bag, then report race.

For each memory location, SP-bag only records the latest read and write, incurring constant space overhead. Since SP-bag only performs lookups on the S-bag and P-bag for race detection, the time overhead is also low.

Derived from SP-bag, ESP-bag [6] extends the algorithm to tackle constructs in HJ [14]. Apart from tasks, ESP-bag also maintains a P-bag for finish scope. It adds two additional rules to support the semantics of finish scope.

- **Start a finish scope $F$.** $P_F = \emptyset$
- **End a finish scope $F$ in task $A$.** $S_A = S_A \bigcup P_F, P_F = \emptyset$

SP-bag and ESP-bag fully utilize the structural feature in spawn-sync and async-finish parallel runtimes to reduce time and space overhead. However, in most cases an OCR program cannot execute in a depth-first fashion, so both SP-bag and ESP-bag cannot be applied to OCR programs.

## 7.4 Dynamic Task Reachability Graph

Dynamic Task Reachability Graph(DTRG) [26] is a more compact representation than the computation graph that encodes the dependence in a parallel program. It records the reachability information in task level. Each task is denoted by a single node and dependence between tasks is denoted by an directed edge. If there exists a path between two tasks, the two tasks can only execute sequentially. Otherwise, they can run in parallel. DTRG classifies all edges into three categories: *a* ) continue edge, *b* ) spawn edge, *c* ) join edge, according to the type of dependences. DTRG is constructed on the fly, keeping the recorded reachability information up-to-date.

Based on DTRG [26], the author also proposes a serial determinacy race detection algorithm for task parallelism with futures. The race detection algorithm detects races against DTRG and spawning tree. It executes the program in a depth-first fashion and builds a corresponding DTRG. By eliminating non-tree joining edges it recovers the spawning tree from the graph. The race detection process contains two independent reachability checks. The algorithm reports data race if the both checks fail.

- **Reachability check on the spawn tree.** The algorithm checks the parent-child relation. By the labeling scheme in [34], each node in the spawn tree holds a label. Through a comparison between labels, the reachability can be determined in constant time.

- **Reachability check on the DTRG.** If the memory access fails to pass the reachability check on the spawn tree, the algorithm carries out a graph traversal to detect the reachability on the DTRG.

Since the reachability check on the spawn tree is efficient and in most cases the expensive graph traversal is unnecessary, the DTRG based algorithm can report exact race with low time overhead.

DTRG is pretty similar to our reachability graph. Both of them represents reachability on task-level and treats edges differently according to the type of dependences. In addition, both the DTRG based race detection algorithm and our algorithm utilize graph traversal to check the reachability between tasks. However, the DTRG based race detection algorithm targets for Habanero-Java. It requires that the program can execute in a depth-first fashion to calculate the label in the spawning tree. Since there is no guarantee that OCR program can execute in a depth-first fashion, we cannot apply the spawning tree based reachability check to optimize our algorithm.

# Chapter 8

# Conclusion & Future Work

## 8.1 Conclusion

In this dissertation, we introduce a *reachability graph* for EDP runtimes, a novel representation of the happens-before relation in an EDP program. The reachability graph expresses the happens-before relation as directed paths, taking into account the unique dependence properties of EDP runtimes. It enables the use of a graph traversal based data race detection algorithm. After one execution, the race detection algorithm can detect data races in all possible thread interleavings for the same input. In order to reduce the time complexity for race detection, we propose a few optimizations, such as *reachability cache* and *reversed reachability graph* to avoid unnecessary graph traversals and *path compression* to reduce the number of steps performed for graph traversal. Based on our race detection technique, we have developed a prototype implementation for the Open Community Runtime (OCR). Our evaluation on a set of open source OCR benchmarks shows that our tool handles all OCR constructs and incurs acceptable time and space overhead to the program execution.

## 8.2 Future Work

For future directions on our data race detection algorithm, we plan to add a static analysis pass before the dynamic data race detection to avoid unnecessary reachability check. We plan to learn the structural features in the program to eliminate terminated

nodes from the reachability graph.

For our race detection tool, We plan to reduce the size of injected code to improve efficiency. We plan to utilize other binary instrumentation frameworks to reduce the time overhead of instrumentation.

# Bibliography

[1] T. Mattson, R. Cledat, Z. Budimlic, V. Cave, S. Chatterjee, B. Seshasayee, R. van der Wijngaart, and V. Sarkar, "Ocr the open community runtime interface, version 1.0. 0," 2015.

[2] J. Dokulil and S. Benkner, "Retargeting of the open community runtime to intel xeon phi," *Procedia Computer Science*, vol. 51, pp. 1453–1462, 2015.

[3] 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 (TOCS)*, vol. 15, no. 4, pp. 391–411, 1997.

[4] C. Flanagan and S. N. Freund, "Fasttrack: efficient and precise dynamic race detection," in *ACM Sigplan Notices*, vol. 44, no. 6. ACM, 2009, pp. 121–133.

[5] M. Feng and C. E. Leiserson, "Efficient detection of determinacy races in cilk programs," *Theory of Computing Systems*, vol. 32, no. 3, pp. 301–326, 1999.

[6] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Scalable and precise dynamic datarace detection for structured parallelism," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 531–542, 2012.

[7] F. Mattern *et al.*, "Virtual time and global states of distributed systems," *Parallel and Distributed Algorithms*, vol. 1, no. 23, pp. 215–226, 1989.

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[9] Wikipedia, The Free Encyclopedia, "Task parallelism," 2017. [Online]. Available: https://en.wikipedia.org/wiki/Task_parallelism

[10] B. Barney, "Message passing interface," 2017. [Online]. Available: https://computing.llnl.gov/tutorials/mpi

[11] ——, "Posix threads programming," 2017. [Online]. Available: https://computing.llnl.gov/tutorials/pthreads

[12] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of parallel and distributed computing*, vol. 37, no. 1, pp. 55–69, 1996.

[13] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Acm Sigplan Notices*, vol. 40, no. 10. ACM, 2005, pp. 519–538.

[14] S. Imam and V. Sarkar, "Habanero-java library: a java 8 framework for multicore programming," in *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*. ACM, 2014, pp. 75–86.

[15] "Intel threading building blocks developer reference," 2017. [Online]. Available: https://www.threadingbuildingblocks.org/docs/help/index.htm

[16] "Openmp application programming interface," 2015. [Online]. Available: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[17] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, p. 6.

[18] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach *et al.*, "Concurrent collections," *Scientific Programming*, vol. 18, no. 3-4, pp. 203–217, 2010.

[19] S. Treichler, M. Bauer, and A. Aiken, "Realm: An event-based low-level runtime for distributed memory architectures," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014, pp. 263–276.

[20] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[21] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with mpi," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 712–725.

[22] D. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 237–252.

[23] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*. ACM, 2009, pp. 62–71.

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

[25] E. Pozniansky and A. Schuster, "Multirace: efficient on-the-fly data race detection in multithreaded c++ programs," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 3, pp. 327–340, 2007.

[26] R. Surendran and V. Sarkar, "Dynamic determinacy race detection for task parallelism with futures." in *RV*, 2016, pp. 368–385.

[27] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Efficient data race detection for async-finish parallelism," *Formal Methods in System Design*, vol. 41, no. 3, pp. 321–347, 2012.

[28] M. A. Bender, J. T. Fineman, S. Gilbert, and C. 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*. ACM, 2004, pp. 133–144.

[29] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *Proceedings of the 3rd international conference on Virtual execution environments*. ACM, 2007, pp. 65–74.

[30] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen, "A theory of data race detection," in *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*. ACM, 2006, pp. 69–78.

[31] "Pin 3.2 user guide," 2017. [Online]. Available: https://software.intel.com/sites/landingpage/pintool/docs/81205/Pin/html

[32] Wikipedia, The Free Encyclopedia, "Cache Replacement Policies," 2017. [Online]. Available: https://en.wikipedia.org/wiki/Cache_replacement_policies

[33] "OCR benchmark repository," 2017. [Online]. Available: https://xstack.exascale-tech.com/git/public/apps.git

[34] P. Dietz and D. Sleator, "Two algorithms for maintaining order in a list," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM, 1987, pp. 365–372.