RICE UNIVERSITY

Debugging, Repair, and Synthesis of Task-Parallel Programs

by

Rishi Surendran

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Vivek Sarkar

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

Swarat Chaudhuri
Associate Professor of Computer Science

Lin Zhong
Professor of Electrical and Computer Engineering

HOUSTON, TEXAS
MARCH, 2017

ABSTRACT

Debugging, Repair, and Synthesis
of Task-Parallel Programs

by

Rishi Surendran

Parallelizing sequential programs to effectively utilize modern multicore architectures is a key challenge facing application developers and domain experts. Therefore, it is a need of the hour to create tools that aid programmers in developing correct and efficient parallel programs. In this thesis, we present algorithms for debugging, repairing, and synthesizing task-parallel programs that can provide a foundation for creating such tools. Our work focuses on task-parallel programs with both imperative async-finish parallelism and functional-style parallelism using futures.

First, we address the problem of detecting races in parallel programs with async, finish and future constructs. Existing dynamic determinacy race detectors for task-parallel programs are limited to programs with strict computation graphs in which a task can only wait for some subset of its descendant tasks to complete. In this thesis, we present the first known determinacy race detector for non-strict computation graphs generated using futures. The space and time complexity of our algorithm degenerate to those of the classical SP-bags algorithm when using only structured parallel constructs such as spawn-sync and async-finish. In the presence of point-to-point synchronization using futures, the complexity of the algorithm increases by a factor determined by the number of future task creation and get operations as well as the number of non-tree edges in the computation graph.

Next, we introduce a hybrid static+dynamic test-driven approach to repairing data races in task-parallel programs. Past solutions to the problem of repairing parallel programs have used static-only or dynamic-only approaches, both of which incur significant limitations in practice. Static approaches can guarantee soundness in many cases but are limited in precision when analyzing medium or large-scale software with accesses to pointer-based data structures in multiple procedures. Dynamic approaches are more precise, but their proposed repairs are limited to a single input and are not reflected back in the original source program. Our approach includes a novel coupling between static and dynamic analyses. First, we execute the program on a concrete test input and determine the set of data races for this input dynamically. Next, we compute a set of static "finish" placements that repairs these races and also respects the static scoping rules of the program while maximizing parallelism.

Finally, we introduce a novel approach to automatically synthesize task-parallel programs with futures from sequential programs through identification of pure method calls. Our approach is built on three new techniques to address the challenge of automatic parallelization via future synthesis: candidate future synthesis, parallelism benefit analysis, and threshold expression synthesis. In *candidate future synthesis*, our system annotates pure method calls as async expressions and synthesizes a parallel program with future objects and their type declarations that are more precise than those from past work. Next, the system performs a novel *parallel benefit analysis* to determine which async expressions may need to be executed sequentially due to overhead reasons, based on execution profile information collected from multiple test inputs. Finally, *threshold expression synthesis* uses the output from parallelism benefit analysis to synthesize predicate expressions that can be used to determine at runtime if a specific pure method call should be executed sequentially or in parallel. These algorithms have been implemented and evaluated on a range of benchmark programs. The evaluation establishes the effectiveness of our approach with respect to dynamic data race detection overhead, compile-time overhead, and precision and performance of the repaired and synthesized code.

Acknowledgments

First, I would like to express my deepest gratitude to my thesis advisor Prof. Vivek Sarkar for his guidance, support, and encouragement academically as well as personally throughout my life as a graduate student. I thank him for providing me the opportunity to attend graduate school at Rice and be a part of the Habanero group. He has been a wonderful mentor and a facilitator. He has always accommodated me in his extremely busy schedule whenever I needed his guidance or support. It has been an honor and pleasure to have him as my advisor.

I would like to thank Prof. Swarat Chaudhuri for being part of my thesis committee and providing me guidance and feedback. I have had the pleasure of collaborating with him on the program repair work which is part of this thesis. His inputs on syntax-guided synthesis has also helped in synthesis of futures. His knowledge and insights on program synthesis and program repair have been extremely valuable and have greatly enriched this work.

I would like to thank Prof. Lin Zhong for agreeing to be on my thesis committee. His feedback and insights have greatly improved this thesis.

I express my sincere gratitude to Prof. John Mellor-Crummey for providing me guidance on the program repair work presented in this thesis. A major part of the program repair work was done as part of COMP 522 project during which I made significant progress. His constant feedback and suggestions helped in improving the algorithms used for repair.

I would like to thank all members of the Habanero group for all their feedback, support during my PhD. In particular, I would like to thank Shams Imam, Vincent Cave, and Jisheng Zhao for helping me with Habanero Java infrastructure. I am grateful to Raghavan Raman for the collaborative work on data race detection and program repair. I would like to thank Rajkishore Barik for collaborating with me on the scalar replacement work and providing

me an internship opportunity at Intel Labs.

I thank my Masters thesis advisor Prof. Vineeth Paleri for kindling interest in compiler optimizations and programming languages. His guidance and inspiration has helped me through out my graduate student years.

I express my deepest gratitude to my friends, mentors, and colleagues at Hewlett Packard, Sandya Mannarswamy and Vidya Praveen for encouraging me to pursue PhD and helping me with the PhD application. I would also like to express my gratitude to my colleagues at Hewlett Packard, Eddie Gornish and Teresa Johnson for helping me by providing recommendations for my PhD application.

I would like to thank my friends at Rice, Etienne Ackermann, Apoorv Agarwal, Kumud Bhandari, Calvin Charles, Prasanth Chatarasi, Arghya Chatterjee, Tiago Cogumbreiro, Shams Imam, Rahul Kumar, Deepak Majeti, Karthik Murthy, and Sri Raj Paul for all the support and encouragement.

My sincere thanks to my parents, sister, and in-laws who have been very supportive and encouraging all along during my graduate course. I am grateful for the unconditional and unquestioning support from my loving wife Priyanka throughout my work.

Contents

Abstract ii

Acknowledgments iv

List of Illustrations x

List of Tables xii

1 Introduction 1

1.1 Thesis Statement 4

1.2 Contributions 4

1.3 Outline 5

2 Programming and Execution Model 6

2.1 Async-Finish Parallelism 6

2.2 Futures 7

2.3 Computation Graphs 9

2.4 Properties of Async-Finish-Future Programming Model 11

2.4.1 Serial Elision 12

2.4.2 Data Race Freedom and Deadlock Freedom 12

2.4.3 Data Race Freedom and Functional and Structural Determinism . . 15

3 Debugging Parallel Programs with Futures 17

3.1 Problem Statement 19

3.2 Contributions 19

3.3 Determinacy Races in Programs with Futures 20

3.4 Determinacy Race Detection Algorithm 21

3.4.1 Dynamic Task Reachability Graph 22
3.4.2 Shadow Memory 27
3.4.3 Algorithm 27
3.5 Theoretical Results 34
3.6 Experimental Results 41
3.7 Related Work 46
3.8 Summary and Future Wrok 47

4 Test-Driven Repair of Data Races in Task-Parallel Programs 49
4.1 Problem Statement 51
4.2 Contributions 53
4.3 Overview 54
4.4 Data Race Detection 56
4.4.1 Multiple Reader-Writer ESP-Bags 56
4.4.2 Scoped Dynamic Program Structure Tree 57
4.5 Dynamic Finish Placement 60
4.5.1 Dynamic Dependence Graph Construction 61
4.5.2 Algorithm 62
4.5.3 Correctness and Optimality 68
4.6 Static Finish Placement 69
4.6.1 Algorithm 69
4.6.2 Correctness & Conditions for Optimality 71
4.7 Experimental Results 71
4.7.1 Repairing Programs 74
4.7.2 Time for Program Repair 75
4.7.3 Comparison of SRW and MRW ESP-Bags 76
4.7.4 Student Homework Evaluation 76
4.8 Related Work 77

4.9 Summary and Future Work 79

5 Automatic Parallelization via Synthesis of Futures 80

5.1 Problem Statement 82
5.2 Contributions 84
5.3 Overview of Approach 85
5.4 Candidate Future Synthesis 87
5.4.1 Inter-procedural Future Analysis 88
5.4.2 Future Transformation 92
5.4.3 Candidate Future Identification 94
5.4.4 Preserving Data Dependences 97
5.5 Parallelism Benefit Analysis 97
5.5.1 Weighted Computation Graph 98
5.5.2 Classification of Pure Function Calls 101
5.6 Threshold Expression Synthesis 104
5.7 Final Future Synthesis 109
5.8 Experimental Evaluation 110
5.8.1 Experimental Setup 110
5.8.2 Experimental Results 112
5.9 Related Work 116
5.9.1 Futures 116
5.9.2 Parallelism and Performance Profiling 118
5.10 Summary and Future Work 120

6 Putting It Together 122

6.1 Data Race Detection and Manual Repair 122
6.2 Data Race Detection and Automatic Repair 123
6.3 Synthesis of Futures 124
6.4 Synthesis, Data Race Detection and Automatic Repair 125

7 Conclusions and Future Work **126**

7.1 Conclusions 126

7.2 Future Work 127

Bibliography **129**

Illustrations

2.1 Example program with HJ futures 8
2.2 Example program with futures. 10
2.3 Computation graph of the program in Figure 3.1. 10
2.4 Example program that may deadlock 11
2.5 Illustration of Lemma 1. 13

3.1 Example program with futures. 21
3.2 Computation graph of the program in Figure 3.1. 22
3.3 A computation graph with non-tree joins. 25

4.1 Mergesort program with data races. 50
4.2 Quicksort program with data races. 50
4.3 Example async-finish program with execution times 52
4.4 Example finish placements 53
4.5 Async-finish code which demonstrates the scoping issues in finish insertion 53
4.6 High level view of test-driven repair 55
4.7 Async-finish code with multiple data races 57
4.8 Incorrectly synchronized Fibonacci program 59
4.9 Subtree of S-DPST for Fibonacci 60
4.10 A subtree rooted at NS-LCA for Fibonacci 61
4.11 Dependence graph constructed from the subtree in Figure 4.10 62
4.12 Optimal substructure of finish placement 63

4.13 Earliest start time computation 63
4.14 Subtree in Figure 4.10 after inserting finish 67
4.15 Fibonacci program from Figure 4.8 after finish insertion 70
4.16 Performance of repaired programs 72

5.1 Sequential binary tree program 81
5.2 Binary tree program from Figure 5.1 after parallelization 83
5.3 High level view of synthesis 85
5.4 Binary tree program after method purity analysis 87
5.5 Examples of normal flow function for computing $\mathcal{M}$ (may-be-future) 89
5.6 Examples of normal flow function for computing $\mathcal{N}$ 91
5.7 Binary tree program after synthesis of futures 92
5.8 Weighted computation graph for binary bree program 99
5.9 Merge parent transformation on the computation graph 103
5.10 Conditional parallel execution of method call `obj.f(`a_1, a_n`)` 105
5.11 Sequential recursive Fibonacci invocation 112

6.1 Debugging of parallel programs with async, finish, and future 122
6.2 Debugging and repair of parallel programs with async, finish, and future . . 123
6.3 Synthesis of futures in parallel programs 123
6.4 Synthesis of futures with profitability analysis 124
6.5 A workflow which includes synthesis, debugging and repair of task-parallel programs 125

Tables

3.1 Example of dynamic task reachability graph 26
3.2 Example of dynamic task reachability graph 27
3.3 Runtime overhead for determinacy race detection 42

4.1 List of benchmarks evaluated 72
4.2 Time for program repair 73
4.3 Comparison of SRW ESP-Bags and MRW ESP-Bags 74
4.4 Number of data races detected by SRW ESP-Bags and MRW ESP-Bags . . 75

5.1 Example transformation rules based on `future`-analysis results 93
5.2 List of benchmarks evaluated 110
5.3 Result of parallelism benefit analysis 112
5.4 Synthesis statistics 113
5.5 Performance of synthesized programs 114
5.6 Sensitivity of threshold values 115

Chapter 1

Introduction

Today, inexpensive multicore processors are ubiquitous, and the demand for parallel programming is higher than ever. However, despite advances in parallel programming models, it is widely acknowledged that writing correct and efficient parallel programs is a challenging task. Therefore, it is a need of the hour to create tools that aid programmers in parallel software development. In this dissertation, we present algorithms for debugging, repair, and synthesis of parallel programs.

Debugging Parallel Programs. Many parallel programs are intended to be *deterministic*, in that the program computes the same answer when given the same inputs. A program execution contains a *data race* when there are two or more accesses to the same variable, at least one of which is a write, and the accesses are unordered by either synchronization or program order. A program may behave nondeterministically in the presence of data races. That is, different runs of the same program may compute different answers when given the same inputs. Data races are hard to detect and reproduce using traditional debugging techniques such as breakpointing because they may occur in only some of the possible schedules of a program.

Past work on dynamic detection of data races focused either on structured parallelism or unstructured parallelism. For unstructured parallelism, vector clock algorithms serve as a popular foundation for dynamic data race detection, e.g., [1, 2]. A drawback of vector clock-based approaches is that the storage requirements for vector clocks grow directly

proportional to the degree of logical parallelism in the program. For programs with structured parallelism, prior work has identified more scalable solutions, e.g., [3, 4, 5]. These algorithms do not support race detection for futures because the computation graphs in the presence of futures are more general than the computation graphs handled by these approaches. In this dissertation, we present the first known sound and precise algorithm for dynamically detecting races in task-parallel programs with async, finish and future constructs [6].

Repair of Parallel Programs. A major challenge in writing parallel programs is identifying the right amount of synchronization for correctness and performance. Introducing too little synchronization may cause *data races* that are hard to detect, and inserting too much synchronization can reduce parallelism. Most mainstream programmers lack the level of expertise required to develop high performance, data race free parallel programs. With this premise in mind, we propose a novel algorithm for repairing data races in task-parallel programs [7].

Past solutions to the problem of repairing parallel programs have used static-only [8] or dynamic-only [9, 10] approaches, both of which have significant limitations in practice. Static approaches can guarantee soundness in many cases but face severe limitations in precision when analyzing medium or large-scale software with accesses to pointer-based data structures in multiple procedures. Dynamic approaches are more precise than static analyses, but their proposed repairs are limited to a single input and are not reflected back in the original source program. Our approach spans the middle ground between these two classes of approaches. As in dynamic approaches, we execute the program on a concrete test input and determine the set of data races for this input dynamically. However, next, we compute a set of finish placements that rule out these races but also respect the static

scoping rules of the program, and can therefore be inserted back into the static program.

Automatic Parallelization via Synthesis of Futures. Automatic parallelization is a very challenging problem in general. As an example, the current state of the art in automatic parallelization of loops with array accesses (including program dependence graphs [11] and polyhedral frameworks [12]) was developed over four decades with many restrictions along the way with respect to array subscript expressions and procedure calls in loops. The seminal papers in this field (e.g. [13]) included results for simple loop nests, and it took many years for the original ideas to be refined and applied to real-world programs. Our work is the first to address the problem of automatic parallelization by generating futures [14], which is different from past work on automatic loop/statement-level parallelization using control and data dependences, since future references can be copied without waiting for the result to be computed.

Futures are traditionally used for enabling functional style parallelism, and therefore, are a natural fit for parallelizing pure method calls. They also have the advantage that references to future objects can be freely copied without waiting for the future tasks to have completed, thereby exposing more parallelism than in imperative-style task parallel constructs. However, there remain significant challenges when using futures to manually parallelize programs with pure method calls. First, it is necessary to deal with potentially conflicting write operations after the pure method call. Second, in strongly typed languages like Java, C#, and C++, it is necessary to introduce type declarations for propagating future references, as well as dereference (*get*) operations when the value returned by a future task needs to be accessed. Finally, it is also important to identify what subset of available parallelism is profitable for a given system. We introduce a novel approach for using futures to automatically and efficiently parallelize the execution of pure method calls.

1.1 Thesis Statement

Current approaches for debugging, repair, and synthesis have known limitations in functionality, performance or precision when applied to general task-parallel programs with non-strict computation graphs. Our thesis is that debugging, repair, and synthesis can be performed efficiently and precisely for task-parallel programs with async, finish and future constructs, using a combination of static and dynamic techniques.

1.2 Contributions

This dissertation makes the following contributions:

- The first known sound and precise on-the-fly *debugging* algorithm for detecting races in programs containing async, finish, and future parallel constructs. We show that the algorithm can detect determinacy races by effectively analyzing all possible executions for a given input.

- A test-driven approach for *repairing* data races in parallel programs containing async and finish constructs. Our approach takes as input an under-synchronized parallel program with async and finish constructs and a test input and inserts finish constructs to repair all data races for that particular test input while maximizing parallelism in the resulting race-free program.

- A novel approach for *synthesizing* futures for automatically parallelizing the execution of pure method calls. Our approach synthesizes object clones when needed for handling anti dependences, generates precise type information for future objects, automatically determines the profitability of executing a method call as future task and synthesizes threshold expressions which determine dynamically whether a specific method call should be executed sequentially or in parallel.

- An implementation of these algorithms as analyses and transformations on the Java bytecode intermediate representation for Habanero-Java programs, and evaluation on a range of benchmark programs. The evaluation establishes the effectiveness of our approach with respect to dynamic data race detection overhead, compile-time overhead, and precision and performance of the repaired and synthesized code.

1.3 Outline

The rest of this thesis is organized as follows:

- Chapter 2 summarizes the task-parallel programming model targeted by this thesis.
- Chapter 3 discusses how we address the problem of debugging parallel programs with dynamic data race detection for async, finish, and future constructs.
- Chapter 4 presents our approach for repairing data races in parallel programs by inserting finish constructs.
- Chapter 5 presents our approach for synthesizing futures in parallel programs.
- Chapter 6 describes how the techniques in Chapters 3-5 can be put together for different workflows using our toolchain
- Chapter 7 summarizes our conclusions and future work.

Chapter 2

Programming and Execution Model

Our work addresses task-parallel programming models that can support combinations of functional-style futures and imperative-style tasks; examples include the X10 [15], Habanero Java [16], Chapel [17], and C++11 languages. Section 2.1 presents the async-finish task-parallel programming model and Section 2.2 presents the future construct. Section 2.3 summarizes the computation graph abstraction which models the execution of a parallel program as a partial order, and Section 2.4 discusses the conditions for determinism and deadlock-freedom for the async-finish-future task-parallel programming model.

2.1 Async-Finish Parallelism

We will use X10 and Habanero Java's finish and async notation for task parallelism in this thesis, though our algorithms are applicable to other task-parallel constructs as well. In this notation, the statement "`async { S }`" causes the parent task to create a new child task to execute `S` asynchronously (i.e., before, after, or in parallel) with the remainder of the parent task following the async statement. The statement "`finish { S }`" causes the parent task to execute `S` and then wait for the completion of all asynchronous tasks created within `S`. Each dynamic instance T_A of an async task has a unique dynamic *Immediately Enclosing Finish (IEF)* instance F of a finish statement during program execution, where F is the innermost finish containing T_A. There is an implicit finish scope surrounding the body of `main()`, so program execution will end only after all async tasks have completed.

2.2 Futures

A future [18] (or promise [19]) refers to an object that acts as a proxy for a result that may initially be unknown, because the computation of its value may still be in progress as a parallel task. In the notation used in this thesis, the statement, "`future<T> f = async<T> Expr;`" creates a new child task to evaluate `Expr` asynchronously, where `T` is the type of the expression `Expr`. It also assigns to `f` a reference to a handle (future object) for the return value from `Expr`. The operation `f.get()` can be performed to obtain the result of the future task. If the future task has not completed as yet, the task performing the `f.get()` operation blocks until the result of `Expr` becomes available. Futures are traditionally used for enabling functional-style parallelism and are guaranteed not to exhibit data races in their return values. However, imperative programming languages allow future tasks to contain side effects in the task bodies. These side effects on shared memory locations may cause determinacy races if the program has insufficient synchronization. Another use case for futures is that of future tasks with void return values, where the `get()` operation is used purely for point-to-point synchronization and the tasks only communicate values through side effects, just as with async and finish constructs.

Comparison with spawn-sync and async-finish. In the spawn-sync programming model, a task is created using the spawn construct and a task can wait for its child tasks to complete execution using the sync construct. At a sync construct, a task waits for all the previously created children to complete its execution. There is an implicit sync at the end of every function. The async construct is similar to the spawn construct and the finish construct waits for all descendant tasks in the scope of the finish to complete execution. In both spawn-sync and async-finish programming models, a join operation can be performed only once on a task (by the parent task in spawn-sync and by the ancestor task containing the immediately enclosing finish in async-finish). The class of computations generated by

spawn-sync constructs is said to be *fully strict* [20], and the class of computations generated by async-finish constructs is called *terminally strict* [21].

```
1  // Main task
2  Stmt1;
3  future<T> A = async<T> { ... }; // T_A
4  Stmt2;
5  future<T> B = async<T>{ Stmt3;A.get();Stmt4;}; // T_B
6  Stmt5;
7  future<T> C = async<T>{ Stmt6 ; A.get();
8      Stmt7; B.get();}; // T_C
9  Stmt8;
10 A.get();
11 Stmt9;
12 C.get();
13 Stmt10;
```

Figure 2.1 : Example program with HJ futures. T_A, T_B, and T_C are future tasks created by the main program

The introduction of future as a parallel construct increases the possible synchronization patterns. Task X can wait for a previously created task Y if X has a reference to Y by performing the get() operation. Moreover, this join operation on task Y can be performed by multiple tasks. As an example, consider the program in Figure 2.1, where the main program creates three future tasks T_A, T_B and T_C. There are three join operations on task T_A performed by sibling tasks T_B, T_C and the parent task. Here Stmt3, Stmt6 and Stmt8 may execute in parallel with task T_A, where Stmt4, Stmt7 and Stmt9 can execute only after the completion of task T_A. Synchronization using get() can lead to transitive dependence between tasks. For example, although the main task in Figure 2.1 did not perform an explicit join on task T_B, there is a transitive join dependence from T_B to the main task, because task T_C performed a get operation on task T_B due to which Stmt10 can execute only after tasks T_A, T_B and T_C complete their execution.

2.3 Computation Graphs

A *computation graph* [20] for a dynamic execution of a task-parallel program is a directed acyclic graph, in which each node corresponds to a *step* which is defined as follows:

Definition 1. A step is a sequence of instruction instances contained in a task such that no instance in the sequence includes the start or end of an async, finish or a get operation. □

The edges in a computation graph represent different forms of happens-before relationships. For the task-parallel constructs discussed in this thesis (async, finish, future), there are three different types of edges:

1. **Continue Edges** capture the sequencing of steps within a task. All steps in a task are connected by continue edges.

2. **Spawn Edges** represent the parent-child relationship among tasks. When task A creates task B, a spawn edge is inserted from the step that ends with the async in task A to the step that starts task B.

3. **Join Edges** represent synchronization among tasks. When task A performs a get on future B, a join edge (also referred to as a "future join edge") is inserted from the last step of B to the step in task A that immediately follows the `get()` operation. In addition, "finish join edges" are also inserted from the last step of every task in a finish scope to the step in the ancestor task immediately following the Immediately Enclosing Finish (IEF).

Definition 2. A step u is said to precede step v, denoted as $u \prec v$, if there exists a path from u to v in the computation graph.

```
1  S1 ;
2  future<T> A = async<T> { S2; // T_A
3                  future<T> B = async<T> { S3; }; // T_B
4                  S4;  B.get();  S5; };
5  S6 ;
6  future<T> C = async<T>{ S7; A.get(); S8;}; // T_C
7  S9 ;
8  future<T> D = async<T>{ S10; C.get(); S11;}; // T_D
9  D.get();
10 S12 ;
```

Figure 2.2 : Example program with futures. T_A,T_B,T_C and T_D are future tasks. S1-S12 are steps (sequential computations with no parallel constructs) in the program.

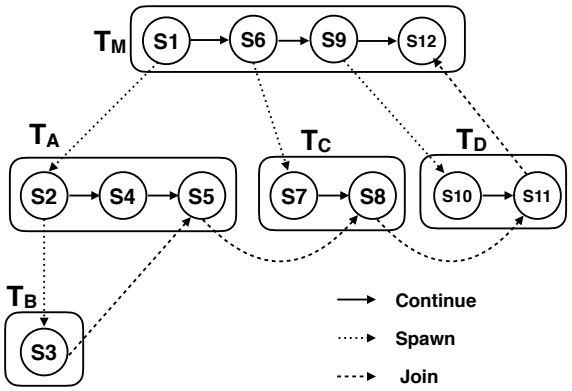

Figure 2.3 : Computation graph of the program in Figure 3.1. The circles represent the steps in the program. The rectangles represents tasks. T_M is the main task and T_A, T_B, T_C and T_D are the tasks created during the execution of the program. The edges between the steps in the same task are continue edges. The edges between steps in different tasks are spawn edges or join edges

This precedence relation is a partial order, and is also referred to as the "happens-before" relation in past work [22]. We use the notation Task(u) = T to represent that step node u belongs to task T, and $u \not\prec v$ to denote the fact that there is no path from step u to step v in the computation graph. Two distinct steps, u and v may execute in parallel, denoted as $u \parallel v$, iff $u \not\prec v$ and $v \not\prec u$.

As an example, consider the program in Figure 2.2 which creates four future tasks: T_A, T_B, T_C, and T_D. S1-S12 represent the steps in the program. The computation graph of the program is shown in Figure 2.3. Here S2 || S10 because there is no directed path from S2 to S10, or from S10 to S2, in the computation graph, and S2 $\prec$ S12 since there is a directed path from S2 to S12.

2.4 Properties of Async-Finish-Future Programming Model

In this section, we discuss sufficient conditions for a program containing async, finish and future constructs to be deterministic and deadlock-free. Figure 2.4 shows an example of a program containing two future tasks which may deadlock in certain executions (or encounter a NullPointerException in other executions). Because of the data race, this program can exhibit multiple possible computation graphs when executed with the same input, some of which result in deadlock due to a cyclic dependence between tasks. Deadlock can occur in the case when future tasks F1 and F2 wait for each other indefinitely via the `a.get()` and `b.get()` operations.

```
1 future<T> a = null, b = null;
2 async { a = async<T> /*F1*/ { b.get(); ...}; }
3 async { b = async<T> /*F2*/ { a.get(); ...}; }
```

Figure 2.4 : Example program with shared locations a and b, and future tasks F1 and F2 that may deadlock (or encounter a NullPointerException). F1 and F2 are themselves contained within async tasks. The program has a data race, because the write of shared location `a` in line 2 may execute in parallel with the read of `a` in line 3 (and similarly for shared location `b`).

In Section 2.4.1, we describe the ***serial elision*** for a program containing async, finish and future constructs, and in Sections 2.4.2-2.4.3 we show that a program with no data races is guaranteed to be deterministic, deadlock-free and have the same semantics as the serial

elision of the program. As a result, and following past conventions (e.g., [4]), we will refer to data races in programs containing async, finish and future constructs as *determinacy races*.

2.4.1 Serial Elision

Similar to spawn-sync and async-finish programs, programs with futures have a well defined ***serial elision version***, which is the serial program obtained after removing all the parallel constructs. To be specific, the serial elision of a program is obtained by removing all async and finish constructs, and by replacing "`future<T> f`" by "`T f`", "`async<T> Expr;`" by "`Expr;`" and `f.get()` by `f`. The presence of a well-defined serial elision version for any async-finish-future parallel program ensures that it is always possible to execute that program sequentially in depth-first order on a single processor. Our race detection algorithm makes use of this property to identify memory accesses that can logically execute in parallel.

2.4.2 Data Race Freedom and Deadlock Freedom

In this section, we show that a program with async, finish, and future constructs can deadlock only if there are data races on references to futures. Let us first take a closer look at a future task creation and a get() operation on a future task. When the parent task creates a task using the statement "`A = async<T> Expr;`", it performs two actions in sequence: 1) it creates the child task which is represented in the computation graph as a spawn edge and 2) updates `A` with the reference to the child task. Any task performing a get() operation on the child task must have the reference to the task. The following lemma (illustrated in Figure 2.5) shows that if a get operation on a future task executes in parallel with its creation, the program has a data race on a reference to future object.

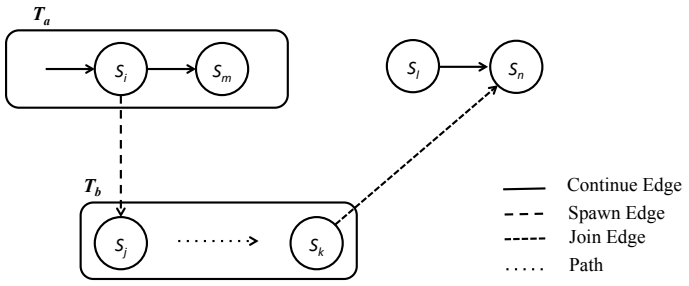

Figure 2.5 : Illustration of Lemma 1. (s_i, s_j) is the spawn edge for task T_b. (s_k, s_n) is a get() operation on T_b. If $s_m \neq s_l$ and $s_m \not\prec s_l$, the input program has a data race on a reference to future object.

Lemma 1. Let (s_i, s_j) be a spawn edge in computation graph G, where Task$(s_i) = T_A$ and Task$(s_j) = T_B$ (which must be distinct from T_A), such that T_B is a future task. Let (s_i, s_m) be a continue edge in task T_A. Let s_n be a step which is executed after a join operation on T_B and let (s_l, s_n) be a continue edge in G. Then, if $s_m \neq s_l$ and $s_m \not\prec s_l$, the input program has a data race on a reference to future object.

Proof. The edge (s_i, s_j) corresponds to the creation of task T_B. The continue edge (s_i, s_m) represents the sequencing of the two steps s_i and s_m, where s_m is the first step in G where the creation of task T_B is guaranteed to be complete. The get operation on T_B occurs after the execution of s_l and before the execution of s_n. Therefore, a reference to task T_B must be available at s_l.

Let us assume that $s_m \parallel s_l$. Since the first step in G at which a reference to T_B is available is s_m, the only possible way for s_l to have a reference to T_B is through a data race (e.g. s_m writing to a shared memory location and s_l reading from the same memory location in parallel).

Therefore, if $s_m \not\prec s_l$, the program has a data race on a reference to a future object. □

Next we establish a property for all edges in a race free computation graph, which in

turn proves that a race free program is deadlock free.

Lemma 2. Suppose that two steps s_1 and s_2 execute in order in a serial, depth-first execution of a program P, and suppose that there exists an edge (s_2, s_1) in a computation graph G of P. Then, the program has a data race on a reference to a future object.

Proof. A continue edge cannot exist from s_2 to s_1 because a continue edge represents the sequencing of steps within a single task. If s_1 and s_2 belong to the same task and s_1 executes before s_2 in the serial depth-first execution, then edge (s_2, s_1) cannot be a continue edge.

Now, suppose there exists a spawn edge from s_2 to s_1. A spawn edge represents a control dependence and therefore $s_2 \prec s_1$. This implies that s_2 must execute before s_1 during the serial depth-first execution. This is a contradiction of our initial assumption that s_1 executes before s_2 during the serial depth-first execution.

Let Task(s_1) = T_a and Task(s_2) = T_b and let us consider if it's possible to have a join edge from s_2 to s_1. There are four possible scenarios:

1. T_a is an ancestor of T_b: Here $s_1 \prec s_2$ and therefore a join edge cannot exist from s_2 to s_1, because the task creation of T_b happened after the execution of s_1.

2. T_b is an ancestor of T_a: Here the spawn of T_a precedes s_2 because otherwise s_2 will execute before s_1 during a depth-first execution. A join edge can exist from s_2 to s_1 only in the presence of data race on a reference to a future object according to Lemma 1.

3. T_a completes execution before T_b during the depth-first execution: In this case s_1 either precedes the spawn of T_b or s_1 can execute in parallel with the spawn of T_b. If s_1 precedes the spawn of T_b, a reference to T_b is not available during the execution of s_1. If s_1 can execute in parallel with the spawn of T_b, then according to Lemma 1 the program execution has a data race on a reference to a future object.

4. T_b completes execution before T_a during the depth-first execution: In this case the creation of T_b must happen in parallel with s_1 and therefore, according to Lemma 1 the program execution has a data race on a reference to a future object. □

Lemma 2 proves that if the program is free of races, the program has no cyclic dependences and is free of deadlocks.

2.4.3 Data Race Freedom and Functional and Structural Determinism

We say that a parallel program is *functionally deterministic* if it always computes the same answer, when given the same inputs. By default, any sequential computation is expected to be deterministic with respect to its inputs; if the computation interacts with the environment (e.g., a GUI event such as a mouse click, or a system call like System.nanoTime()) then the values returned by the environment are also considered to be inputs to the computation. Further, we refer to a program as *structurally deterministic* if it always computes the same computation graph, when given the same inputs. Finally, following past work [23, 24], we say that a program is *determinate* if it is both functionally and structurally deterministic.

The presence of data races may lead to functional and/or structural nondeterminism because a parallel program with data races can exhibit different behaviors for the same input, depending on the relative scheduling and timing of memory accesses involved in a data race. In general, the absence of data races in a parallel program is not sufficient to guarantee determinism, e.g., it is possible to write data-race-free nondeterministic parallel programs using locks. However, the parallel constructs that are the focus of this thesis (async, finish and future) were carefully selected to ensure the following *Determinism Property*:

> If a parallel program is written using only async, finish and future constructs, *and is guaranteed to never exhibit a data race*, then it must be determinate, i.e., both functionally and structurally deterministic.

Note that the determinism property states that all data-race-free programs written using async, finish and future constructs are guaranteed to be determinate, but it does not imply that all racy programs are non-determinate. For instance, a program with parallel writes of the same value to a common memory location is racy, yet determinate.

The execution of a race-free program containing async, finish and future constructs is also ***dag-consistent*** [25] because 1) the computation graph does not contain any cycles and 2) a read can "see" a write only if there is some serial execution order of the dag in which the read sees the write, thereby ensuring that the write seen by a read is always the same for all executions with the same input. Such a program execution is guaranteed to be deterministic. Therefore, verifying race freedom of parallel programs is an important step in proving the correctness of those programs.

Chapter 3

Debugging Parallel Programs with Futures

Current dynamic race detection algorithms for task parallelism are limited to parallel constructs in which a task may synchronize with the parent task [4, 26], ancestor task [5, 27] or with the immediate left sibling [28]. However, current parallel programming models include parallel constructs that support more general synchronization patterns. For example, the OpenMP depends clause allows tasks to wait on previously spawned sibling tasks and the future construct in X10, HJ, C# and C++11 enables a task to wait on any previously created task to which the waiter task has a reference. Algorithms based on vector clocks [1, 2] are impractical for these constructs because either the vector clocks have to be allocated with a size proportional to the maximum number of simultaneously live tasks (which can be unboundedly large) or precision has to be sacrificed by assigning one clock per processor or worker thread, thereby missing potential data races when two tasks execute on the same worker.

The approaches in [4, 26, 5, 27] focus on a structured task-parallel model, in which tasks communicate through side effects on shared variables. In contrast, our work focuses on enabling the use of futures for functional-style parallelism, while also allowing futures to co-exist with imperative async-finish parallelism. The addition of point-to-point synchronization for futures makes the race detection more challenging than for async-finish task parallelism since the computation graphs that can be generated using futures are more general than those that can be generated by fork-join parallel constructs such as async-finish constructs in X10 [15] and Habanero-Java [16], spawn-sync constructs in Cilk [29],

and task-taskwait constructs in OpenMP [30].

Existing algorithms for detecting determinacy races for dynamic task parallelism, do not support race detection for futures. For instance, Cilk data race detectors [4, 26] handle only spawn-sync constructs where the computation graph is a Series-Parallel (SP) dag. Although the computation graphs for async-finish parallelism [5, 27] are more general than SP-dags, whether two instructions may logically execute in parallel can still be determined efficiently by a lookup of the lowest common ancestor of the instructions in the dynamic program structure tree [5, 27]. Dimitrov et.al [28] extended the SP-bags algorithm to work for 2D-lattices by formulating the race detection problem as the suprema (least upper bound) computation on the computation graph. The computation graphs in the presence of futures may not have any of the structures discussed above, and therefore, the past approaches are not directly applicable to parallel programs with futures. However, parallel programs written with futures enjoy the property that data race freedom implies determinacy, i.e., if a parallel program is written using only async, finish, and future constructs, and is known to not exhibit a data race, then it must be determinate [23, 24]. (Determinacy includes *functional determinism* and *structural determinism* — repeated executions of a determinate parallel program with the same input are guaranteed to always produce the same output and the same computation graph.). Thus, a data race detector for programs with async, finish, and future constructs, can be used as a determinacy checker for these programs.

The remainder of the chapter is organized as follows. Section 3.1 states the problem statement, and Section 3.2 presents our contributions. Section 3.3 defines determinacy races for our programming model. Section 3.4 presents the algorithm for determinacy race detection for parallel programs with futures, and Section 3.5 discusses the complexity and correctness of our algorithm. Section 3.6 discusses the implementation and experimental

results for our race detection algorithm. Section 3.7 discusses related work, and Section 3.8 contains a summary of the chapter.

3.1 Problem Statement

The problem statement for debugging parallel programs containing async, finish, and future constructs is given below:

Problem 1. (Debugging) Given a parallel program P with async, finish, and future constructs, and test input, ψ, either determine a memory location in the program that are subject to data races when the program is run on test input, ψ, or else certify that the program is race-free when run on ψ. The data race detector must satisfy the following properties:

1. it must only report input, ψ, as race-free if no race is possible for any schedule of P for that input, and

2. it must not report any false positives or false negatives. □

3.2 Contributions

The main contributions of this work are:

1. The first known sound and precise on-the-fly algorithm for detecting races in programs containing async, finish and future parallel constructs. Instead of using brute force approaches such as building transitive closure, our algorithm relies on a novel data structure called the ***dynamic task reachability graph*** to efficiently detect races in the input program.

2. Correctness and complexity analysis of the race detection algorithm. We show that the algorithm can be used to analyze all possible executions of the program on a

given input and detect determinacy races or certify that the program is free of races. We also present a complexity analysis of the algorithm and show that the algorithm performs similarly to other efficient algorithms such as SP-bags algorithm when using only structured parallel constructs such as async-finish and spawn-sync, or using futures to express the same dependences. In the presence of future operations, the complexity of the algorithm increases only by a factor determined by the number of future operations, and the extent to which they go beyond strict computation graphs.

3. An implementation and evaluation of the algorithm on programs with structured parallelism and point-to-point synchronization. We implemented the algorithm in the Habanero Java compiler and runtime system and evaluated it on a suite of benchmarks containing async, finish and future constructs. The experiments show that the algorithm performs similarly to SP-bags in the presence of structured synchronization and degrades gracefully in the presence of point-to-point synchronization.

3.3 Determinacy Races in Programs with Futures

In this section, we formalize the definition of data races in programs containing async, finish, and future constructs as a preamble to defining determinacy races. Our definition uses the notion of a *computation graph* for a dynamic execution of a parallel program presented in Section 2.3.

Definition 3. A data race may occur between steps u and v, iff $u \parallel v$ and both u and v include accesses to a common memory location, at least one of which is a write.

Next, we classify the join edges in computation graphs with async, finish, and future as follows: A join edge from task B to task A is referred to as *tree join* if A is an ancestor of

B; otherwise, it is referred to as a *non-tree join*. Note that all finish join edges must be tree joins, and some future join edges may be tree edges and some may be non tree edges.

As an example, consider the program in Figure 3.1 which creates four future tasks: T_A, T_B, T_C, and T_D. S1-S12 represent the steps in the program. The computation graph of the program is shown in Figure 3.2. Here S2 $\parallel$ S10 because there is no directed path from S2 to S10, or from S10 to S2, in the computation graph, and S2 $\prec$ S12 since there is a directed path from S2 to S12. The join edge from S3 to S5 is a tree join since T_A is an ancestor of T_B. The edge from S5 to S8 is a non-tree join since T_C is not an ancestor T_A.

```
1  S1 ;
2  future<T> A = async<T> { S2; // T_A
3                  future<T> B = async<T> { S3; }; // T_B
4                  S4;  B.get();  S5; };
5  S6 ;
6  future<T> C = async<T>{ S7; A.get(); S8;}; // T_C
7  S9 ;
8  future<T> D = async<T>{ S10; C.get(); S11;}; // T_D
9  D.get();
10 S12 ;
```

Figure 3.1 : Example program with futures. T_A,T_B,T_C and T_D are future tasks. S1-S12 are steps (sequential computations with no parallel constructs) in the program.

3.4 Determinacy Race Detection Algorithm

In this section, we present our algorithm for detecting determinacy races in programs with async, finish and future as parallel constructs. A dynamic determinacy race detector needs to provide mechanisms that answers two questions: for any pair of memory accesses, at least one of which is a write, 1) can the two accesses logically execute in parallel?, and 2) do they access the same memory location? To answer the first question, we introduce a program representation referred to as *dynamic task reachability graph* which is presented

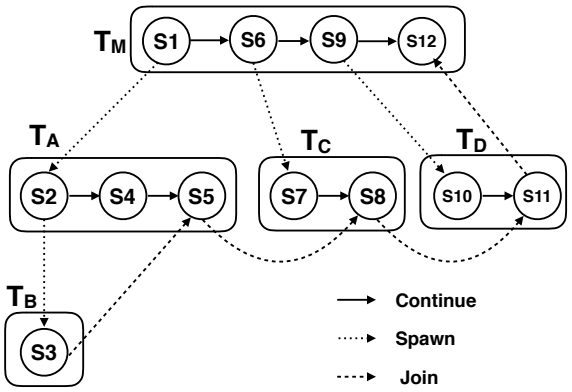

Figure 3.2 : Computation graph of the program in Figure 3.1. The circles represent the steps in the program. The rectangles represents tasks. T_M is the main task and T_A, T_B, T_C and T_D are the tasks created during the execution of the program. The edges between the steps in the same task are continue edges. The edges between steps in different tasks are spawn edges or join edges

in Section 3.4.1. Similar to most race detectors, we use a shadow memory mechanism (presented in Section 3.4.2) to answer the second question. Section 3.4.3 presents our overall determinacy race detection algorithm.

3.4.1 Dynamic Task Reachability Graph

Since storing the entire computation graph of the program execution is usually intractable due to memory limitations (akin to storing a complete dynamic trace of a program), we introduce a more compact representation that still retains sufficient information to precisely answer all reachability queries during race detection. Our program representation, referred to as a *dynamic task reachability graph*, represents reachability information at the task-level instead of the step-level. The representation assumes that the input program is executed serially in depth-first order, and leverages the following three ideas for encoding reachability information between steps in the computation graph of the input program:

Disjoint set representation of tree joins The reachability information between tasks which are connected by tree join edges is represented using a disjoint set data structure. Two tasks A and B are in the same set if and only if B is a descendant of A and there is a path in the computation graph from B to A which includes only tree-join edges and continue edges. Similar to the SP-bags algorithm, our algorithm uses the fast disjoint-set data structure [31, Chapter 22], which maintains a dynamic collection of disjoint sets Σ and provides three operations:

1. MakeSet(x) which creates a new set that contains x and adds it to Σ
2. Union(X, Y) which performs a set union of X and Y, adds the resulting set to Σ and destroys set X and Y
3. FindSet(x) which returns the set $X \in \Sigma$ such that $x \in X$.

Any m of these three operations on n sets takes a total of $O(m\alpha(m, n))$ time [32]. Here α is functional inverse of Ackermann's function which, for all practical purposes is bounded above by 4.

Interval encoding of spawn tree In order to efficiently store and answer reachability information from a task to its descendants, we use a labeling scheme [33], in which each task is assigned a label according to preorder and postorder numbering schemes. The values are assigned according to the order in which the tasks are visited during a depth-first-traversal of the *spawn tree*, where the nodes in the spawn tree correspond to tasks and edges represent the parent-child spawn relationship. Using this scheme, the ancestor-descendant relationship queries between task pairs can be answered by checking if the interval of one task subsumes the interval of the other task. For example, if $[x.pre, x.post]$ is the interval associated with task x and $[y.pre, y.post]$ is the interval associated with task y, then x is an ancestor of y if and only if $x.pre \leq y.pre$ and $y.post \leq x.post$. When task A performs a join

operation on a descendant task B, the disjoint sets of A and B are merged together and the new set will have the label originally associated with A. Although, a label is assigned to every task when it is spawned, the labels are associated with each disjoint set in general. Compared to past work [33] which used labeling schemes on static trees, the tree is dynamic in our approach since race detection is performed on-the-fly. This requires a more general labeling scheme, where a temporary label is assigned when a task is spawned and the label is updated when the task returns to its parent.

Immediate predecessors+significant ancestor representation of non-tree joins The non-tree joins in the computation graph are represented in the dynamic task reachability graph as follows:

- *immediate predecessors*: For each non-tree join from task A to task B, B stores A in its set of predecessors.

- *lowest significant ancestor*: We define the *significant ancestors* of task A as the set of ancestors of A in the spawn tree that have performed at least one non-tree join operation. For each task, we store only the lowest significant ancestor.

Definition 4. A *dynamic task reachability graph* of a computation graph G is a 5-tuple $R = (N, D, L, P, A)$, where

- N is the set of vertices, where each vertex represents a dynamic task instance.

- $D = \{D_i\}_{i=1}^{n}$ is a partitioning of the vertices in N into disjoint sets. $\bigcup_{i=1}^{n} D_i = N$. Each partition consists of tasks which are connected by tree-join edges.

- $L : N \to \mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}$ is a map from vertices to their labels, where each label consists of the preorder and postorder value of the vertex in the spawn tree. A label is also

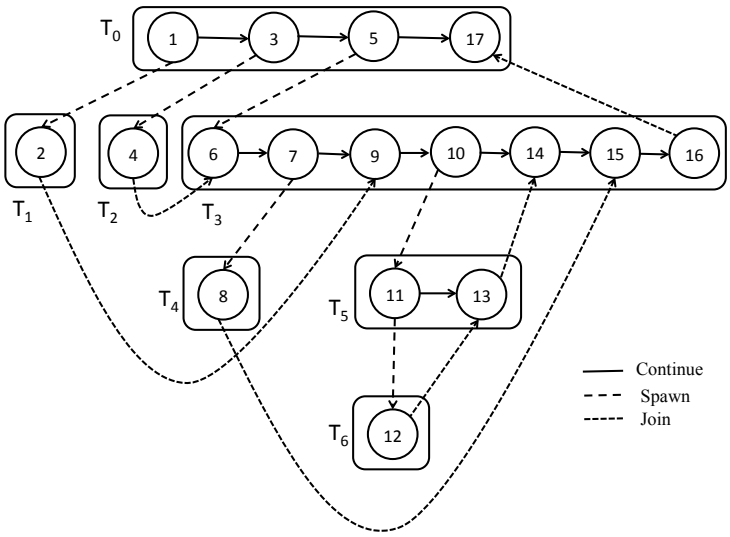

Figure 3.3 : A computation graph with non-tree joins. The join edges (2,9) and (4,6) are non-tree joins because T_1 and T_2 are not descendants of T_3.

associated with each disjoint set $D_i \in D$, where the label for D_i is same as the label of u, where $u \in D_i$ and u is the node in D_i that is closest to the root of the spawn tree.

- $P : N \rightarrow 2^N$ represents the set of non-tree edges $P(u) = \{v_1, .., v_k\}$ if and only if there are non-tree join edges from tasks $v_1..v_k$ to u.

- $A : N \rightarrow N$ represents the lowest ancestor with at least one incoming non-tree edge. $A(u) = v$, if and only if $w_1, w_2..w_k..w_m$ (where $r = w_1$, $v = w_k$ and $u = w_m$) is the path consisting of spawn edges from the root r of G to u, and $P(w_j) = \emptyset, \forall j$ such that $k + 1 \leq j \leq m - 1$ and $P(v) \neq \emptyset$. v is referred to as the *lowest significant ancestor (LSA)* of u.

All steps in a task T are represented using a single node in the dynamic task reachability graph. The continue edges in the computation graph are not present in the dynamic task

reachability graph. For every spawn edges (x, y) in the computation graph there is a corresponding edge (u, v) in the task graph, where Task(x) = u and Task(y) = v. A join edge from a task to its ancestor is not explicitly present in the task graph. When a task B joins with its ancestor A, the disjoint sets corresponding to A and B are merged together. The rest of the joins performed by task t are referred to as *non-tree joins* and are represented as $P(t)$. Section 3.4.3 presents our algorithm for race detection, which uses the dynamic task reachability graph to answer if two steps may execute in parallel.

Disjoint Set	**Task**	**L (Label)**	**P (NT)**	**A (LSA)**
0	T_0	[0, MAXINT]	( )	-
1	T_1	[1, 2]	( )	-
2	T_2	[3, 4]	( )	-
3	T_3	[5, MAXINT-1]	{T_1,T_2}	-
4	T_4	[6, 7]	( )	T_3
5	T_5	[8, MAXINT-2]	( )	T_3
6	T_6	[9, MAXINT-3]	( )	T_3

Table 3.1 : The dynamic task reachability graph for the computation graph in Figure 3.3 after execution of step 11. Task T_3 performed join operations on T_2 and T_1. Therefore $P(T_3) = \{T_1, T_2\}$. The least significant ancestor of T_4, T_5 and T_6 is T_3 because T_3 is their lowest ancestor which performed a non-tree join.

Table 3.1 shows the dynamic task reachability graph for the computation graph in Figure 3.3 after the execution of step 11. Here the postorder values assigned to T_0, T_3, T_5 and T_6 are temporary values (See Section 3.4.3). All tasks are in a separate disjoint sets, because no tree joins have been performed yet. Table 3.2 shows the dynamic task reachability graph for the computation graph in Figure 3.3 after the execution of step 17.

Disjoint Set	Task	L (Label)	P (NT)	A (LSA)
0	T_0 T_3 T_4 T_5 T_6	[0, 13] [5, 12] [6, 7] [8, 11] [9, 10]	$\{T_1, T_2\}$	-
1	T_1	[1, 2]	( )	-
2	T_2	[3, 4]	( )	-

Table 3.2 : The dynamic task reachability graph for the computation graph in Figure 3.3 after execution of step 17. T_0, T_3, T_4, T_5 and T_6 are all in the same disjoint set because they are connected by tree join edges.

3.4.2 Shadow Memory

As in past work [5, 27], our algorithm maintains a shadow memory M_s for every shared memory location M. M_s contains the following fields

- w, a reference to a task that wrote to M. $M_s.w$ is initialized to *null* and is updated at every write to M. It refers to the task that last wrote to M.
- r, a set of references to tasks that read M. $M_s.r$ is initialized to $\emptyset$ and is updated at reads of M. It contains references to all future tasks that read M in parallel, since the last write to M. It also contains a reference to one non-future (async) task which read M since the last write to M.

3.4.3 Algorithm

The overall determinacy race detection algorithm is given in Algorithms 1-10. As the input program executes in serial, depth-first order the race detection algorithm performs additional operations whenever one of the following actions occurs: task creation, task return, begin-finish, end-finish, `get()` operation, shared memory read and shared memory

write. The race detector stores the following information associated with every disjoint set of tasks.

- *pre* and *post* together form the interval label assigned to the disjoint set.
- *nt* is the set of incoming non-tree edges.
- *parent* refers to the parent task.
- *lsa* represents the least significant ancestor.

Next, we describe the actions performed by our race detector:

Algorithm 1 Initialization

Require: Main task M

1: $dfid \leftarrow 0$
2: $tmpid \leftarrow MAXINT$
3: $S_M \leftarrow$ MAKE-SET(M)
4: $S_M.pre \leftarrow dfid$
5: $dfid \leftarrow dfid + 1$
6: $S_M.post \leftarrow tmpid$
7: $tmpid \leftarrow tmpid - 1$
8: $S_M.parent \leftarrow null$
9: $S_M.lsa \leftarrow null$

Initialization: Algorithm 1 shows the initialization performed by our race detector when the main task M is created. The set S_M is initialized to contain task M. It assigns [0, *MAXINT*] as the interval label for the main task. Since the postorder value of a node is known only after the full tree has unfolded, we assign a temporary postorder value *MAXINT* (the largest integer value). The *parent* and *lsa* fields are initialized to *null*.

Task Creation: Algorithm 2 shows the actions performed by our race detector during task creation. Whenever a task P spawns a new task C, C is assigned the preorder value and

a temporary postorder value. Our algorithm assigns temporary postorder values starting at the largest integer value (*MAXINT*) in decreasing order. This assignment scheme maintains the interval label property, where the label of an ancestor subsumes the labels of descendants. The set S_C is initialized to contain task C. The least significant ancestor for task C is initialized at task creation time based on whether task P has performed any non-tree joins.

Algorithm 2 Task creation

Require: Parent task P, Child task C

1: $S_C \leftarrow$ MAKE-SET(C)
2: $S_C.pre \leftarrow dfid$
3: $dfid \leftarrow dfid + 1$
4: $S_C.post \leftarrow tmpid$
5: $tmpid \leftarrow tmpid - 1$
6: $S_C.parent \leftarrow S_P$
7: **if** $S_P.nt = \{\}$ **then**
8: $\quad S_C.lsa \leftarrow S_P.lsa$
9: **else**
10: $\quad S_C.lsa \leftarrow S_P$
11: **end if**

Task Termination: When task C terminates, the postorder value of C is updated with the final value. This is shown in Algorithm 3.

Algorithm 3 Task termination

Require: Terminating task C

1: $S_C.post \leftarrow dfid$
2: $dfid \leftarrow dfid + 1$
3: $tmpid \leftarrow tmpid + 1$

Get Operation: Algorithm 4 shows the actions performed by the race detector at a `get()` operation. When task A performs a `get()` operation on task B, there are two possible cases: 1) A is an ancestor of B and there are join edges from all tasks which are descendants of

A and ancestors of B to A. In this case, the algorithm performs a union of the disjoint sets S_A and S_B by invoking the *Merge* function given in Algorithm 5, and 2) there is a non-tree join edge from B to A. In this case, B is added to the sequence of non-tree predecessors of A.

Algorithm 4 Get operation

Require: Tasks A, B such that A performs B.`get`()

```
1: if FIND-SET(A) =
2:    FIND-SET(B.parent) then
3:      MERGE(S_A, S_B)
4: else
5:      S_A.nt ← S_A.nt ∪ {B}
6: end if
```

Algorithm 5 Merge tasks

Require: Disjoint sets S_A, S_B

```
1: procedure MERGE(S_A, S_B)
2:      nt ← S_A.nt ∪ S_B.nt
3:      lsa ← S_A.lsa
4:      S_A ← S_B ← UNION(S_A, S_B)
5:      S_A.nt ← nt
6:      S_A.lsa ← lsa
7: end procedure
```

Finish: Algorithm 6 and Algorithm 7 shows the actions performed by the race detector at the start and end of a finish. At the end of a finish F, the disjoint sets of all tasks with F as the immediately enclosing finish is merged with the disjoint set of the ancestor task executing the finish.

Shared Memory Access: Determinacy races are detected when a read or write to a shared memory location occurs. When a write to a memory location M is performed by step u,

Algorithm 6 Start finish

Require: Start of finish F in task A

1: $F.parent \leftarrow A$

Algorithm 7 End finish

Require: Finish F

1: $A \leftarrow F.parent$
2: **for** $B \in F.joins$ **do**
3: MERGE(S_A, S_B)
4: **end for**

Algorithm 8 Write check

Require: Memory location M, Task A that writes to M

1: **for** $X \in M_s.r$ **do**
2: **if not** PRECEDE(X, A) **then**
3: *a determinacy race exists*
4: **else**
5: $M_s.r \leftarrow M_s.r - \{X\}$
6: **end if**
7: **end for**
8: **if not** PRECEDE($M_s.w, A$) **then**
9: *a determinacy race exists*
10: **end if**
11: $M_s.w \leftarrow A$

Algorithm 9 Read check

Require: Memory location M, Task A that reads M

```
1:  update = false
2:  for X ∈ M_s.r do
3:      if PRECEDE(X, A) then
4:          M_s.r ← M_s.r − {X}
5:          update ← true
6:      else if ISFUTURE(X) or ISFUTURE(A) then
7:          update ← true
8:      end if
9:  end for
10: if not PRECEDE(M_s.w, A) then
11:     a determinacy race exists
12: end if
13: if update then
14:     M_s.r ← M_s.r ∪ {A}
15: end if
```

the algorithm checks if the previous *writer* or the previous *readers* in the shadow memory space may execute in parallel with the currently executing step and reports a race. It updates the *writer* shadow space of M with the current task and removes any reader r if $r \prec u$. This is shown in Algorithm 8. When a read to a memory location M is performed by step u, the algorithm checks if the previous *writer* in the shadow memory space may execute in parallel with the currently executing step and reports a race. It adds the current task to the set of readers of M and removes any task r if $r \prec u$. Our algorithm differentiates between future tasks and async tasks: async tasks can be waited upon by only ancestor tasks using the finish construct and future tasks can be waited upon using the get() operation. Given a task A as argument, ISFUTURE returns true, if A is a future task. The readers shadow memory contains a maximum of one async task, but may contain multiple future tasks. During the read of a shared memory location by step s of an async task A, the algorithm replaces the previous async reader X by A, if X precedes s. This is shown in Algorithm 9.

Algorithm 10 Reachability check

Require: Tasks A, B

```
1:  procedure PRECEDE(A, B)
2:      return VISIT(A, B, {})
3:  end procedure
```

```
1:  procedure VISIT(A, B, Visited)
2:      if B ∈ Visited then
3:          return false
4:      end if
5:      Visited ← Visited ∪ {B}
6:      S_A ← FIND-SET(A)
7:      S_B ← FIND-SET(B)
8:      if S_A.pre ≤ S_B.pre and S_A.post ≥ S_B.post then
9:          return true
10:     end if
11:     if S_A.pre > S_B.pre then
12:         return false
13:     end if
14:     for all x in S_B.nt do
15:         if VISIT(A, x, Visited)
16:       then
17:             return true
18:         end if
19:     end for
20:     sa ← B.lsa
21:     while sa ≠ null do
22:         for all x in sa.nt do
23:             if VISIT(A, x, Visited) then
24:                 return true
25:             end if
26:         end for
27:         sa ← sa.lsa
28:     end while
29:     return false
30: end procedure
```

Given tasks A and B, PRECEDE routine shown in Algorithm 10 checks if task A must precede B by invoking routine VISIT which is also given in Algorithm 10. Lines 6–11 of VISIT routine returns true if the interval corresponding to the disjoint set of B is contained in the interval corresponding to the disjoint set of A. Lines 12–14 returns false, if the preorder value of A is greater than the preorder value of B, since the source of a non-tree join edge must have a lower preorder value than the sink of the non-tree edge. Lines 15–20 checks if B is reachable from A along the immediate non-tree predecessors of B. Lines 21–29 traverses paths which include the non-tree predecessors of the significant ancestors of B starting with the least significant ancestor of B. The routine returns true when a path from A to B is found or returns false when all the non-tree edges whose source has a preorder value greater than the preorder value of A are visited.

3.5 Theoretical Results

In this section, we present theoretical results for the computational complexity and correctness of our determinacy race detection algorithms for async-finish-future parallel programs. The following theorem presents the asymptotic complexity of the race detection algorithm.

Theorem 1. *Consider a program with async, finish and future constructs that executes in time T on one processor, creates a async tasks, f future tasks, performs n non-tree join edges and references v shared memory locations. Algorithms 1–10 can be implemented to check this program for determinacy races in $O(T(f+1)(n+1)\alpha(T, a+f))$ time using $O(a+f+n+v*(f+1))$ space.*

Proof. The size of the dynamic task dependence graph is $O(a+f+n)$ which includes $O(a+f)$ for the disjoint set, interval labels and LSA information and $O(n)$ for the non-tree joins. The size of the writer shadow memory is $O(v)$ and the worst case size of the reader

shadow memory is $O(v * (f + 1))$.

For every shared memory access, the PRECEDE function may be invoked $f + 1$ times in the worst case. The worst case complexity of a single invocation to PRECEDE function is $O((n + 1)\alpha(T, a + f))$ because it may visit n non-tree edges in the worst case and each time it involves a disjoint set operation. □

Here α is Tarjan's functional inverse of Ackermann's function which, for all practical purposes is bounded above by a constant, 4.

When the input program has no future tasks and no non-tree joins, the complexity of the algorithm is same as the SP-bags algorithm, because f and n are zero and the reader shadow space for each shared memory location contains a maximum of one reader. One important point to note is that the number of disjoint set union operations performed by our algorithm is same as the number of disjoint set union operations performed by the SP-bags algorithm. Although at a sync point, the SP-bags algorithm performs one union operation and our algorithm performs multiple union operations at the end of a finish construct, the total number of union operations is the same. This is because the SP-bags algorithm performs a set union operation when a task returns to its parent task, whereas the union operation in our algorithm is postponed until the end of finish corresponding to the IEF of the task.

We next present a proof sketch as to why a single run of Algorithms 1–10 correctly detects races in programs with async, finish and future. The following two lemmas gives the reasoning for storing one (async or future) task in the writer shadow memory and one async task (and multiple future tasks) in the reader shadow memory. Lemma 3 was presented in [4] for Cilk computation graphs with spawn and sync constructs, but it also holds for computation graphs with async, finish and future constructs.

Lemma 3. Suppose that three steps s_1, s_2, and s_3 execute in order in a serial, depth-first

execution of a computation graph, and suppose that $s_1 \prec s_2$ and $s_1 \parallel s_3$. Then, we have $s_2 \parallel s_3$.

Proof. Please refer to [4]. □

Lemma 4. Suppose that three steps s_1, s_2, and s_3 execute in order in a serial, depth-first execution of a computation graph, and let $Task(s_1) = T_A$, $Task(s_2) = T_B$, $Task(s_3) = T_C$, where T_A, T_B and T_C are async (non-future) tasks. Suppose that $s_1 \parallel s_2$ and $s_2 \parallel s_3$. Then, we have $s_1 \parallel s_3$.

Proof. Since T_A, T_B and T_C are async tasks, they can be waited upon using only finish operations and not by get() operations. Let us assume $s_1 \parallel s_2$, $s_2 \parallel s_3$ and $s_1 \prec s_3$. Since $s_1 \prec s_3$, a finish F must be executed by the task which is the lowest common ancestor of T_A and T_C in the spawn tree. There are two cases:

1. T_B is enclosed inside F. In this case F ensures that $s_2 \prec s_3$, which contradicts our assumption that $s_2 \parallel s_3$.

2. T_B is not enclosed inside F. In this case F ensures that $s_1 \prec s_2$, which contradicts our assumption that $s_1 \parallel s_2$. □

The following Lemma explains why it is sufficient to represent the reachability information at task level during a depth-first execution.

Lemma 5. Consider an execution of Algorithms 1–10 on a computation graph G. Suppose $s_{A1} \prec s_{B1}$ and $s_{B2} \prec s_C$ where $\text{Task}(s_{A1}) = T_A$, $\text{Task}(s_{B1}) = T_B$, $\text{Task}(s_{B2}) = T_B$ and $\text{Task}(s_C) = T_C$ and let s_C be the current step being executed and suppose that s_{A1}, s_{B1} and s_{B2} have executed before s_C during the depth-first execution. Then, for all completed steps s_A such that $\text{Task}(s_A) = T_A$, we have $s_A \prec s_C$.

Proof. We will consider four different cases based on whether the tasks are related by ancestor-descendant relationships or not:

1. T_A is an ancestor of T_B and T_B is an ancestor of T_C: In this case, the completed steps of T_A are the steps before the spawn of T_B (or an ancestor of T_B). All such steps must precede the steps of T_C, due to the spawn edge from T_B to T_C.

2. T_A is an ancestor of T_B and T_B is not an ancestor of T_C: If T_C is a descendant of T_A, then all completed steps of T_A must precede the spawn of T_C. Next, let us consider the case where T_C is not a descendant of T_A. In this case, T_A has completed the execution before the execution of T_C. Since T_B is not an ancestor of T_C and $s_{B2} \prec s_C$, there must be a join operation on T_B along the path from s_{B2} to s_C. Since T_A is an ancestor of T_B, there must also be a join operation on T_A along the path from s_{B2} to s_C before the join on T_B. This is based on the assumption that no data races have been detected so far during the execution of the program (See Lemma 1).

3. T_A is not an ancestor of T_B and T_B is an ancestor of T_C: Since T_A is not an ancestor of T_B, there must be a join edge from T_A along the path to s_{B1}. This means that for all steps s_C such that $\text{Task}(s_C) = T_C$, $s_A \prec s_C$.

4. T_A is not an ancestor of T_B and T_B is not an ancestor of T_C: In this case T_A must have completed execution before T_B and T_B must have completed execution before T_C during the depth first execution. There must be a join edge from T_A along the path to s_{B1} and a join edge from T_B along the path to s_C. This ensures that $s_A \prec s_C$. □

The next lemma discuss the correctness of the *Precede* function given in Algorithm 10.

Lemma 6. Consider an execution of Algorithms 1–10 on a computation graph G. PRECEDE(T_A,T_B) = *true* during the execution of s_j, where $\text{Task}(s_j) = T_B$ if and only if $s_i \prec s_j, \forall s_i$

such that Task(s_i) = T_A and s_i executes before s_j during the depth first execution of G.

Proof. ($\Rightarrow$) We will prove this by induction. Let $s_i = v_1, ..v_k, ..v_n = s_j$ be a path from s_i to s_j in the computation graph. Let $n = 1$ in which case there is an edge (s_i, s_j) in the computation graph.

1. (s_i, s_j) is a continue edge: In this case, s_i and s_j belong to the same task and are represented in the dynamic task reachability graph by the same node.

2. (s_i, s_j) is a spawn edge: In this case, T_A is the parent of T_B in the spawn tree and therefore T_B will have a higher preorder value and a lower postorder value than T_A.

3. (s_i, s_j) is a join edge: T_A and T_B will belong to the same disjoint set, if it is a tree join. If (s_i, s_j) is a non-tree join edge, then $T_A \in P(T_B)$, the non-tree predecessors of T_B.

In all three cases $Precede(T_A, T_B) = true$. Let us assume that if $s_i \prec s_j$ with a path length of $n = k$, then $Precede(T_A, T_B) = true$. Now, consider the case where $s_i \prec s_j$ with a path length of $n = k + 1$. Let (s_l, s_j) be the last edge along the path from s_i to s_j. Consider the following cases:

1. (s_l, s_j) is a continue edge: In this case there is a path of length k from s_i and s_l, where Task(s_l) = T_B. Therefore, by our induction hypothesis $Precede(T_A, T_B) = true$.

2. (s_l, s_j) is a spawn edge: If T_A is an ancestor of T_B in the dynamic task reachability graph, then T_B will have a higher preorder value and a lower postorder value than T_A and therefore $Precede(T_A, T_B) = true$. If T_A is not an ancestor of T_B, then there is a path from T_A to T_C and a path from T_C to T_B both of length less than or equal to k, where T_C is an ancestor of T_B. There are two possible cases: 1) T_C is the LSA of T_B, in which case $Precede(T_A, T_C) = true$ by our inductive hypothesis and 2) T_A and T_C are in the same disjoint set, in which case the label of T_A subsumes the label of T_B.

3. (s_l, s_j) is a join edge, where Task(s_l) = T_C: If (s_l, s_j) is a tree join edge, T_B and T_C are in the same disjoint set and $Precede(T_A, T_C) = true$ according to our inductive hypothesis. If (s_i, s_j) is a non-tree join edge, then $T_C \in P(T_B)$, the non-tree predecessors of T_B and $Precede(T_A, T_C) = true$ according to our inductive hypothesis.

($\Leftarrow$) Since $Precede(T_A, T_B)$ returned true, there must be a sequence of calls to Visit with arguments (T_A,T_B), (T_A,T_{X_1}) .. (T_A,T_{X_n}) where (T_A,T_{X_n}) returned true. Here, T_A and T_{X_n} must be in the same disjoint set or the label of T_A must subsume the label of T_{X_n}. (T_A,$T_{X_{k+1}}$) is invoked from (T_A,T_{X_k}) if and only if $T_{X_{k+1}}$ is an immediate predecessor of the LSA of T_{X_k} or if $T_{X_{k+1}}$ is an immediate predecessor of T_{X_k}. Therefore according to Lemma 5, $s_i \prec s_j$. □

Theorem 2. *Algorithms 1–10 detect a determinacy race for a given parallel program and data input if and only if a determinacy race exists.*

Proof. ($\Rightarrow$) Suppose Algorithms 1–10 detect a determinacy race when executing a step s_2. The three possible cases are

1. s_2 performs a write and $Precede(T_A, T_B) = false$, where Task($s_2$) = T_B and $T_A \in readers(l)$

2. s_2 performs a write and $Precede(T_A, T_B) = false$, where Task($s_2$) = T_B and $writer(l) = T_A$

3. s_2 performs a read and $Precede(T_A, T_B) = false$, where Task($s_2$) = T_B and $writer(l) = T_A$

In the first case, T_A is added to $readers(l)$ by step s_1 when s_1 performed a read of l (where Task(s_1) = T_A). Here s_1 executes before s_2 during the depth first execution. Since $Precede(T_A, T_B) = false$, s_1 and s_2 can logically execute in parallel according to Lemma 6 and therefore a determinacy race exists. The other two cases are similar.

($\Leftarrow$) We now show that if a program contains a determinacy race on a location l, our race detection algorithm reports a determinacy race on location l. Let s_1 and s_2 be two steps involved in a determinacy race on location l, where if there are multiple races on l, we choose the determinacy race for which the second step executes earliest in the depth-first execution order of the program. There are three possible ways the determinacy race could occur:

1. s_1 writes l and s_2 reads l: Suppose $writer(l) = T$ when s_2 is executed. If $T = T_A$ where Task(s_1) = T_A, then since $s_1 \parallel s_2$, according to Lemma 6 $Precede(T_A, T_B) = false$ and the algorithm will report a race. If $T \neq T_A$, then $writer(l)$ was last updated by step s such that s executed after s_1 but before s_2. If $s_1 \parallel s$, there must exist a write-write determinacy race between s_1 and s since they access a common memory location l in parallel. This contradicts our assumption that s_2 executes earliest during a depth-first execution among all second steps that are involved in a determinacy race on l. If $s_1 \prec s$, then we have have $s \parallel s_2$ by Lemma 3. According to Lemma 6, $Precede(T_A, T_B) = false$ and the algorithm will report a race.

2. s_1 writes l and s_2 writes l: This case is similar to write-read race.

3. s_1 reads l and s_2 writes l: Suppose $T_A \in readers(l)$, where Task(s_1) = T_A when s_2 is executed. Then since $s_1 \parallel s_2$, according to Lemma 6 $Precede(T_A, T_B) = false$ and the algorithm will report a race. Now let us consider the case $T_A \notin readers(l)$. If s_1 adds T_A to $readers(l)$, then T_A was removed from $readers(l)$ by a step s such that s executes after s_1 but before s_2 during the depth-first execution. This implies that $s_1 \prec s$. Let us assume that there is a sequence of steps $s'_1, s'_2..s'_n$ which reads l before s_2 such that $s'_1 \prec s'_2 \prec .. \prec s'_n$, where $s = s'_1$. By transitivity, we have $s_1 \prec s'_n$. By Lemma 3, it follows that $s'_n \parallel s_2$, since $s_1 \parallel s_2$ and therefore a race between s'_n

and s_2 will be reported. Now let us consider the case where s_1 does not add T_A to *readers*(l). In this case, T_A must be an async task and during the execution of s_1, *readers*(l) contains T', where Task(s') = T'. Here s' performed a read of l and $s' \parallel s_1$ and T' also must be an async task. Since $s_1 \parallel s_2$, by Lemma 4 $s' \parallel s_2$. Looking at the sequence of updates to *readers*(l), there must exist a task $T_K \in$ *readers*(l), where Task(s_k) = T_K, s_k performed a read of l and $s' \prec s_k$. Since $s' \parallel s_2$, Lemma 3 implies that $s_k \parallel s_2$ and a race between s_k and s_2 will be reported. □

3.6 Experimental Results

In this section, we present experimental results for our determinacy race detection algorithm. The race detector was implemented as a new Java library for detecting determinacy races in HJ programs containing async, finish and future constructs. The benchmarks written in HJ were instrumented for race detection during a bytecode-level transformation pass implemented on HJ's Parallel Intermediate Representation (PIR) [34]. The PIR extends Soot's Jimple IR [35] with parallel constructs such as async, finish, and future. The instrumentation pass adds the necessary calls to our race detection library at async, finish and future boundaries, future get operations, and also on reads and writes to shared memory locations.

Our experiments were conducted on a 16-core Intel Ivybridge 2.6 GHz system with 48 GB memory, running Red Hat Enterprise Linux Server release 7.1, and Sun Hotspot JDK 1.7. To reduce the impact of JIT compilation, garbage collection and other JVM services, we report the mean execution time of 10 runs repeated in the same JVM instance for each data point. We evaluated the algorithm on the following benchmarks:

Benchmark	#Tasks	#NTJoins	#SharedMem	#AvgReaders	Seq (millisecs)	Racedet (millisecs)	Slowdown (Racedet/Seq)
Series-af	999,999	0	4,000,059	0.75	483,224	484,746	1.00
Series-future	999,999	0	6,000,059	0.66	487,134	487,985	1.00
Crypt-af	12,500,000	0	1,150,000,682	0.74	15,375	119,504	7.77
Crypt-future	12,500,000	0	1,175,000,682	1.23	15,517	128,234	8.26
Jacobi	8,192	34,944	641,499,805	1.70	3,402	27,388	8.05
Strassen	30,811	33,612	1,610,522,196	0.94	6,281	33,618	5.35
Smith-Waterman	1,608	4,641	1,652,175,806	1.56	3,488	34,558	9.92

Table 3.3 : Runtime overhead for determinacy race detection. **#Tasks** is the dynamic number of tasks created, **#NTJoins** is the dynamic number of non-tree joins performed, **#SharedMem** is the total number of dynamic shared memory accesses performed and **#AvgReaders** is the average number of readers stored in the shadow memory per access and memory location. **Seq** is the sequential execution time in milliseconds and **Racedet** is the execution time with race detection enabled in milliseconds. **Slowdown** is the slowdown due to race detection.

- **Series-af:** Fourier coefficient analysis from JGF [36] benchmark suite (Size C), parallelized using async and finish.
- **Series-future:** Fourier coefficient analysis from JGF benchmark suite (Size C), parallelized using futures.
- **Crypt-af:** IDEA encryption algorithm from JGF benchmark suite (Size C), parallelized using async and finish.
- **Crypt-future:** IDEA encryption algorithm from JGF benchmark suite (Size C), parallelized using futures.
- **Jacobi:** 2 dimensional 5-point stencil computation on a 2048×2048 matrix, where each tasks computes a 64×64 submatrix.

- **Strassen:** Multiplication of 1024×1024 matrices using Strassen's algorithm. The implementation uses a recursive cutoff of 32×32.

- **Smith-Waterman:** Sequence alignment of two sequences of size 10000. The alignment matrix computation is done by 40×40 future tasks.

The first four benchmarks were derived from the original versions in the JGF suite. The next two, Jacobi and Strassen were translated by the authors from OpenMP versions of those programs in the Kastors [37] benchmark suite. The original versions of these benchmarks used the OpenMP 4.0 *depends* clause, in which tasks specify data dependence using **in**, **out** and **inout** clauses. The translated versions of these benchmarks used future as the main parallel construct, with get() operations used to synchronize with previously data dependent tasks. In general, this kind of task dependences cannot be represented using only async-finish constructs without loss of parallelism. The Smith-Waterman benchmarks uses futures and is based on a programming project in COMP322, an undergraduate course on parallel computing at Rice University.

The results of our evaluation is given in Table 3.3. The first column lists the benchmark name, and the second column shows the dynamic number of tasks (#Tasks) created for the inputs specified above. The third column shows the number of non-tree joins (#NTJoins) performed by each of the applications (the subset of future get() operations that are non-tree-joins). The fourth column shows the total number of shared memory accesses (#SharedMem) performed by the applications (all accesses to instance/static fields and array elements). The fifth column (#AvgReaders) shows the average number of past parallel readers per location stored in the shadow memory when a read/write access is performed on that location. (The average is computed across all accesses and all locations.) For a given access, the number of such stored readers will be either zero or one for programs containing

only async and finish constructs, thereby ensuring that the average must be in the $0 \ldots 1$ range for async-finish programs. For programs with futures, the number of stored readers can be greater than one, if the location being accessed is in the read-shared state and is read by multiple tasks that can potentially execute in parallel each other. Thus, #AvgReaders can be any value that is ≥ 0, for programs with futures.

The next column (Seq) reports the average execution time of the sequential (serial elision) version of the benchmark, and the following column (Racedet) reports the average execution time of a 1-processor execution of the parallel benchmark using the determinacy race detection algorithm introduced in this chapter. Finally, the Slowdown column reports the ratio of the Racedet and Seq values.

We can make a number of observations from the data in Table 3.3. First, if we compute the Seq/#Tasks ratio for all the benchmarks, we can see that the Crypt-af and Crypt-future benchmarks perform $\cong 100\times$ less work per task on average, relative to all the other benchmarks. This is the primary reason why the Crypt-af and Crypt-future benchmarks exhibit slowdowns of $7.77\times$ and $8.26\times$. With less work per task, the overhead per task during race detection becomes more significant than in other benchmarks; further, creating data structures for large numbers of tasks puts an extra burden on garbage collection and memory management. However, it is important to note that the slowdowns for Series-af and Crypt-af are comparable to the slowdowns reported for the ESP-Bags algorithm [38] that only supported async and finish, thereby showing that our determinacy race detector does not incur additional overhead for async/finish constructs relative to state-of-the-art implementations.

Next, we see that the number of non-tree joins performed by Series-af and Crypt-af is zero, since they are async-finish programs for which all join (finish) operations appear as tree-join edges in the computation graph (Section 3.3). Since their corresponding future versions, Series-future and Crypt-future, used futures to implement async-finish synchro-

nization, their future get() operations also appear as tree-join edges in the computation graph, thereby resulting in zero non-tree joins as well. However, the future versions of these two benchmarks have higher number of shared memory accesses than the async-finish versions, due to the additional writes and reads of future references which happened to be stored in shared (heap) locations for both benchmarks. In particular, we know that the reference to each future task must be subjected to at least one write access (when the future task is created) and one read access (when a get() operation is performed on the future), though more accesses are possible. Since Series-future creates 999,999 future tasks, we see that the difference in the #SharedMem values for Series-future and Series-af is 2,000,000 which is very close to the lower bound of $2 \times 999,999$. Likewise, for Crypt-future and Crypt-sf, the number of tasks created is 12,500,000 and the difference in the #SharedMem values is 25,000,000 which exactly matches the lower bound of $2 \times 12,500,000$. The slowdown for Crypt-future is higher than that of Crypt-af due to two reasons: 1) the additional number of memory accesses due to the future references and 2) the average number of readers stored in the shadow memory is higher, because of the presence of future tasks.

The slowdowns for Jacobi, Smith-Waterman and Strassen (8.05×, 9.92×, and 5.35×) are positively correlated by the values of #SharedMem, #AvgReaders, and 1/Seq, and these correlations can help explain the relative slowdowns for the three benchmarks. A larger value of #SharedMem leads to a larger slowdown due to the overhead of processing additional shared memory accesses. A larger value of #AvgReaders leads to a larger slowdown because the number of reachability queries required per shared memory access is equal to the number of readers present in the shadow memory for that location. A larger value of 1/Seq indirectly leads to a larger slowdown due to the smaller available time to amortize the overheads of race detection.

Finally, we observe that the slowdowns are not significantly impacted by the number of

non-tree edges. This is because the producer and consumer tasks of a future object happen to be closely located to each other in the computation graph (for these benchmarks), usually only requiring 1-2 hops involving non-tree edges in the graph traversal.

3.7 Related Work

Dynamic data race detection techniques target either structured parallelism or unstructured parallelism. Race detection for unstructured parallelism typically uses vector clock algorithms, e.g., [1, 2]. These algorithms are impractical for task parallelism because either the vector clocks have to be allocated with a size proportional to the maximum number of simultaneously live tasks (which can be unboundedly large) or precision has to be sacrificed by assigning one clock per processor or worker thread, thereby missing potential data races when two tasks execute on the same worker.

For programs with structured fork-join parallelism, prior work has shown that for a single program input, in a single execution, one can pinpoint an example data race, or else there can be no data races with any interleaving based on that input [3]. Mellor-Crummey [3] presented Offset-Span labeling algorithm for nested fork-join constructs, which is an extension of English-Hebrew labeling scheme [39]. The idea behind their techniques is to attach a label to every thread in the program and use these labels to check if two threads can execute concurrently. The length of the labels associated with each thread is bounded by the maximum nesting depth of fork-join in the program. Our approach uses a labeling scheme which is of constant size to store reachability information between ancestor-descendant tasks. While Offset-Span labeling algorithm supports only nested fork-join constructs, our algorithm supports a more general set of computation graphs.

Feng and Leiserson [4] introduced the SP-bags algorithm for Cilk's fully-strict par-

allelism, which uses only a constant factor more memory than does the program itself. Bender et al. [26] presented parallel SP-hybrid algorithm which uses English-Hebrew labels and SP-bags to detect races in Cilk programs. Despite its good theoretical bounds, the paper did not provide an implementation of the algorithm. Raman et al. [5] extended the SP-bags algorithm to support async-finish parallelism. They subsequently proposed SPD3 algorithm [27] also for async-finish parallelism, which operates in parallel. The algorithm determines series-parallel relationships between steps by a lookup of the lowest common ancestor in the dynamic program structure tree. In contrast to these approaches, our data race detection algorithm handles async, finish and futures, which can create more general computation graphs than those that can be generated by async-finish parallelism.

3.8 Summary and Future Wrok

In this chapter, we presented the first known determinacy race detector for dynamic task parallelism with futures. As with past determinacy race detectors, our algorithm guarantees that all potential determinacy races will be checked so that if a race is reported for a given input in one run of our algorithm, it will always be reported in all runs. Likewise, if no race is reported for a given input, then all executions with that input are guaranteed to be race-free. Our approach builds on a novel data structure called the *dynamic task reachability graph* which models task reachability information for non-strict computation graphs in an efficient manner. We presented a complexity analysis of our algorithm, and also discussed its correctness. We implemented the algorithm, and evaluated it on benchmarks which generate both strict and non-strict computations. The results indicate that the performance of our approach is similar to other efficient algorithms for spawn-sync and async-finish programs and degrades gracefully in the presence of futures. Specifically, the experimental results show that the slowdown factor observed for our algorithm relative to the sequential

version is in the range of 1.00× – 9.92×, which is very much in line with slowdowns experienced for fully strict computation graphs.

There are many opportunities for future research to build on the results of this chapter. One direction for future work is to use a hybrid static+dynamic approach for computing the task reachability information, thereby reducing the runtime overhead of race detection. Another opportunity for future work is to support race detection in parallel programs with other kinds of point-to-point synchronization constructs including task dependences, doacross, and phasers.

Chapter 4

Test-Driven Repair of Data Races in Task-Parallel Programs

In this chapter, we address the problem of inserting `finish` statements in parallel programs, where parallelism is expressed using `async` statements and (for the sake of generality) the program may already contain some `finish` statements inserted by the programmer. Our approach determines where additional finish statements should be inserted to guarantee correctness, with the goal of maximizing parallelism. This insertion of `finish` statements can be viewed as *repairing unsynchronized or under-synchronized parallel programs*.

Past solutions to the problem of repairing parallel programs have used static-only or dynamic-only approaches, both of which have significant limitations in practice. Static approaches can guarantee soundness in many cases but face severe limitations in precision when analyzing medium or large-scale software with accesses to pointer-based data structures in multiple procedures. Dynamic approaches are more precise than static analyses, but their proposed repairs are limited to a single input and are not reflected back in the original source program. Our method treads the middle ground between these two classes of approaches. As in dynamic approaches, we execute the program on a concrete test input and determine the set of data races for this input dynamically. However, next we compute a set of finish placements that rule out these races but also respect the static scoping rules of the program, and can therefore be inserted back into the program.

Consider the Mergesort example in Figure 4.1. The programmer expressed their intuition that the two recursive calls in lines 4 and 5 can execute in parallel. The algorithms

```
1  static void mergesort(int[] A, int M, int N) {
2    if (M < N) {
3      final int mid = M + (N - M) / 2;
4      async mergesort(A, M, mid);
5      async mergesort(A, mid + 1, N);
6      merge(A, M, mid, N);
7    }
8  }
9  ...
10 mergesort(A, 0, size-1);  //Call inside main
```

Figure 4.1 : Mergesort program. A finish statement is needed around lines 4-5 for correctness and maximal parallelism.

```
1  static void quicksort(int[] A, int M, int N) {
2    if(M < N) {
3      point p = partition(A, M, N);
4      int I = p.get(0);
5      int J =p.get(1);
6      async quicksort(A, M, J);
7      async quicksort(A, I, N);
8    }
9  }
10 ...
11 quicksort(A, 0, size-1);  //Call inside main
```

Figure 4.2 : Quicksort program. A finish statement is needed around line 11 for correctness and maximal parallelism.

introduced in this chapter can determine that a finish statement is needed around lines 4 and 5 for correctness and maximal parallelism. Now, consider the Quicksort example in Figure 4.2. Again, the programmer expressed their intuition that the two recursive calls in lines 6 and 7 can execute in parallel. While inserting a finish statement around lines 6 and 7 would be correct, our algorithms can determine that inserting a finish around line 11 is better because it also prevents data races, yet yields more parallelism than a finish statement around lines 6 and 7 by avoiding the need for nested recursive finish constructs.

The rest of the chapter is organized as follows. In Section 4.1, we formulate the problem that we are solving. Section 4.2 presents our contributions. Sections 4.3-4.6 present our solution. Section 4.7 describes our experiments. Section 4.8 discusses related work. Finally, Section 4.9 summarizes our conclusions.

4.1 Problem Statement

A program execution contains a *data race* when there are two or more accesses to the same variable, at least one of which is a write, and the accesses are unordered by either synchronization or program order. The primary goal of test-driven repair tool is to ensure data race freedom* for the provided inputs. In addition to this, the repaired programs must be well formed and must provide good performance. Although the tool is applied iteratively for different test inputs, we define the problem statement for a single iteration of the tool as follows:

Problem 2. (Repair) Given a program P and input, ψ, find a set of program locations in P where finish statements must be introduced such that

1. The program after insertion of finish statements has no data races for input, ψ.
2. The newly inserted finish statements must respect the lexical scope of the input program.
3. The program after insertion of finish statements must maximize the available parallelism.
4. The program after insertion of finish statements must have the same semantics as the serial elision, i.e., the program with no parallel constructs.

*Since the repaired program is data-race-free, it has the same semantics for all memory models.

```
1 async A( ); // Execution Time = 500
2 async B( ); // Execution Time = 10
3 async C( ); // Execution Time = 10
4 async D( ); // Execution Time = 400
5 async E( ); // Execution Time = 600
6 async F( ); // Execution Time = 500
```

Figure 4.3 : async-finish program with execution times. The dependences in the program are `B` → `D`, `A` → `F` and `D` → `F`

5. The program statements remain in the same order. □

The criterion of maximal available parallelism is abstractly defined as follows:

Definition 5. A program is said to have maximal parallelism, if it has minimum critical path length (CPL), where critical path is the longest path in the computation graph of the program. Critical path length could also be defined as the execution time of a program on a computer with unbounded number of processors.

Consider the program shown in Figure 4.3, where the execution times for each of the tasks is given. Let us assume that `D` is dependent on `B` and `F` is dependent on both `A` and `D`. Some of the possible finish placements to satisfy these dependences are given in Figure 4.4, along with their critical path lengths. For example `( A B C ) ( D ) E F` corresponds to inserting a finish statement which encloses `async A`, `async B`, `async C` and another finish statement which encloses `async D`. The choice of finish placements can have a big impact on the critical path length and available parallelism. The number of possible finish placements can grow exponentially with program size, and finding the best possible finish placement is a complex problem. The problem becomes harder in the presence of function calls and nested task parallelism, where asyncs are nested inside asyncs.

To demonstrate how a finish statement can violate the scope of the input program, let us consider the program given in Figure 4.5. There are two data races in this program: `A2`

```
( A ) ( B ) C ( D ) E F  // CPL = 1510
( A B ) C ( D ) E F      // CPL = 1500
( A B C ) ( D ) E F      // CPL = 1500
( A ( B ) C D E ) F      // CPL = 1110
```

Figure 4.4 : Few possible finish placements for the program in Figure 4.3 and their critical path lengths. Parentheses represent finish statements

```
1 if (...) {
2   async {    ...     } // A1
3   async { x = ...   } // A2
4 }
5 async { y    = ... } // A3
6 async { ...   =  x+y } // A4
```

Figure 4.5 : Async-finish code which demonstrates the scoping issues in finish insertion

→ A4 and A3 → A4. There are two ways to fix these data races using finish statements:

- Enclose A2 inside a finish statement and A3 inside another finish statement.
- Enclose A1, A2 and A3 inside a single finish statement.

Note that we cannot insert a new finish statement which encloses A2 and A3, but does not enclose A1. A program with such finish statements is not well formed. The finish placement algorithm must eliminate such cases from the set of potential repairs.

In this chapter, we present a tool that computes finish placements which guarantee data race freedom for the provided inputs, retain maximal parallelism, and respect scope rules of the input program.

4.2 Contributions

The main contributions of this work are as follows:

- A dynamic finish placement algorithm which determines points in scoped dynamic program structure tree (S-DPST) where additional finish constructs need to be inserted to repair data races in the input program. We introduce a data structure called scoped dynamic program structure tree (S-DPST), which can be used to analyze the execution of parallel programs with async and finish constructs.

- A static finish placement algorithm which maps points in the S-DPST where additional finish statements are required to points in the program (AST).

- An evaluation of our method on a range of benchmarks, including standard benchmarks from the HJ Bench, BOTS, JGF, and Shootout suites, as well as student homework submissions from a parallel computing course. The evaluation establishes the effectiveness of our approach with respect to compile-time overhead, precision, and performance of the repaired code.

4.3 Overview

In this section, we present an overview of our approach to test-driven repair of parallel programs. The overall approach is incremental by design. A single iteration of the tool takes as input an unsynchronized or under-synchronized parallel program with async and finish constructs and a test input. It executes the program in the canonical sequential (depth-first) order with the given input, and identifies all potential data races by employing a modified version of the ESP-bags algorithm [5] that builds an extended Dynamic Program Structure Tree (DPST) [27] for that execution. The output of the iteration identifies static points in the program where finish statements should be inserted to cover all data races for that particular execution.

A high level view of the tool is given in Figure 4.6. The three main steps in test-driven

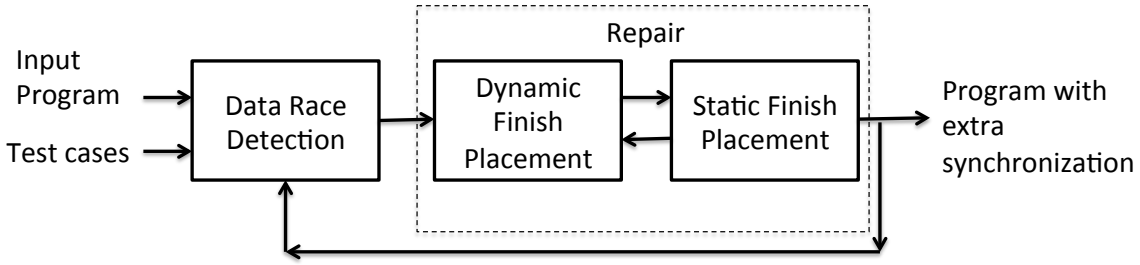

Figure 4.6 : High level view of test-driven repair

repair of data races are:

- **Data Race Detection:** Our tool executes the program sequentially with the provided input to identify data races in the program. To identify data races, the tool uses a modified version of the ESP-bags algorithm, which is explained in Section 4.4.

  While the program executes, the data race detector constructs a data structure called Scoped Dynamic Program Structure Tree (S-DPST). S-DPST is an ordered rooted tree that captures the relationships among the async, finish, scope and step instances in the program, where a step instance is a maximal sequence of statement instances in a particular scope with no asyncs and finishes. Section 4.4.2 describes the S-DPST in detail.

- **Dynamic Finish Placement:** Our tool analyzes the S-DPST, which is annotated with the set of data races, to find the points in the S-DPST where additional finish statements are required. Section 4.5 presents our dynamic finish placement algorithm.

- **Static Finish Placement:** Our tool maps points in the S-DPST where additional finish statements are required to points in the program (AST). Section 4.6 presents our static finish placement algorithm.

We iteratively perform dynamic and static finish placements until all data races discovered in the program with the test input are repaired.

4.4 Data Race Detection

In this section, we present the modified version of ESP-Bags data race detection algorithm and S-DPST, the principal data structure for our analysis.

4.4.1 Multiple Reader-Writer ESP-Bags

Detecting data races is an important step in identifying the synchronization necessary to maintain correctness of parallel programs. In this work, we use the ESP-bags algorithm [5] to identify data races in parallel programs with `async` and `finish` constructs. The ESP-bags algorithm detects data races in a given program if and only if a data race exists. This algorithm performs a sequential depth first execution of the parallel program with the given input. By monitoring the memory accesses in the sequential execution, the algorithm identifies data races that may occur in some parallel execution of the program for that input. If the algorithm reports no data races for an execution of a program with an input, then no execution of the program for that input will encounter a data race. The sequential depth first execution of a parallel program is similar to the execution of an equivalent sequential program obtained by eliding the keywords `async` and `finish`. The ESP-bags algorithm maintains an access summary for each memory location monitored for data races; each location's access summary requires $O(1)$ space.

The ESP-bags algorithm reports only a subset of all the data races present in the program for a given input. This limitation is due to the constraint that ESP-bags keeps track of only one writer and one reader corresponding to each memory location. Consider the async-finish code given in Figure 4.7. There are two Read $\rightarrow$ Write data races in this code

```
1 async { ... = x;} // A1
2 async { ... = x;} // A2
3 async { x = ... } // A3
```

Figure 4.7 : Async-finish code with multiple data races

snippet due to parallel accesses to the global variable `x`. The first data race is from the async `A1` to the async `A3` and the second data race is from the async `A2` to the async `A3`. The ESP-bags algorithm reports only the data race `A1` → `A3`, because it keeps track of only one of the readers of a memory location. This data race could be fixed by enclosing `A1` inside a finish, but this will not fix the data race from `A2` to `A3`.

The goal of our tool is to fix all potential data races for a given input. To achieve this goal, we use a modified version of the ESP-bags algorithm that keeps track of all readers and writers for each memory location. In the rest of the chapter, we refer to the original ESP-bags algorithm as Single Reader-Writer ESP-Bags (SRW ESP-Bags) and the modified version as Multiple Reader-Writer ESP-Bags (MRW ESP-Bags).

4.4.2 Scoped Dynamic Program Structure Tree

In this section, we present the principal data structure used for our analysis: Scoped Dynamic Program Structure Tree (S-DPST). S-DPST is an extension of Dynamic Program Structure Tree (DPST) [27], which is used in parallel data race detection for structured parallel programs.

Definition 6. The Scoped Dynamic Program Structure Tree (S-DPST) for a given execution is a tree in which all leaves are step instances, and all interior nodes are async, finish and scope instances. The parent relation is defined as follows:

- Async instance A is the parent of all async, finish, scope and step instances directly

executed within A.

- Finish instance F is the parent of all async, finish, scope and step instances directly executed within F.

- Scope instance S is the parent of all async, finish, scope and step instances directly executed within S.

There is a left-to-right ordering of all S-DPST siblings that reflects the left-to-right sequencing of computations belonging to their common parent.

A Scope node represents a scope encountered during the execution of the program. For instance, a scope node may represent an `if` statement, a `while` loop or a function call. The scope nodes ensures that the start and end points of a newly introduced finish statement are in the same scope of the input program (see Section 4.5). A *non scope node* refers to an async, finish or step node.

Step S1 is called the *source* of a data race involving steps S1 and S2, if S1 occurs before S2 in the depth first traversal of the S-DPST. S2 is said to be the *sink* of the data race. Data races are represented in S-DPST using directed edges from the step which is the *source* of the data race to the step which is the *sink* of the data race.

Example: Consider the incorrectly synchronized Fibonacci program in Figure 4.8. The program is incorrect because the async statement in line 12 can execute in parallel with line 14, both of them access the field `X.v` and one of them is write. Similarly there is a data race due to the access to field `Y.v`. Fig. 4.9 shows a subtree of S-DPST for the Fibonacci program. Each node is labeled with the type of the node and a number which indicates the order in which the node is visited in a depth first traversal of the tree. `Async0` corresponds to instances of the async statement in line 19, `Async1` corresponds to instances of the async

```
1  static class BoxInteger {
2    public int v;
3  }
4
5  void fib (BoxInteger ret, int n) {
6    if ( n < 2) {
7      ret.v = n;
8      return;
9    }
10   final BoxInteger X = new BoxInteger();
11   final BoxInteger Y = new BoxInteger();
12   async fib (X, n-1); // Async1
13   async fib (Y, n-2); // Async2
14   ret.v = X.v + Y.v;
15 }
16
17 public static void main (String[] args) {
18   ....
19   async fib(result, 3); // Async0
20   ....
21 }
```

Figure 4.8 : Incorrectly synchronized Fibonacci program

in line 12 and `Async2` corresponds to instances of the async in line 13. The scope nodes in the S-DPST are labeled `Fib` and `If`, which corresponds to the scope of the `fib` function and the `if` statement in line 6 respectively. The data races due to the parallel accesses to `X.v` and `Y.v` are shown using the dotted directed edges.

Definition 7. A non-scope child of a node p in S-DPST is a node c, which is a direct descendent of p with only scope nodes along the path from p to c.

Definition 8. The non-scope least common ancestor (NS-LCA) of two nodes n_i and n_j in S-DPST is a node l_{ij} such that if l'_{ij} is the Least Common Ancestor (LCA) of n_i and n_j in the S-DPST, then l_{ij} is the first non-scope node along the path from l'_{ij} to the root of the S-DPST.

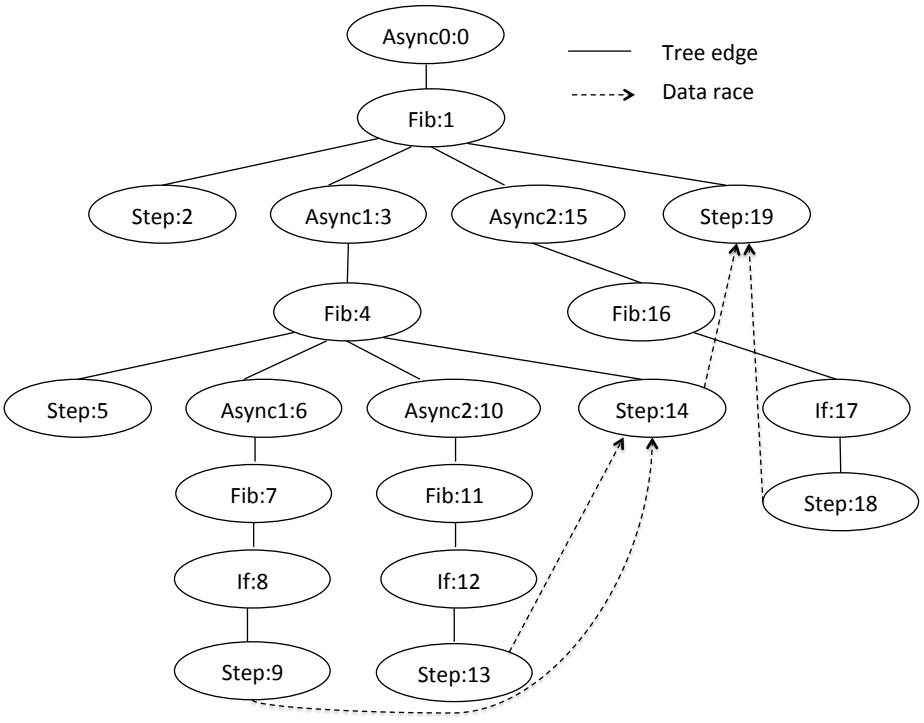

Figure 4.9 : Subtree of S-DPST for Fibonacci

Definition 9. The non-scope least common ancestor (NS-LCA) of a data race, D in S-DPST is the NS-LCA of n_i and n_j, where n_i is the source and n_j is the sink of the data race, D.

In Figure 4.9, *Step:5*, *Async1:6*, *Async2:10* and *Step:14* are the non-scope children of the node *Async1:3*. The NS-LCA of *Step:9* and *Step:14* is *Async1:3*.

4.5 Dynamic Finish Placement

In this section, we present the algorithm for dynamic finish placement, which involves two main steps: dynamic dependence graph construction which is presented in Section 4.5.1 and the application of a dynamic programming algorithm on the dependence graph, which is presented in Section 4.5.2.

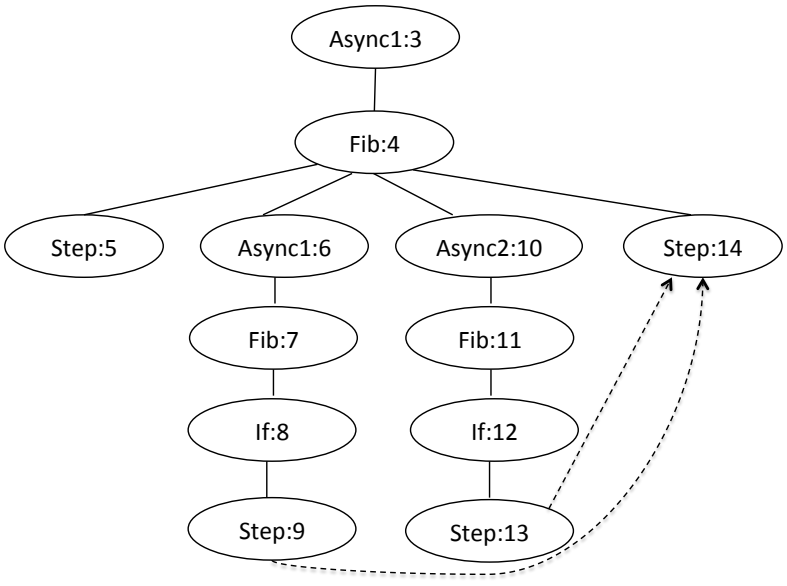

Figure 4.10 : A subtree rooted at NS-LCA for Fibonacci

4.5.1 Dynamic Dependence Graph Construction

In this section we present the method used to construct a dependence graph from the subtree rooted at a NS-LCA. Consider the subtree of the S-DPST rooted at L, where L is the NS-LCA of each of the data races $D_1..D_k$. Let $C_1..C_n$ be the non-scope children of the node L in the S-DPST. The graph we construct has n nodes, where each node corresponds to a non-scope child of L. The dependence graph has k edges, where each edge corresponds to a data race. The source of the edge corresponding to data race, D_i is the non-scope child of L, which is the ancestor of the source of D_i. Similarly the sink of the edge corresponding to data race, D_i is the non-scope child of L, which is the ancestor of the sink of D_i.

Example: The S-DPST in Figure 4.9 has two NS-LCAs: *Async0:0* and *Async1:3*. We demonstrate the dependence graph construction on the subtree rooted at *Async1:3*, which is given in Figure 4.10. Figure 4.11 shows the dependence graph constructed using the

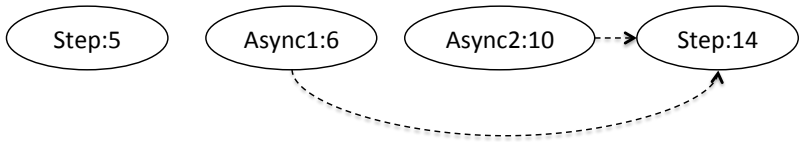

Figure 4.11 : Dependence graph constructed from the subtree in Figure 4.10

method presented above. The nodes in the graph are the non-scope children of *Async1:3* and the edges represent the data races between their descendent steps.

4.5.2 Algorithm

In this section, we present the core algorithm of our test-driven repair tool. The algorithm takes as input the dependence graph constructed from the subtree rooted at a NS-LCA by the method presented in Section 4.5.1 and finds the set of finishes needed to fix the data races represented in the dependence graph. The problem of optimal finish placement could be stated formally as follows: Let $G = (V, E)$ be a directed acyclic graph where $V = \{1..n\}$ is the set of vertices and $E = \{(x_1, y_1)..(x_m, y_m)\}$ is the set of edges. The set of edges, E satisfy the property $\forall (x_i, y_i) \in E, x_i < y_i$. The execution time of each vertex i is represented as t_i. We are interested in finding a set of points in the graph, where finish nodes need to be introduced such that it resolves all the data races and minimizes the execution time of G. The set of program points where finish nodes need to be introduced is represented using a set of ordered pairs, $FinishSet = \{(s_1, e_1)..(s_n, e_n)\}$, where each $(s_i, e_i) \in FinishSet$ represents a finish block which encloses the set of vertices $s_i..e_i$. To summarize, we need to compute $FinishSet$, such that

- if $(i, j) \in E$, $\exists\ (s, e) \in FinishSet$ where $1 \leq s \leq i \leq e < j$
- $COST(G) = \max_{i=1,n}(EST(i, 1..i-1) + t_i)$, is minimized

$$OPT(i,j) = \min_{i \le k < j} \begin{cases} OPT(i,k) + OPT(k+1,j) & succ(i..k) \cap \{k+1..j\} \neq \emptyset \\ \max(OPT(i,k), EST(k+1, i..k) + OPT(k+1,j)) & \text{otherwise} \end{cases} \tag{4.1}$$

Figure 4.12 : Optimal substructure of finish placement

$$EST(i+1, i..i) = \begin{cases} 0 & \text{if } i \text{ is an async} \\ t_i & \text{otherwise} \end{cases} \tag{4.2}$$

$$EST(i,j) = \begin{cases} OPT(i,k) + EST(k+1, i..k) & succ(i..k) \cap \{k+1..j\} \neq \emptyset \\ EST(k+1, i..k) + EST(j+1, k+1..j) & \text{otherwise} \end{cases} \tag{4.3}$$

Figure 4.13 : Earliest start time computation. k is the optimal partitioning point for $i..j$

where $EST(j, i..j-1)$ represents the *earliest start time* of the node j with respect to the nodes $i..j-1$. The quantity $EST(i, 1..i-1) + t_i$ represents the *earliest completion time* of the node i.

This problem exhibits *optimal substructure*. That is, the solution to the problem could be expressed in terms of solutions to smaller subproblems as shown in Figure 4.12. $OPT(i,j)$ represents the optimal cost for the subproblem involving the nodes $i..j$ and the corresponding edges. $succ(i)$ represents the set of nodes to which there is an edge from i, and $succ(i..k) = succ(i) \cup succ(i+1).. \cup succ(k)$. $OPT(i,j)$ is computed from two sub problems $i..k$ and $k+1..j$. The value of k is chosen such that it gives us the minimum value of $OPT(i,j)$. We refer to k as the *optimal partitioning point* for the problem $i..j$. The possible *partitioning points* are $i, i+1, ..j-1$. The first case represents where there are edges from the first partition ($i..k$) to the second partition ($k+1..j$). In this case, a finish is required around the first partition to satisfy the dependence from the first partition to the second partition. The second case represents the case where there are no edges from the

Algorithm 11 Dynamic finish placement algorithm

Require: Graph, G and Execution time $t[1..n]$ of nodes $1..n$ in G

Ensure: Opt, $Partition$, $Finish$ arrays

1: **for** i = 1 to n **do**
2: $Opt[i][i] \leftarrow t[i]$
3: $Partition[i][i] \leftarrow i$
4: $Finish[i][i] \leftarrow false$
5: **if** i is an *async* node **then**
6: $EST[i+1, i..i] = 0$
7: **else**
8: $EST[i+1, i..i] = t[i]$
9: **end if**
10: **end for**
11: **for** $s = 2$ to n **do**
12: **for** $i = 1$ to $n - s + 1$ **do**
13: $j \leftarrow i + s - 1$
14: **for** $k = i$ to $j - 1$ **do**
15: $C_{min} = +\infty$
16: **if** $succ(i..k) \cap \{k+1..j\} = \emptyset$ **then**
17: $c \leftarrow max(Opt[i][k], EST[k+1, i..k] + Opt[k+1][j])$
18: $f \leftarrow false$
19: $e \leftarrow EST[k+1, i..k] + EST[j+1, k+1..j]$
20: **else if** VALID(i, k) **then**
21: $c \leftarrow Opt[i][k] + Opt[k+1][j]$
22: $f \leftarrow true$
23: $e \leftarrow Opt[i][k] + EST[j+1, k+1..j]$
24: **end if**
25: **if** $c < C_{min}$ **then**
26: $C_{min} \leftarrow c$
27: $p \leftarrow k$
28: $finish \leftarrow f$
29: $est \leftarrow e$
30: **end if**
31: **end for**
32: **end for**
33: $Opt[i][j] \leftarrow C_{min}$
34: $Partition[i][j] \leftarrow p$
35: $Finish[i][j] \leftarrow finish$
36: $EST[j+1, i..j] \leftarrow est$
37: **end for**

Algorithm 12 Checks the validity of a finish placement

```
 1: procedure ValidHelp(node_1, node_2, left, right)
 2:     lca_1l = LCA(node_1, left)
 3:     lca_12 = LCA(node_1, node_2)
 4:     lca_2r = LCA(node_2, right)
 5:     d_1l = lca_1l.depth
 6:     d_12 = lca_12.depth
 7:     d_2r = lca_2r.depth
 8:     if (d_1l > d_12) ∨ (d_2r > d_12) then
 9:         return false
10:     end if
11:     return true
12: end procedure
13: procedure Valid(i, j)
14:     return ValidHelp(node[i], node[j], node[i − 1], node[j + 1])
15: end procedure
```

first partition to the second partition. In this case a finish is not required. Figure 4.13 shows the computation of EST.

Algorithm 11 shows the dynamic programming method used to compute the optimal finish placement. $Opt[i][j]$ holds optimal cost for the subproblem $i..j$. To help us keep track of how to construct an optimal solution, we save the optimal partitioning point of $i..j$ in $Partition[i][j]$. $Finish[i][j]$ keeps track of whether a finish is required around the block $i..k$.

The loop in lines 14-32 iterates through each of the possible partitioning points and finds the optimal one. Lines 16-20 handle the case where a finish is not required and lines 21-25 handle the case where a finish is required. Note that it considers only values of k for which (i, k) has a valid static finish placement. Procedure VALID(i, j) in Algorithm 12 checks the validity of a finish placement. (i, j) is a valid finish placement, only if there exists a point in the DPST where we can introduce a finish node, whose descendents include the

nodes $i..j$, but do not include nodes $i-1$ or $j+1$. The array *node* used in VALID contains all the nodes in the dependence graph, ordered from left to right.

Algorithm 11 computes the optimal solution in $O(n^3 \times d)$ time, by taking advantage of the overlapping-subproblems property, where d represents the height of the subtree rooted at LCA. There are only $\Theta(n^2)$ different subproblems in total. The solution for each of these subproblems is computed once in $O(n \times d)$ time.

Algorithm 13 Find the set of finishes from the output of Algorithm 11

Require: Partition, Finish
Ensure: Set of finishes

```
1:  procedure FIND(begin, end)
2:      if begin = end then
3:          return ∅
4:      end if
5:      p = Partition[begin][end]
6:      left = FIND(begin, p)
7:      right = FIND(p, end)
8:      if Finish[begin][end] = true then
9:          return {(begin, p)} ∪ left ∪ right
10:     else
11:         return left ∪ right
12:     end if
13: end procedure
14: return FIND(1, n)
```

Algorithm 11 computes the optimal cost for a finish placement, which satisfies all the dependences in the dependence graph. It does not directly show the partitioning points. This information can be easily determined from the arrays *Partition* and $Finish$. 1..(*Partition*[1][n]) and (*Partition*[1][n] + 1)..n are the two subproblems used to compute the optimal solution for 1..n. The value of $Finish$[1][n] determines whether a finish is required around 1..*Partition*[1][n]. The finish placements for all the subproblems can be computed

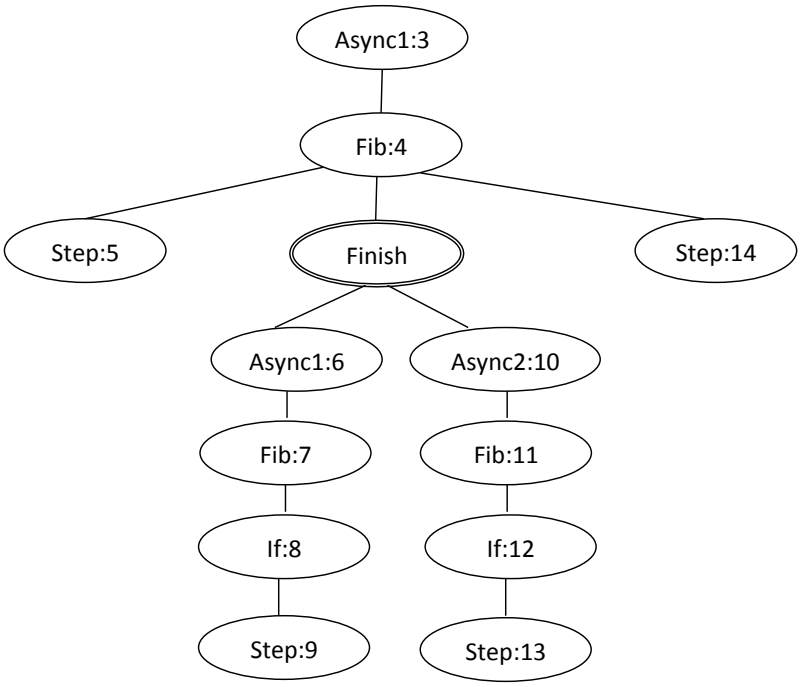

Figure 4.14 : Subtree in Figure 4.10 after inserting finish

recursively as presented in Algorithm 13.

The next step is to find the exact location in the S-DPST where the finish nodes must be inserted. For each $(i, j) \in FinishSet$, this is found by a bottom-up traversal of the S-DPST, where we find the highest node in the S-DPST where we can introduce a new $finish$ node as the ancestor of $i..j$, but is not an ancestor of $i-1$ or $j+1$.

Example: Lets now consider the application of Algorithm 11 on the dependence graph in Figure 4.11. The set of vertices for the graph is $V = \{1, 2, 3, 4\}$ and the set of edges is $E = \{(2, 4), (3, 4)\}$, where vertex 1 refers to *Step:5*, 2 refers to *Async1:6*, 3 refers to *Async2:10* and 4 refers to *Step:14*. Lets assume $t_1 = 5, t_2 = 20, t_3 = 15, t_4 = 5$. The application of Algorithm 11 would infer a finish placement of $\{(2, 3)\}$. Figure 4.14 shows the new subtree after the insertion of finish node.

4.5.3 Correctness and Optimality

The dependence graph constructed using the method in Section 4.5.1 models the data races as dependences between the children of the NS-LCA. The dynamic programming algorithm given in Section 4.5.2 ensures that new finish nodes are inserted such that the source and sink of a data race may not happen in parallel.

Theorem 3. *Consider two leaf nodes S1 and S2 in a S-DPST, where S1 ≠ S2 and S1 is to the left of S2. Let N be the node denoting the non-scope least common ancestor of S1 and S2. Let node A be the ancestor of S1 that is the non-scope child of N. Then, S1 and S2 can execute in parallel if and only if A is an async node.*

Theorem 3 is an extension of a result from [27] which gives the necessary and sufficient condition for 2 steps to execute in parallel. To resolve a data race between 2 steps S1 and S2, we need to introduce a finish node, F in the S-DPST, such that

- F is the non-scope child of the NS-LCA, N of S1 and S2
- F is an ancestor of S1 but not an ancestor of S2.

Theorem 4. *Consider a node L in S-DPST. Let $G = (V, E)$ be a directed acyclic graph (DAG) in which nodes $V = \{1, .., n\}$ represent the dynamic children of L and E represent the set of data races whose dynamic LCA is L. Algorithm 11 finds an optimal set of finish placement which resolves all the data races represented by E.*

Proof. By induction on s in Algorithm 11. At the start of s iteration of the loop in line 11-36, optimal solutions for all subproblems of size $s - 1$ have been computed. The next iteration of the loop computes the optimal solutions for all subproblems of size s. □

4.6 Static Finish Placement

In this section, we present the algorithm for finding a static finish placement from the dynamic finish placements computed using the algorithms presented in Section 4.5.

4.6.1 Algorithm

Dynamic finish placement algorithm finds the finish placements required to resolve the data races at a single NS-LCA. The choice of finish placements at a single NS-LCA can have impact on the rest of S-DPST and the input program. The static finish placement algorithm handles these issues, by propagating these finish placements to the rest of the S-DPST and the input program. The complete steps in static finish placement algorithm are given below.

1. Find the NS-LCA of the source and sink of each of the data races in the S-DPST
2. Group all the data races which have a common NS-LCA
3. For each unique NS-LCA , *N*
   (a) Reduce the subtree rooted at *N* to a DAG as described in Section 4.5.1.
   (b) Find the set of finish nodes needed to fix the data races with NS-LCA, *N* using the dynamic programming algorithm presented in Section 4.5.2.
   (c) Find the locations in the S-DPST where a finish needs to be inserted by a bottom-up traversal.
   (d) Insert the finish statements in the input program and update the S-DPST with the new finish nodes
   (e) Remove the data races which are fixed by the insertion of the finish nodes.
   (f) Update the data races for which the NS-LCA have changed due to the insertion of new finish nodes.

```
1  static class BoxInteger {
2    public int v;
3  }
4
5  void fib (BoxInteger ret, int n) {
6    if ( n < 2) {
7      ret.v = n;
8      return;
9    }
10   final BoxInteger X = new BoxInteger();
11   final BoxInteger Y = new BoxInteger();
12   finish { // Newly inserted finish
13     async fib (X, n-1); // Async1
14     async fib (Y, n-2); // Async2
15   }
16   ret.v = X.v + Y.v;
17 }
18 ....
19 async fib(3); // Async0
```

Figure 4.15 : Fibonacci program from Figure 4.8 after finish insertion

The algorithm iterates through each of the unique NS-LCAs and finds the set of dynamic finish placements needed to fix the data races. These finish placements are then mapped to the input program. The S-DPST is then updated with the new set of finish statements. Note that a finish placement at one NS-LCA may lead to the insertion of finish nodes in subtrees rooted at other NS-LCAs. At the termination of the algorithm, we have a program free of data races for the given input.

Example: The subtree rooted at *Async1:3* after dynamic finish placement is shown in Figure 4.14. The finish placement in this subtree are then propagated to the rest of the S-DPST. This will introduce another Finish node as the child of *Fib:1* and as the parent of *Async1:3* and *Async2:15* in Figure 4.9. At this point all the data races have been fixed and the program with the newly introduced finish statement is given in Figure 4.15.

4.6.2 Correctness & Conditions for Optimality

The static finish placement algorithm iterates through each unique NS-LCA and finds the finish placement for a subset of all the data races, instead of finding a finish placement which covers all the data races. The following theorem gives the intuition behind this approach.

Theorem 5. *Consider a program P with n data races* $\{D_1,..,D_n\}$. *Let* $\{L_1, .., Ln\}$ *be the non-scope least common ancestors (NS-LCA) of the nodes corresponding to the steps involved in the data race in the S-DPST. A finish node in the S-DPST which resolves a data race* D_i *may resolve a data race* D_j *only if* $L_i = L_j$.

From Theorem 5, it follows that the problem of global finish placement could be solved by grouping data races by their NS-LCA and solving the problem locally. If the subproblems at different NS-LCAs are non-overlapping this leads to an optimal solution. If the subproblems are overlapping, the decisions made in the solution of earlier subproblems may lead to non-optimal solutions for later subproblems. In most real world programs, we observed that the solutions required at different NS-LCAs are identical or non-overlapping, which leads to a global optimal solution.

4.7 Experimental Results

In this section, we present experimental results for our test-driven repair tool. The different components of the tool shown in Figure 4.6 were implemented as follows. Programs were instrumented for race detection, S-DPST construction and computation of execution time of steps during a byte-level transformation pass on HJ's Parallel Intermediate Representation (PIR) [34]. The data race detector (modified version of ESP-Bags) was implemented as a Java library for detecting data races in HJ programs containing async and finish con-

Source	Benchmark	Description	Input Size (Repair)	Input Size (Performance)
HJ Bench	Fibonacci	Compute *n*th Fibonacci number	16	40
	Quicksort	Quicksort	1,000	100,000,000
	Mergesort	Mergesort	1,000	100,000,000
	Spanning Tree	Compute spanning tree of an undirected graph	nodes = 200, neighbors = 4	nodes = 1,000,000, neighbors = 100
BOTS	Nqueens	N Queens problem	6	13
JGF	Series	Fourier coefficient analysis	rows = 25	rows=100,000
	SOR	Successive over-relaxation	size =100, iters = 1	size = 6,000, iters = 100
	Crypt	IDEA encryption	3,000	50,000,000
	Sparse	Sparse matrix multiplication	100	2,500,000
	LUFact	LU Factorization	25×25	1000×1000
Shootout	FannKuch	Indexed-access to tiny integer-sequence	6	12
	Mandelbrot	Generate Mandelbrot set portable bitmap	50	10,000

Table 4.1 : List of benchmarks evaluated

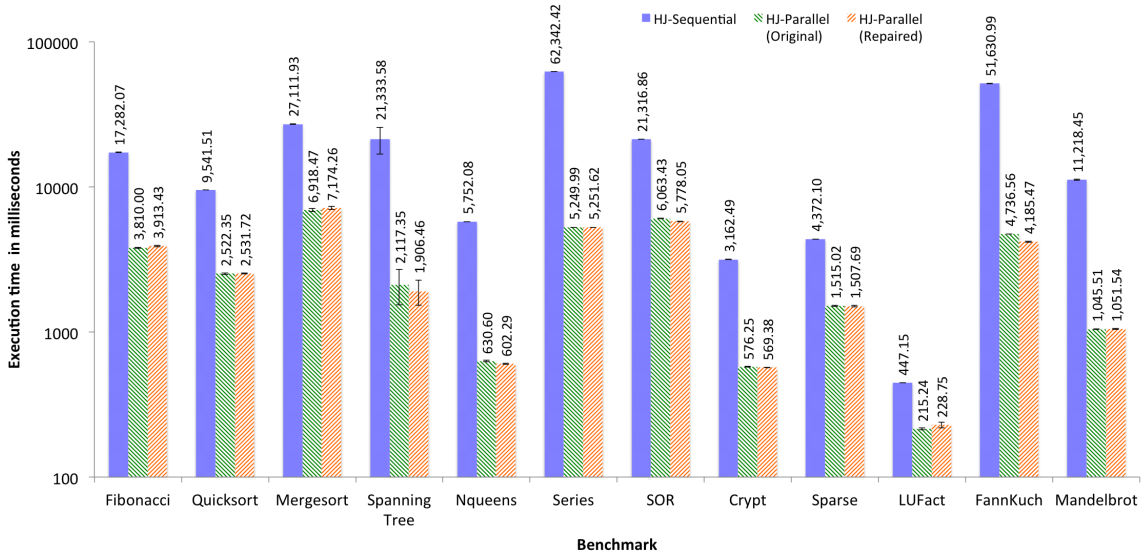

Figure 4.16 : Average execution times in milliseconds and 95% confidence interval of 30 runs for sequential, original parallel, and repaired parallel versions, for "Performance" input size. Parallel versions were run on 12 cores.

Benchmark	HJ-Seq (millisecs)	RaceDet Time (millisecs)	#S-DPST Nodes	Number of Data Races	Repair Time (secs)
Fibonacci	17.41	229.03	17,568	3,192	4.77
Quicksort	11.01	554.50	54,857	17,727	21.35
Mergesort	6.08	827.04	120,688	424,436	647.43
Spanning Tree	9.15	360.00	37,410	3,261	11.21
Nqueens	2.67	283.75	32,434	4	6.02
Series	37.07	559.30	98,226	6	45.30
SOR	26.01	275.11	59,422	19,110	21.11
Crypt	32.10	603.05	30,596	3,375	5.44
Sparse	13.52	223.02	46,561	260	15.21
LUFact	13.01	299.32	24,430	99,563	56.27
FannKuch	3.41	2,853.18	19,785	7	3.09
Mandelbrot	10.62	430.45	271,354	100	55.66

Table 4.2 : Time for program repair. Input size: Repair

structs, and generating trace files containing all the data races detected. The dynamic and static finish placement algorithms were implemented as subsequent compiler passes in the HJ compiler, that read the trace files generated by the data race detector, update the PIR representation of the program, and then output the source code positions where additional finish constructs should be inserted.

Our experiments were conducted on a 12-core Intel Westmere 2.83 GHz system with 48 GB memory, running Red Hat Enterprise Linux Server release 6.2, and Sun Hotspot JDK 1.7. To reduce the impact of JIT compilation, garbage collection and other JVM services, we report the mean execution time measured in 30 runs repeated in the same JVM instance for each data point.

We evaluated the repair tool on a suite of 12 task-parallel benchmarks listed in Table 4.1. The fourth column of Table 4.1 shows the input sizes used for repair mode (which includes data race detection and S-DPST construction). The fifth column shows the input size used for performance evaluation of the repaired programs.

4.7.1 Repairing Programs

To evaluate our tool, we performed the following test. We removed all finish statements from the benchmarks in Table 4.1, and then ran the repair tool on each of the resulting buggy programs. For these programs, a single iteration of the tool with a single test case (input size shown in column 4 of Table 4.1) was sufficient to obtain a repair that satisfied all task dependences. Further, the tool's repair (insertion of finish statements) resulted in parallel performance that was almost identical to that of the original benchmark in each case. Figure 4.16 shows the execution times for the sequential, original parallel, and repaired parallel versions of each benchmark when executed on 12 cores. A visual inspection confirmed that the tool was able to insert finish statements so as to obtain comparable parallelism to that created by the experts who wrote the original benchmarks.

Benchmark	**Data Race Detection 1 (millisecs)**		**Repair Time (secs)**		**Data Race Detection 2**	**Total Time (secs)**	
	SRW	**MRW**	**SRW**	**MRW**	**SRW Only**	**SRW**	**MRW**
Fibonacci	213.13	229.03	4.71	4.77	170.12	5.09	5.00
Quicksort	411.52	554.50	21.11	21.35	386.35	21.91	21.90
Mergesort	623.12	827.04	155.91	647.43	557.13	157.09	648.26
Spanning Tree	291.00	360.00	10.03	11.21	138.67	10.46	11.57
Nqueens	255.12	283.75	6.11	6.02	253.44	6.62	6.30
Series	560.03	559.30	45.03	45.30	526.47	46.12	45.86
SOR	220.31	275.11	21.01	21.11	198.05	21.43	21.39
Crypt	374.52	603.05	5.36	5.44	314.15	6.05	6.04
Sparse	171.11	223.02	15.11	15.21	170.21	15.45	15.43
LUFact	216.21	299.32	56.41	56.27	157.63	56.78	56.57
FannKuch	203.51	2,853.18	3.06	3.09	204.23	3.47	5.94
Mandelbrot	426.31	430.45	55.13	55.66	382.36	55.94	56.09

Table 4.3 : Comparison of SRW ESP-Bags and MRW ESP-Bags. Input size: Repair

Benchmark	SRW ESP-Bags	MRW ESP-Bags
Fibonacci	3,192	3,192
Quicksort	1,780	17,727
Mergesort	39,684	424,436
Spanning Tree	397	3,261
Nqueens	4	4
Series	6	6
SOR	19,110	19,110
Crypt	3,375	3,375
Sparse	100	260
LUFact	99,563	99,563
FannKuch	7	7
Mandelbrot	100	100

Table 4.4 : Number of data races detected by SRW ESP-Bags and MRW ESP-Bags. Input size: Repair

4.7.2 Time for Program Repair

Table 4.2 shows the time to repair each of the programs using input sizes given in column 4 of Table 4.1. HJ-Seq is the sequential runtime of the benchmarks. The third column shows the time taken for data race detection and S-DPST construction. The fourth and fifth column gives the number of S-DPST nodes and the number of data races reported by MRW ESP-Bags algorithm respectively. Repair time is the time taken for static and dynamic finish placements. We observed that, as the number of S-DPST nodes and the number of data races increases, the time taken for the program repair also increases. This is because the time to repair is dominated by the time taken to read the trace files generated by data race detector and building the S-DPST. Although the worst-case time complexity for dynamic finish placement is $O(n^3 \times d)$, the time taken in practice is very small because n and d are small in practice.

4.7.3 Comparison of SRW and MRW ESP-Bags

In this section, we compare the overheads and results produced by the two data race detectors. Our tool uses the MRW ESP-Bags algorithm for data race detection by default. This guarantees that all data races are reported in a single run, for a given test case. Using the SRW ESP-Bags algorithm may require multiple iterations of the tool for the same test case to ensure that the program does not contain data races that were not identified and fixed in a prior iteration. For the benchmarks used in this work, only two SRW iterations were needed in each case (one for repair, and one to confirm that no data races remain). The main reason to consider SRW is that each SRW iteration may generate smaller trace files than that generated by a single MRW run, and smaller trace files directly result in a smaller memory footprint and execution time for repair. This hypothesis is confirmed in Table 4.4, which compares the number of data races detected by a single run of the SRW and MRW algorithms.

Table 4.3 compares the total repair time for the MRW and SRW algorithms (including two runs in the SRW case). We see that the execution times are comparable in many cases, but there is a big difference for mergesort for which MRW's repair time is more than 4× slower than SRW's repair time. This can be explained by the large absolute number of reported data races for the MRW algorithm in Table 4.4, for the mergesort benchmark. The time for synthesizing a correct program from MRW race reports for mergesort is higher due to the cost of reading the large traces and adding the race edges to our internal representation.

4.7.4 Student Homework Evaluation

We also used our repair tool to evaluate student homework submissions as part of an undergraduate course on parallel computing. The assignment for the students was to per-

form a manual repair on a parallel quicksort program i.e., to insert finish statements with maximal parallelism while ensuring that no data races remain. The initial version of the program contained async statements, but no finish statements. We then evaluated the student submissions against the finish statements automatically generated by the tool. Out of 59 student submissions, 5 submissions still had data races, 29 submissions were over-synchronized (i.e., had reduced parallelism), and 25 submissions matched the output from our repair tool. We believe that our repair tool will be a valuable aid for future courses on parallel programming, especially in on-line offerings where automated feedback is critical to improve the learning experience for students.

4.8 Related Work

The last decade has seen much activity in repair and synthesis of programs, including concurrent programs. Vechev, Yahav, and Yorsh [40] use abstract interpretation to analyze atomicity violations and abstraction-guided synthesis to introduce minimal critical regions into a program to eliminate atomicity violations by restricting potential interleavings. A prototype based on this approach has been demonstrated on program fragments consisting of tens of lines. Cerny et al. [41] present a method for using a performance model to guide refinement of a non-deterministic partial program into one that is both race and deadlock free. They use manually derived abstractions of programs and the SPIN model checker to reason about transitions. This system was also applied to tens of lines of code. Raychev et al. [8] used abstract interpretation to compute an over-approximation of the possible program behaviors. If the over-approximation is not free of conflicts, the algorithm synthesizes a repair that enforces conflict freedom. Solar-Lezama et al. [42] synthesizes concurrent data structures from high-level "sketches".

The most closely related work on concurrent program repair is described in a pair of

papers by Jin et al. [9, 10]. In a 2011 paper [9], they describe AFix—a system for detection and repair of concurrency bugs resulting from single-variable atomicity violations. Their system detects atomicity violations, designs a repair by adding a critical section protected by a lock to prevent problematic interleavings. They identify nodes on all intra-procedural paths between two instructions that need to be protected and insert lock acquire and releases at the boundaries as one enters and exits the protected region. Critical sections added by AFix need not be block structured. Finally, AFix merges overlapping critical regions introduced by one or more repairs. A 2012 paper [10] describes CFix—a more comprehensive system for detecting and repairing several kinds of concurrency bugs, including atomicity violations, ordering violations, data races, and def-use errors. CFix relies on several different existing bug detectors to detect different types of concurrency bugs. They fix bugs by adding ordering and/or mutual exclusion. For mutual exclusion, they rely on their earlier work on AFix. With respect to [9] and [10], which avoids atomicity violations by ordering accesses: we add finish constructs which (a) eliminate the observed races, and (b) ensure that the semantics of the accesses in the parallel program is the same as in the sequential version. [9] and [10] merely ensure that the accesses aren't concurrent by adding mutual exclusion or pairwise ordering.

Kelk et al. uses a combination of genetic algorithms and program optimization to automatically repair concurrent Java programs [43]. The usefulness of genetic algorithms in program repair was previously demonstrated by Le Goues et al. [44]. Other relevant program repair Griesmayer et al. [45], which uses a method based on model checking to repair boolean programs, and Logozzo and Ball [46], which describes a system for reasoning about .NET software that uses abstract interpretation to suggest program repairs. However, the focus in these approaches is not on concurrency bugs.

4.9 Summary and Future Work

We presented a tool for test-driven repair of data races in structured parallel programs. The tool identifies static points in the program where additional synchronization is required to fix data races for a given set of input test cases. These static points obey the scoping constraints in the input program, and are inserted with the goal of maximizing parallelism. We evaluated an implementation of our tool on a wide variety of benchmarks which require different synchronization patterns. Our experimental results indicate that the tool is effective — for all benchmarks, the tool was able to insert finishes to avoid data races and maximize parallelism. Further, the evaluation of the tool on student homeworks shows the potential for such tools in future offerings of parallel programming courses, especially in online versions.

There are multiple possibilities for future work. One of the limitations of the tool is in analyzing long-running programs, which may lead to the creation of S-DPSTs that do not fit in memory. One possible extension for the future is to enable garbage collection of parts of the S-DPST that do not exhibit race conditions. Some other directions for future work include generation of context sensitive finishes (where a finish is conditionally executed only in contexts where a data race is observed), and test coverage analysis to evaluate the suitability of a given set of test cases for program repair.

Chapter 5

Automatic Parallelization via Synthesis of Futures

Parallelizing programs to effectively utilize multicore architectures is a major challenge facing application developers and domain experts. In this work, we introduce a novel approach for automatically parallelizing pure method calls using futures as the primary parallel construct. A method is pure [47, 48] if it (or any method that it calls) does not mutate any object in the program state that exists before the method is invoked. However, a pure method is permitted to mutate objects that are allocated during its execution and return a newly constructed object as the result. Further, a pure method is allowed to read global state that may be later mutated by the method's caller.

Futures are traditionally used for enabling functional style parallelism, and therefore, are a natural fit for parallelizing the execution of pure method calls. They also have the advantage that references to future objects can be copied without waiting for the future tasks to have completed, thereby exposing more parallelism than in imperative-style task parallel constructs. Finally, the synchronization patterns that can be expressed by structured fork-join models (such as OpenMP's task parallelism [30], Cilk's spawn-sync [29] parallelism) are inherently limited to series-parallel computation graphs, while futures can be used to generate any arbitrary computation graph.

As an example, consider the program from the Computer Language Benchmarks Game [49] in Figure 5.1, which constructs a binary tree using method `bottomUpTree` and then performs a traversal of the tree using method `itemCheck`. The program after parallelization using the approach presented in this chapter (using a test input that constructs a tree

```
1  class TreeNode {
2    private TreeNode left, right;
3    ...
4    TreeNode bottomUpTree(int item, int depth){
5      if (depth > 0) {
6        TreeNode l = bottomUpTree(2*item-1, depth-1);
7        TreeNode r = bottomUpTree(2*item, depth-1);
8        return new TreeNode(l, r, item);
9      } else {
10       return new TreeNode(item);
11     }
12   }
13   ...
14   int itemCheck() {
15     ...
16     return item + left.itemCheck() - right.itemCheck();
17   }
18 }
```

Figure 5.1 : Sequential binary tree program [49] from computer language benchmarks game

with height = 14) is shown in Figure 5.2. The parallelization algorithm made the following changes to the program: 1) The construction of the tree is performed as future tasks, if the current depth* is greater than or equal to a certain threshold; 2) The types of the fields `left` and `right` and variables `l` and `r` are changed from `TreeNode` to `mayfuture<TreeNode>`, where `mayfuture<T>` may refer to a `future<T>` object or an object of type `T`; 3) A new constructor is added to the `TreeNode` class which accepts `mayfuture<TreeNode>` as its first and second arguments; and, 4) `get()` calls are inserted before the result of a future/-mayfuture object is used. Another important aspect of our approach is that even though both `bottomUpTree` and `itemCheck` are pure methods, our approach only parallelizes the execution of the `bottomUpTree` method since the work performed by `itemCheck` is not

*This benchmark uses a parameter named depth for what might usually be considered to be the height of the node. For example, the depth parameter is zero for all leaf nodes.

profitable for parallelization. Compared to using imperative-style task parallel constructs, this program has more parallelism because it performs a `get()` only when the result of the future task is used in line 28 of Figure 5.2. If the same program is parallelized using imperative-style constructs such as `spawn-sync` or `async-finish`, the parallelized program will require synchronization to ensure that the tasks created in line 9 and line 15 complete before the constructor invocation in line 20 of Figure 5.2.

The rest of the chapter is organized as follows. In Section 5.1, we formulate the problem that we are solving. Section 5.2 presents our contributions. Section 5.3 presents an overview of our approach. Sections 5.4-5.7 describe the technical details of our solution. Section 5.8 contains our experiment results. Section 5.9 discusses related work, and Section 5.10 summarizes our conclusions.

5.1 Problem Statement

Problem 3. (Synthesis) Given a sequential or parallel program, P in which selected expressions are annotated as `async` for asynchronous execution as futures, and test inputs, ψ_1, ψ_2,..,ψ_n, output a program with increased parallelism by inserting `future` operations, type conversions, and conditional threshold expressions as needed for the `async` expressions, such that

1. for all inputs, the synthesized program does not exhibit any data races on the result objects of `async` expressions,

2. the synthesized program must express the maximum parallelism available from the inserted futures (without any loss of parallelism due to copying of future references),

3. the synthesized program must minimize the loss in type information for future objects, i.e., must make the declared types of futures as precise as possible, thereby

```
1  class TreeNode {
2    private mayfuture<TreeNode> left, right;
3    static int THRESHOLD = 12;
4    ...
5    TreeNode bottomUpTree(int item, int depth){
6      if (depth > 0) {
7        mayfuture<TreeNode> l, r;
8        if (depth-1 >= THRESHOLD) {
9          l = async<TreeNode> {
10           return bottomUpTree(2*item-1, depth-1);
11         }
12       } else
13         l = bottomUpTree(2*item-1, depth-1);
14       if (depth-1 >= THRESHOLD) {
15         r = async<TreeNode> {
16           return bottomUpTree(2*item, depth-1);
17         }
18       } else
19         r = bottomUpTree(2*item, depth-1);
20       return new TreeNode(l, r, item);
21     } else {
22       return new TreeNode(item);
23     }
24   }
25   ...
26   int itemCheck() {
27     ...
28     return item + left.get().itemCheck() -
           right.get().itemCheck();
29   }
30 }
```

Figure 5.2 : Binary tree program from Figure 5.1 after parallelization using the approach presented in this chapter. Our implementation does the transformations on Java bytecode. The equivalent source code is shown here.

minimizing the number of cast and instanceof operations inserted for future objects,

4. the synthesized program must clone objects read by future tasks as required to preserve *anti dependences* in the input program, and

5. the synthesized program must execute an expression as future task only if it is profitable to do so based on execution profile information collected from ψ_1,..,ψ_n. □

5.2 Contributions

The main contributions of this work are as follows:

- A static analysis algorithm for *future synthesis* that can be used to synthesize a parallel program with future objects, their type declarations, and async expressions. Our approach synthesizes object clones when needed, and generates more precise type information for future objects, compared to future synthesis algorithms reported in past work for manual parallelization.

- A *parallelism benefit analysis* algorithm, which determines the profitability of executing a method call as a future task. The analysis is based on execution profile information collected from multiple test inputs.

- An algorithm to synthesize *threshold conditional expressions*, which determine dynamically whether a specific method call should be executed sequentially or in parallel.

- These algorithms have been implemented as analyses and transformations to generate parallel Habanero Java (HJ) [16] code from sequential Java code, and evaluated on a range of benchmark programs. When using 8 processor cores, the evaluation shows that our approach can provide significant parallel speedups of up to 7.4× (geometric mean of 3.69×).

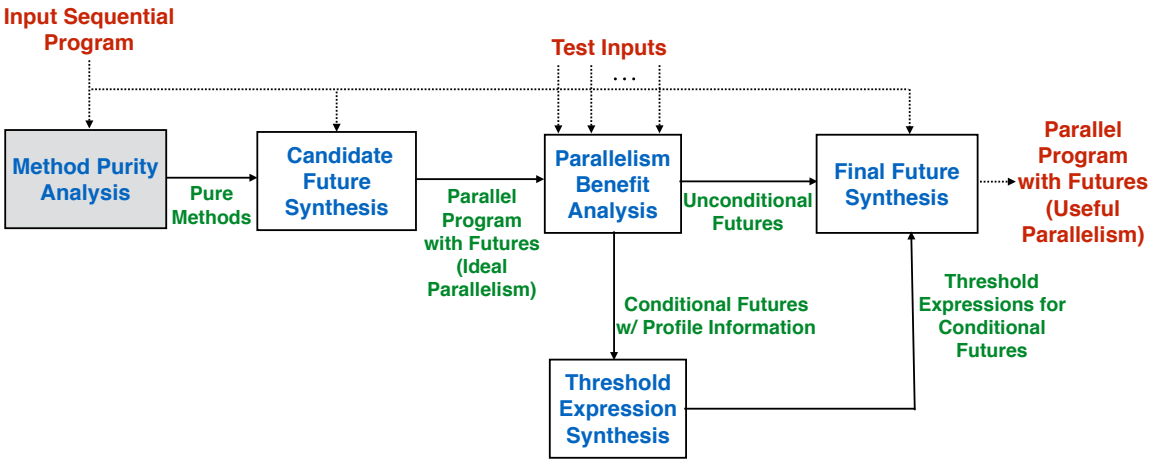

Figure 5.3 : High level view of our approach. The dotted lines represent user inputs and outputs. The grey box (Method Purity Analysis) represents past work leveraged by our approach, whereas the other boxes represent new contributions.

5.3 Overview of Approach

A high level view of our approach is given in Figure 5.3. The five main steps in automatic parallelization of eligible method calls are as follows:

1. **Method Purity Analysis**: The first step in our approach is the identification of pure methods. Our implementation uses past work (ReImInfer [48]) on automatic purity analysis to identify pure methods in Java programs, but can also be applied to programs in which methods are annotated as pure by the programmer.

2. **Candidate Future Synthesis (CFS)**: Our tool annotates calls to a subset of the pure methods identified by ReImInfer as async expressions. (The subset focuses on methods containing iterative/recursive subcomputations, so as not to overwhelm later instrumentation phases with trivial and unprofitable candidates for execution as future tasks.) Next, we generate a parallel program by synthesizing futures from the async expressions. The synthesis algorithm is presented in Section 5.4 and involves two

steps: 1) inter-procedural future analysis, which determines the locations in the input program where a future object may be accessed, as well as the types of the future objects and 2) future transformations, which changes the types of future objects and inserts future get operations.

3. **Parallelism Benefit Analysis (PBA)**: Once we have a parallel program with futures, we construct Weighted Computation Graphs (WCGs) for the program for each of the given test inputs. (The choice of test inputs only impacts the performance, not the correctness, of parallelization.) The weights of the nodes in the WCG represent the work done by each of the steps and the overheads of task creation, task termination and synchronization operations. The weighted computation graphs are then analyzed to identify tasks that provide benefit from parallelization. Based on the analysis results, each method call site is classified as *serial*, *parallel* or *conditional parallel*. The parallelism benefit analysis algorithm is presented in Section 5.5.

4. **Threshold Expression Synthesis (TES)**: For call sites that are identified as conditional parallel by PBA, this step synthesizes an expression that enables conditional parallel execution of method invocations. The threshold expression identifies a subset of method invocations at the call site, for which the work done by the method is greater than a certain threshold, which we refer to as *sequential threshold*. The threshold expression synthesis algorithm is presented in Section 5.6.

5. **Final Future Synthesis**: In the last step, we generate parallel code from the input sequential program based on the analysis done by the previous steps. This involves cloning inputs when needed, annotation of parallel call sites as async expressions, and conditional annotation of conditional parallel call sites as async expressions. The final future synthesis step is described in Section 5.7.

```
1  class TreeNode {
2    private TreeNode left, right;
3    ...
4    TreeNode bottomUpTree(int item, int depth){
5      if (depth > 0) {
6        TreeNode l = async bottomUpTree(2*item-1, depth-1);
7        TreeNode r = async bottomUpTree(2*item, depth-1);
8        return new TreeNode(l, r, item);
9      } else {
10       return new TreeNode(item);
11     }
12   }
13   ...
14   int itemCheck() {
15     ...
16     return item + async left.itemCheck() - async
           right.itemCheck();
17   }
18 }
```

Figure 5.4 : Binary tree program from Figure 5.1 after method purity analysis and `async` expression annotation

5.4 Candidate Future Synthesis

In this section, we present our approach for synthesizing futures in a program annotated with `async` expressions. The `async` expressions that serve as the source for synthesis are inserted based on the output of method purity analysis. The program from Figure 5.1 after async expression annotation using method purity analysis is shown in Figure 5.4. The two functions `bottomUpTree` and `itemCheck` do not cause any side-effects, and their calls are therefore marked as `async` expressions. The async expression annotated program is then passed as input for future synthesis.

The synthesis process involves the following steps:

1. Replacing `async` expressions by typed `future` expressions.

2. Identifying inputs that need to be cloned in the final future synthesis step.

3. Analyzing the whole program and modifying the types of variables and fields that can refer to a future object. Their type is changed from `T` to `future<T>`, if the variable or field *must* refer to a future at all program points where the variable/field is accessed. If the variable or field *may* refer to a future, the type is changed to `mayfuture<T>`. This is in contrast to past work [50] in which all future variables/fields were declared with an `Object` type in both cases, and cast operations were inserted whenever they needed to be accessed as future objects (Section 5.4.1).

4. Identify methods that perform non-constant amount of work and are candidates for asynchronous execution (Section 5.4.3).

5. Insertion of calls to `get()` before the result of a future task is used, along with `instanceof` checks when needed for `mayfuture` objects.

Section 5.4.1 presents an inter-procedural data flow analysis that identifies the locations in the program where a future object may be accessed, as well as the types of the future objects. Section 5.4.2 contains the details of the transformation step based on the result of the data flow analysis. Section 5.4.3 presents an algorithm to identify methods that perform non-constant amount of work and are candidates for asynchronous execution, and Section 5.4.4 discusses how our transformation preserves the data dependences in the input program.

5.4.1 Inter-procedural Future Analysis

As indicated earlier, past work on future synthesis did not analyze the types of the future objects. Instead they set the type of all future objects to `Object` in Java programs as in [50], or worked with untyped programs as in MultiLisp [18]. In contrast, our work

Statement	Data flow function
`t := async e;`	$\lambda Y.(Y \cup \{\texttt{t}\})$
`x := new` τ;	$\lambda Y.(Y - \{\texttt{x}\})$
`x.f := t;`	$\lambda Y.(\text{if } \texttt{t} \in Y \text{ then } Y \cup \{\texttt{f}\} \text{ else } Y)$
`t := x.f;`	$\lambda Y.(\text{if } \texttt{f} \in Y \text{ then } Y \cup \{\texttt{t}\} \text{ else } Y)$
`x := t;`	$\lambda Y.(\text{if } \texttt{t} \in Y \text{ then } Y \cup \{\texttt{x}\} \text{ else } Y)$

Figure 5.5 : Examples of normal flow function for computing $\mathcal{M}$ (may-be-future)

attempts to determine the most precise type information for future objects as possible. We do so by using the IFDS [51] algorithm as the foundation for solving the future-analysis problem. IFDS can be used to compute the meet-over-all-valid-paths solution for all inter-procedural, finite, distributive subset problems. In the IFDS framework, the input program is represented as a directed graph called the *super graph*. The super graph consists of a collection of flow graphs, one for each procedure in the input program. The analysis is solved in polynomial time by reducing it to a graph-reachability problem.

The goal of the analysis is to find the set of variables and fields that *may/must* refer to a future object during all its accesses in the program, as well as the most precise type that can be identified statically for the future object, where a future object is the result of any expression of the form `async expr`. Our analysis finds the solution to two problems: *may-be-future* and *must-be-future*. The solution to the *must-be-future* problem is computed as the complement of the solution to the *may-not-be-future* problem. At any given statement, $\mathcal{M}$ represents the set of variables and fields that *may-be-future* and $\mathcal{N}$ represents the set of variables and fields that *may-not-be-future*. The $\mathcal{M}$ and $\mathcal{N}$ sets need not be disjoint; in fact, the most conservative solution is to simply state that all variables and fields belong to both sets.

Normal flow functions are applied to all statements that contain neither call nor return

statements. Examples of normal flow functions for computing $\mathcal{M}$ are given in Figure 5.5. An `async` expression generates a future object, whereas the result of a `new` expression cannot be a future object. The other three kinds of assignment statements may propagate a reference to a future object from the right hand side to the left hand side of the assignment. Our analysis builds on type-based alias analysis [52], and its precision can be improved by incorporating more complex alias analysis algorithms into the framework. However, more precise whole program alias analysis could be a scalability bottleneck for large applications. One drawback of using type-based alias analysis is that all elements of arrays of type τ in the program will be marked as `may-future` if any future object is stored into an array of type τ. We assume a universe *Var* of variable names, F of field names, $\mathcal{T}$ of class names, where $\mathtt{x,t,r,a_0,...a_n,p_0,...p_n} \in \mathit{Var}$, $\mathtt{f} \in F$, and $\tau \in \mathcal{T}$.

Call flow functions handle the data flow from a method call statement into the called procedure. The context change from the body of the caller to the body of the callee is modeled by replacing references to actual parameters, $\mathtt{a_i}$ by references to formal parameters, $\mathtt{p_i}$. All fields that may be a future at the call site may also be a future at the start node of the callee. The call flow function at call site, `c` is shown below, where $\mathtt{a_i} \xrightarrow{\mathtt{c}} \mathtt{p_i}$ represents the binding of the actual parameter $\mathtt{a_i}$ to the formal parameter $\mathtt{p_i}$.

$$\lambda Y.\{\forall \mathtt{i}, \mathtt{p_i} \mid \mathtt{a_i} \in Y \wedge \mathtt{a_i} \xrightarrow{\mathtt{c}} \mathtt{p_i}\} \cup \{\mathtt{f} \mid \mathtt{f} \in Y \wedge \mathtt{f} \in F\}$$

At a return statement, the data flow set at the callee is mapped back to the caller by the *return flow function*. The return value, `r` in the callee is mapped to the left hand side of the assignment in the caller. All fields that may be a future in the callee may also be a future at the call site. Return flow function at call site, `c` is given below, where $\mathtt{r} \xrightarrow{\mathtt{c}} \mathtt{x}$ represents the binding of the return value, `r` in the callee to the variable `x` in the caller.

$$\lambda Y.\{\mathtt{x} \mid \mathtt{r} \in Y \wedge \mathtt{r} \xrightarrow{\mathtt{c}} \mathtt{x}\} \cup \{\mathtt{f} \mid \mathtt{f} \in Y \wedge \mathtt{f} \in F\}$$

Statement	Data flow function
`s: t := async e;`	λY.(if `e` $\in \mathcal{M}$(`s`) then $Y \cup$ {`t`} else Y)
`x := new` τ;	λY.($Y \cup$ {`x`})
`x.f := t;`	λY.(if `t` $\in Y$ then $Y \cup$ {`f`} else Y)
`t := x.f;`	λY.(if `f` $\in Y$ then $Y \cup$ {`t`} else Y)
`x := t;`	λY.(if `t` $\in Y$ then $Y \cup$ {`x`} else Y)

Figure 5.6 : Examples of normal flow function for computing $\mathcal{N}$

A *call-to-return flow function* intra-procedurally propagates data flow values that are independent of the call. The call-to-return flow function for computing $\mathcal{M}$ is the identity function.

The *may-not-be-future* analysis is performed after *may-be-future* analysis. Examples of normal flow functions for *may-not-be-future* analysis are given in Figure 5.6. The main difference relative to *may-be-future* is in the functions for the `async` and the `new` expressions. The synthesis algorithm does not create a future task from an async expression, `async e`, if `e` may be a future object (thereby ensuring that no nested futures are created). Therefore the flow function for `s: t := async e;` checks that `e` $\in \mathcal{M}$(`s`), before adding `t` to $\mathcal{N}$, where $\mathcal{M}$(`s`) denotes the set of variables and fields that may refer to a future immediately before statement `s`.

The result of future analysis are the sets $\mathcal{M}$ and $\mathcal{N}$, which will be available after the IFDS algorithm converges. The algorithm has worst-case complexity $O(ED^3)$, where E is the number of control-flow edges (or statements) of the analyzed program and D is the size of the analysis domain, where the domain consists of the set of all variables and fields in the program. We have not found this worst-case complexity to be a limitation in practice.

Section 5.4.2 contains the details of the transformation step based on the result of the data flow analysis.

```
1  class TreeNode {
2    private future<TreeNode> left, right;
3    ...
4    TreeNode bottomUpTree(int item, int depth){
5      if (depth > 0) {
6        future<TreeNode> l = async<TreeNode> {
7          return bottomUpTree(2*item-1, depth-1);
8        }
9        future<TreeNode> r = async<TreeNode> {
10         return bottomUpTree(2*item, depth-1);
11       }
12       return new TreeNode(l, r, item);
13     } else {
14       return new TreeNode(item);
15     }
16   }
17   ...
18   int itemCheck() {
19     ...
20     future<Integer> li = async<Integer> {
21       return left.get().itemCheck();
22     };
23     future<Integer> ri = async<Integer> {
24       return right.get().itemCheck();
25     };
26     return item + li.get() - ri.get();
27   }
28 }
```

Figure 5.7 : Binary tree program after synthesis of futures

5.4.2 Future Transformation

In this section, we present transformation rules for synthesizing futures based on the result of inter-procedural analysis from section 5.4.1. The transformation rules are shown in Table 5.1. The third column shows the input code and fourth column shows the transformed code, if the conditions in second column holds true. $\mathcal{M}(\mathtt{s})$ denotes the set of variables and fields which may refer to a future immediately before statement `s`. Similarly $\mathcal{N}(\mathtt{s})$ denotes the set of variables and fields which may not refer to a future immediately before statement

Rule	*IFDS Results*	*Input Code*	*Output Code*
1	$e \notin \mathcal{M}(s)$	`s: async e`	`async<T> { return e; }`
2	$\nexists s_1 : x \in \mathcal{N}(s_1)$	`T x`	`future<T> x`
3	$\exists s_1 : x \in \mathcal{M}(s_1) \wedge$ $\exists s_2 : x \in \mathcal{N}(s_2)$	`T x`	`mayfuture<T> x`
4	$x \in \overline{\mathcal{N}}(s)$	`s: a = x.f;`	`T y = x.get();`
5	$\exists s_1 : x \in \mathcal{M}(s_1) \wedge$ $\exists s_2 : x \in \mathcal{N}(s_2) \wedge$ $x \notin \mathcal{M}(s)$	`s: a = x.f;`	`T y = (T)x;` `a = y.f;`

6	$\exists s_1 : x \in \mathcal{M}(s_1) \wedge$ $\exists s_2 : x \in \mathcal{N}(s_2) \wedge$ $x \in \overline{\mathcal{N}}(s)$	`s: a = x.f;`	`future<T> t = (future<T>)x;`
			`T y = t.get();`
			`a = y.f;`

7	$\exists s_1 : x \in \mathcal{M}(s_1) \wedge$ $\exists s_2 : x \in \mathcal{N}(s_2) \wedge$ $x \in \mathcal{M}(s) \wedge x \notin \overline{\mathcal{N}}(s)$	`s: a = x.f;`	`T y;`
			`if (x instanceof future<T>)`
			`  future<T> t = (future<T>)x;`
			`  y = t.get();`
			`else`
			`  y = x;`
			`a = y.f;`

Table 5.1 : Example transformation rules based on `future`-analysis results

`s`. Rule 1 translates an `async` expression `async e` to a future task creation expression, if `e` may not be a future. Rule 2 changes the type of a variable from `T` to `future<T>`, if the variable must refer to a future object. Rule 3 changes the type of a variable to `mayfuture<T>`, if it may hold a reference to a future and a non-future object, where `T` is an application class. If `T` is a library class, the type is changed to `Object`. Rules 4-7 handle the different cases, where the object field, `f` is accessed. Rule 4 and 6 handle the case, where `x` must refer to a future object by inserting a `x.get()` operation which obtains the result of the future task. Rule 5 handles the case where `x` may not refer to a future object at statement `s`, but may refer to a future object at a different statement s_1. In this case, a cast operation from `mayfuture<T>` to `T` is required before the field access at statement `s`.

Rule 7 is the most general case, where `x` may refer to a future object or a non-future object at statement `s`. In this case, a runtime check is inserted which handles both the possible scenarios.

Our implementation also changes method parameter types and return types based on the result of future analysis. When the parameter type (or return type) of a class member function is changed, the type declaration of the corresponding function in the super class is also updated. For instance, let $D_1, .., D_n$ be the subclasses of C and let $T_1, ..., T_n$ be the inferred type of the ith parameter of member function F in $D_1, .., D_n$ respectively. The transformation algorithm then updates the type of ith parameter of F in C to $lub(T_1, ..., T_n)$ which is the least upper bound type of the types $T_1..T_n$, where for any given type `T`, `T` ≤ `mayfuture<T>` and `future<T>` ≤ `mayfuture<T>`.

The program from Figure 5.4 after synthesis of futures is shown in Figure 5.7. The calls to `bottomupTree` and `itemCheck` are translated to future tasks. The types of the local variables `l` and `r` and fields `left` and `right` are changed to `future<TreeNode>`. The synthesis algorithm also inserts `get` operations before the use of future objects in lines 20-26. Note that the synthesis algorithm does not insert `get` operations in line 12, where references to future objects are passed as arguments to the `TreeNode` constructor.

5.4.3 Candidate Future Identification

Our tool annotates calls to a subset of the pure methods identified by ReImInfer as async expressions. Although it is safe to execute all pure method calls as future tasks, it is not beneficial to execute method calls that perform insignificant work as separate tasks. Therefore, we identify methods that perform repetitive computations as candidates, by analyzing the call graph of the program and control flow graphs of each of the methods in the program.

Algorithm 14 classifies the methods in the input program as `async methods` and

Algorithm 14 Async method identification

Require: Call Graph of the Program , CFGs of each of the Methods
Ensure: Set of async methods A, Set of non-async methods S

1: **for all** $M \in \{M_1, .., M_n\}$ **do**
2: **if** (isLeaf(M) **and not** hasLoops(M)) **or**
3: **not** isPure(M) **or** hasExceptions(M) **then**
4: $S \leftarrow S \cup \{M\}$
5: **end if**
6: **if** (isRecursive(M) **or** hasLoops(M)) **and**
7: isPure(M) **and not** hasExceptions(M) **then**
8: $A \leftarrow A \cup \{M\}$
9: **end if**
10: **end for**
11: $Worklist \leftarrow \{M_1, .., M_n\} - S - A$
12: **while** $Worklist \neq \emptyset$ **do**
13: $M \leftarrow$ Extract($Worklist$)
14: $async \leftarrow$ `False`
15: **for all** $C \in$ Callees(M) **do**
16: **if** $C \in A$ **then**
17: $A \leftarrow A \cup \{M\}$
18: $async \leftarrow$ `True`
19: **break**
20: **end if**
21: **end for**
22: **if** $async$ = `False` **then**
23: $S \leftarrow S \cup \{M\}$
24: **end if**
25: **end while**

`non-async methods`. A method is classified as an async method if it or any method that it calls contains repetitive structures in the form of loops or recursive cycles. This classification is refined later in the Parallelism Benefit Analysis (PBA) step. Lines 1-10 of Algorithm 14 initialize the sets A and S, which are the set of async methods and set of non-async methods respectively. S is initialized to contain all non-pure methods and leaf

methods (methods which do not call other methods) with no loops in the method body. *A* is initialized to contain methods that are either recursive or contain a loop, with two further constraints: the method must be pure (isPure(*M*)), and must not throw any exceptions (hasExceptions(*M*) is false).

Purity ensures that asynchronous execution of the method with a future result will not result in nondeterministic behavior (provided that any input variables that may be mutated after the method call are cloned, as discussed in Section 5.4.4). Exceptions represent a special kind of side effect that is typically not included in the scope of purity analysis. A common assumption in defining the semantics of exceptions with futures is to propagate any exception thrown by the asynchronous task at the point when the get() operation is performed. However that approach makes it challenging to execute `foo()` asynchronously in scenarios such as the following,

```
try { x = foo() ; } catch { } ; y = x.z;
```

in which any exception thrown by `foo()` in the sequential version will be "swallowed" before the result of `x` is accessed as `x.z`, while, under common assumptions, `x.get().z` could throw an exception in the parallel version. The **not** hasExceptions(M) check ensures that this situation will not occur in our approach. While it is possible to identify some weaker sufficient conditions to be used as a replacement for **not** hasExceptions(M), our experience has been that the **not** hasExceptions(M) check provides a simple and effective means for ensuring correctness of our approach in the presence of exception semantics without limiting parallelism in practice.

The loop in lines 12-25 iteratively adds the remaining methods to *S* and *A*. The Extract method extracts a method *M* from the worklist such that all the methods invoked by *M* are already classified as async or non-async. A method that calls an async method is added to *A* and a method that calls only non-async methods is added to *S*.

5.4.4 Preserving Data Dependences

The parallel program after future synthesis is data race free and deterministic if it preserves all data dependences in the input sequential program. Purity analysis ensures that asynchronous execution of the candidate future methods do not violate *flow* (Read after Write) and *output* (Write after Write) dependences, since pure methods do not mutate global data. In order to preserve *anti* (Write after Read) dependences in the input program, our algorithm copies (clones) all mutable data read by candidate futures and the future tasks performs all reads on the copied data. A weaker sufficient condition (which leads to less copying) is to copy all memory locations, L such that $L \in READ(F) \cap MOD(C)$, where F is the candidate future function and C is the continuation after the call to F. *READ* is computed by standard side-effect analysis and *MOD* is computed by a backward data flow analysis on the inter-procedural control flow graph, where *READ* and *MOD* represent the *read set* and *modification set* respectively.

5.5 Parallelism Benefit Analysis

Section 5.4 presented the analysis for synthesizing a parallel program, in which all pure method invocations are executed asynchronously. We refer to this program as a parallel program with *ideal parallelism*, which has the smallest possible critical path length (CPL), if we ignore all overheads of parallelism including those arising from task creation, task termination, and task synchronization. In practice, these operations can incur significant overhead and it is necessary to ensure that every task has sufficient granularity to justify the task creation, task termination and synchronization overheads, and there is parallelism benefit that arises from each task creation. We now present an algorithm to classify invocations of a pure method, M at call site c into one of the following three classes:

- ***Sequential:*** A method call is classified as sequential, if all invocations of M at call site c must be executed sequentially.

- ***Parallel:*** A method call is classified as parallel, if all invocations of M at call site c can be executed asynchronously.

- ***Conditional Parallel:*** A method call is classified as conditional parallel, if a subset of the invocations of M at c must be executed asynchronously and the rest must be executed sequentially. In this case, the determination of whether a specific dynamic call should be executed sequentially or in parallel will be made by evaluating an automatically synthesized predicate expression at runtime.

The classification algorithm first constructs a data structure called the *weighted computation graph* (WCG), which is introduced in Section 5.5.1. Next, Section 5.5.2 presents an algorithm that uses the WCG to classify the calls into the three categories listed above. The WCG construction algorithm takes as input the parallel program with ideal parallelism synthesized by the algorithm in Section 5.4 and one or more test inputs for the program.

5.5.1 Weighted Computation Graph

.

A weighted computation graph (WCG) is a directed acyclic graph that is built at runtime to capture 1) the happens-before relationships among the step instances of a parallel program's execution, 2) the work done by each of the steps, and 3) the overheads incurred in task creation, termination and synchronization. Computations are represented in the WCG using step nodes, which are defined as follows:

Definition 10. A *step node* represents a maximal sequence of statement instances such that no statement instance in the sequence includes the start or end of an async or a future get

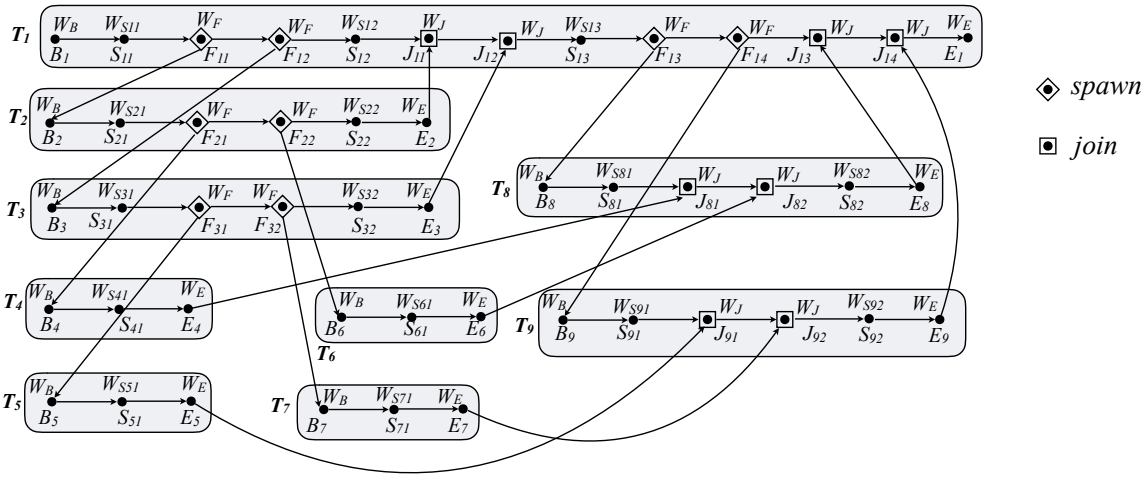

Figure 5.8 : Weighted computation graph for binary tree program in Figure 5.7 for `depth=2`

operation.

Definition 11. The weighted computation graph (WCG) for a given execution is a directed acyclic graph with four different types of nodes:

- A step node, S_n represents a sequential computation. The weight of step node is the total number of instructions executed to complete the step.
- A spawn node F_n represents the creation of child task. The weight of spawn node T_{spawn} represents the overhead in the parent task for task creation. This includes the overhead of copying the mutable data read by the child task.
- A join node J_n represents the join operation with another task. The weight of join node T_{join} represents the overhead of a join operation in the waiter task.
- A start node B_n is the first step in a task. The weight of start node T_{start} represents the overhead of creating and scheduling a task.

- An end node E_n is the last step in a task. The weight of end node T_{end} represents the overhead of task termination.

Next we discuss how to build the WCG during program execution. We first instrument the parallel program with ideal parallelism generated by the algorithm in Section 5.4. The instrumented program is then executed on a test input in serial, depth-first order (like a sequential Java program) to construct the WCG. The instrumented code performs the WCG construction as follows: When the main task starts execution, the WCG will contain two nodes: 1) B_1 which corresponds to the start node of the main task and 2) step node S_1 which corresponds to the starting computation inside main. The edge $B_1 \rightarrow S_1$ represents the ordering between B_1 and S_1.

Future Task Creation When a task T_a creates a child task T_b, a spawn node F_i is created and an edge is inserted from S_j to F_i, where S_j is the step immediately preceding the spawn operation in T_a. A start node B_k corresponding to T_b is created and an edge is inserted from F_i to B_k. Task T_b is now executed and the next node N (step, spawn, join or end node) is added as a successor of B_k.

Future Task Termination When a task T_b completes execution, an end node E_i is created and an edge is inserted from S_j to E_i, where S_j is the last step in T_b. The program execution now continues in T_a which is the parent task of T_b. The next node to be added is the successor of F_k which is the spawn node in T_a corresponding to the creation of T_b.

Future Join When task T_a performs a join operation on task T_b, a join node J_i is created and an edge is inserted from S_j to J_i, where S_j is the step immediately preceding the join operation in T_a. An edge is inserted from E_k to J_i, where E_k is the end node in T_b. Execution of T_a continues at node N, which is the successor of J_i.

Example The WCG for the Binary Tree program in Figure 5.7 for input `depth=2` is shown

in Figure 5.8. The weights of each of the nodes are shown above the node. Task T_1 which consists of the directed path from B_1 to E_1 is the main task. T_2, T_3, T_4, T_5, T_6 and T_7 are future tasks created in line 6 and line 9 for the invocations of `bottomupTree`. T_8 and T_9 are future tasks created in line 20 and line 23 for the invocation of `itemCheck`. This WCG demonstrates the generality of synchronization patterns possible with futures. For instance, the edge from E_7 to J_{92} (T_7 to T_9) due to the future `get` operation is not possible in more structured fork-join models.

5.5.2 Classification of Pure Function Calls

We now analyze the WCG to identify tasks that give no benefit from asynchronous execution. Based on this analysis, we classify pure method invocations as parallel, sequential or conditional parallel. For every task T_a, our analysis tries to answer two questions: 1) Is the work done by task T_a of sufficiently coarse granularity to justify the task creation, termination and synchronization overhead? and 2) Is there sufficient work that can be overlapped with the execution of T_a? We use the critical path length (CPL) as the cost metric for evaluating the profitability of parallelization, where the critical path is defined as follows.

Definition 12. The critical path of a weighted computation graph is the longest weighted path in the WCG, where the weight of a path is the sum of the weights of all the nodes included in the path.

Algorithm 15 presents our approach for evaluating the parallelism benefit for each of the tasks in the WCG. The algorithm takes as input the WCG, G and the set of all tasks, T. The outputs of the algorithm are the set of parallel tasks, P and the set of serial tasks, S. The algorithm uses a greedy strategy, evaluating the tasks in bottom-up, right-to-left order. The algorithm merges a task with its parent, if executing that particular task asynchronously

Algorithm 15 Parallelism benefit analysis

Require: Computation graph G, Set of tasks T

Ensure: Set of sequential tasks S, Set of parallel tasks P

1: **for all** $t \in T$ **do**
2: **if** WORK(t) $< T_{spawn}$ **then**
3: $S \leftarrow S \cup \{t\}$
4: $G \leftarrow$ MERGEPARENT(G, t)
5: $Visited \leftarrow Visited \cup \{t\}$
6: **end if**
7: **end for**
8: **for all** $t \in T - Visited$ **do**
9: $P \leftarrow P \cup \{t\}$
10: **if** CHILDREN(t) $= \emptyset$ **and** RIGHTSIBLINGS(t) $= \emptyset$ **then**
11: $Worklist \leftarrow Worklist \cup \{t\}$
12: **end if**
13: **end for**
14: $CPL \leftarrow$ LONGESTPATH(G)
15: **while** $Worklist \neq \emptyset$ **do**
16: Remove t from $Worklist$
17: $Visited \leftarrow Visited \cup \{t\}$
18: $G' \leftarrow$ MERGEPARENT(G, t)
19: $CPL' \leftarrow$ LONGESTPATH(G')
20: **if** $CPL' < CPL$ **then**
21: $G \leftarrow G'$
22: $P \leftarrow P - \{t\}$
23: $S \leftarrow S \cup \{t\}$
24: $CPL \leftarrow CPL'$
25: **end if**
26: **for all** $t_1 \in T - Visited$ **do**
27: **if** (CHILDREN(t_1) $\cup$ RIGHTSIBLINGS(t_1))
28: $\cap$ $Visited = \emptyset$ **then**
29: $Worklist \leftarrow Worklist \cup \{t_1\}$
30: **end if**
31: **end for**
32: **end while**

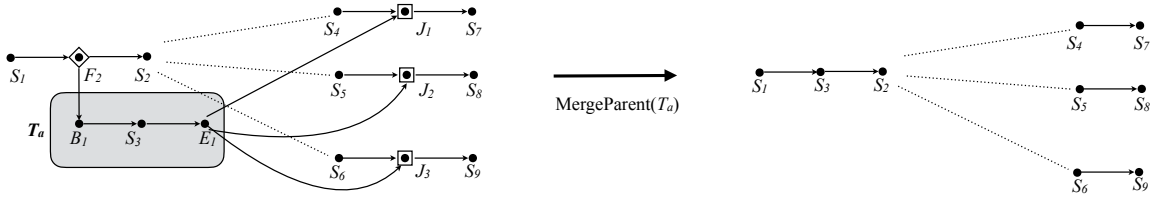

Figure 5.9 : Merge parent transformation on the computation graph, where the task T_a is merged with its parent.

does not yield any benefit. The bottom-up approach ensures that the tasks of smaller granularity are first considered as merge candidates. Evaluating the right siblings of a task T_a as merge candidates before T_a itself ensures that the algorithm obtains a more accurate estimate of the total work in the continuation before making a decision on the profitability of executing T_a asynchronously. Lines 1-7 of Algorithm 15 classify a task as serial and merges it with its parent, if the total work done by the task is less than T_{spawn}. Lines 8-13 initialize the worklist with the set of tasks which have no children and no right siblings. The set of parallel tasks P is initialized to contain all non-serial tasks. Lines 15-32 remove a task t from the worklist, classifies it as serial/parallel and updates the worklist with tasks which are ready to be evaluated. Lines 18-25 merge t with its parent and checks if the CPL after merging is smaller than the CPL before merging. If the merging results in a smaller CPL, the task t is classified as serial and the WCG is updated.

Figure 5.9 shows an example of the MERGEPARENT transformation on the WCG, where the task T_a is merged with its parent. F_2 is the spawn node corresponding to the task T_a in the parent task. Task T_a consists of the step node S_3 in addition to the start node and the end node. There are three separate join nodes corresponding to T_a, which are J_1, J_2 and J_3. The dotted edges from S_2 to S_4, S_5 and S_6 represents paths in the WCG. The MERGEPARENT transformation removed the spawn, start, end and join nodes corresponding to T_a and inserted S_3 along the path from S_1 to S_2 where S_1 and S_2 are the nodes preceding

and succeeding the spawn node F_2 in the WCG. This transformation ensures that there are paths from S_3 to S_7, S_8 and S_9 which are successor nodes of the join nodes in the input WCG.

Our approach performs parallelism benefit analysis separately on each of the WCGs corresponding to each of the test inputs. The output of parallelism benefit analysis is the set of serial and parallel tasks. Next, we merge the output of parallelism benefit analysis for all WCGs at a particular call site and identify parallel and conditional parallel call sites. Classifying call sites as serial, parallel, and conditional parallel based on this output is straightforward. If all instances of a method at a particular call site are serial/parallel, then that call site is classified as serial/parallel. If only a subset of instances of a method at a particular call site are serial, we mark it as conditional parallel.

5.6 Threshold Expression Synthesis

Parallelism benefit analysis classifies the pure method calls as sequential, parallel and conditional parallel. Conditional parallel calls are method calls that should be executed asynchronously only if the work done by the method is greater than *sequential threshold*, which is the minimum amount of work (in terms of instruction count) that must be done by a task to justify the task overhead. For a method invocation `obj.f(`a_1, a_n`)`, which is classified as conditional parallel, our goal is to generate the code shown in Figure 5.10. The conditional expression `cond` determines if the work done by `f` is greater than sequential threshold and we refer to this as the *threshold expression*.

The work done by a pure method call `obj.f(`a_1, a_n`)` typically depends on 1) the type of `obj`, because a class hierarchy can contain multiple implementations for the same method and the type of `obj` determines which implementation of `f` is invoked, 2) the values of the arguments of `f`, and 3) the values of the fields of `obj`. For example, the total work done

```
1 mayfuture<T> x;
2 if (cond)
3   x = async<T> { return obj.f(a1,..,an);};
4 else
5   x = obj.f(a1,..,an);
```

Figure 5.10 : Conditional parallel execution of method call `obj.f`(a_1, a_n)

by the recursive Fibonacci function `int fib(int n)` depends on `n` and the work done by a sort function `int[] sort(int[] A)` depends on `A.length`. Here `n` and `A.length` represent the problem sizes for the two functions respectively.

While instrumenting the input program for WCG construction, we also instrument it to collect the following information for each invocation of a method `f` executed as a future task:

- The type `T` of the receiver object `X`
- The problem size parameters $p_1, .., p_k$ of `f`, which consists of
  - The values of numeric type arguments of `f`
  - The values of numeric type fields of arguments of `f`
  - The values of numeric type fields of `X`, where `X` is the receiver object of `f`
  - The size of collection/data structure (List, Array, String) type arguments of `f`
- The cumulative work, w done by `f` and all the methods that it calls. The work is computed in terms of the number of instructions executed.

Since `f` is a pure method, the threshold expression for `f` is usually a function of the problem size parameters and the type of the receiver object.

At each conditional parallel call site the profile information is divided based on the type of the receiver object. Note that the profile information for each conditional parallel call

site from multiple test inputs is merged before threshold expression synthesis. For each receiver type `T`, we construct a matrix, M with $k+1$ columns, where columns 1 to k contains the values of the problem size parameters p_1 to p_k, and column $k+1$ contains work w. For most pure methods, the problem size is one of $p_1, .., p_k$. In other cases the problem size is an expression involving two or more of the parameters $p_1, .., p_k$. Our approach finds the threshold expression by performing a search on the space of expressions formed from $p_1, .., p_k$ and a set of arithmetic operators. Our current implementation handles arithmetic operators +, -, `min` and `max`. The search algorithm looks for an expression e, such that e has a monotonic relationship with w – as the value of e increases, so does the value of w. We use Spearman's rank correlation [53] coefficient, ρ as the metric for monotonicity. Spearman's correlation coefficient assesses how well the relationship between two variables X and Y can be described using a monotonic function. Note that ρ is a non-parametric measure and is not based on a possible relationship of a parameterized form (such as a linear relationship). The value of ρ lies between +1 and -1 inclusive, where 1 is total positive correlation, 0 is no correlation, and -1 is total negative correlation. Our algorithm starts by computing ρ between each of the problem size parameters p_i and the work w. If the algorithm finds a p_i which has high rank correlation to w ($\rho(p_i, w) > \rho_{threshold}$, where $\rho_{threshold} = 0.9$), the algorithm terminates returning p_i as the problem size. If none of the parameters has high correlation with w, the algorithm computes the correlation of expressions involving two parameters such as $p_i + p_j$ and $p_i - p_j$. This is done by constructing a new matrix in which each of the columns correspond to an expression. The search continues with larger expressions until an expression with the desired correlation is found or when the algorithm has explored all expressions of size n, where n is a tuning parameter for the search. This search algorithm for an expression with syntactic constraints which meets a correctness specification (high correlation) is similar in spirit to syntax guided synthesis [54]. If TES for a call site c is

unable to find a threshold expression, we classify c as parallel/serial based on the frequency of parallel/serial invocations in the output of parallelism benefit analysis. Next we find the minimum value, v of e for which the method must be executed asynchronously by a lookup of M. This information on which rows in M corresponds to parallel execution is available from parallelism benefit analysis. The result of threshold expression synthesis for a given receiver type `T` is the expression ((`obj instanceof T`) $\wedge$ ($e \leq v$)) or ((`obj instanceof T`) $\wedge$ ($e \geq v$)), depending on whether the correlation between the expression and the work is positive or negative. The final threshold expression is a disjunction of threshold expressions for each of the receiver types. The instanceof check can be eliminated if the declared type of the receiver object is same as `T`.

As an example, consider the `int[] mergesort(int a[], int start, int end)` function, which sorts the elements of array `a[]` starting at index `start` and ending at index `end`. The problem size parameters for this function are 1) `a.length`, which is the length of the array `a`, 2) `start` and 3) `end`. The TES algorithm computes the rank correlation between each of these parameters and the work w done by the function. The algorithm continues the search, since none of these parameters have high correlation with the work. Next, the algorithm computes the correlation between expressions involving two parameters such as `a.length + start`, `start + end`, and `end - start`. Finally, the algorithm returns `end - start` as the threshold expression, since it has high correlation to the work done by the function.

Algorithm 16 presents the high level view of our approach for threshold expression synthesis for a given call site `obj.f(`$a_1..a_n$`)`. Line 1 of the algorithm initializes the threshold expression to false. The loop in lines 2-20 iterates through each of the possible types of the receiver object and finds the threshold expression. Lines 3-16 handle the case where the call site is conditional parallel for type `T`. ConstructMatrix constructs a matrix with

Algorithm 16 Threshold expression synthesis

Require: Profile data, P for call site `obj.f(`$a_1..a_n$`)`

Ensure: Threshold expression for call site `obj.f(`$a_1..a_n$`)`

```
1:  TExpr ← False
2:  for all T ∈ Types(obj) do
3:      if obj.f(a1..an) is conditional parallel for type T then
4:          for size = 1 to n do
5:              M ← ConstructMatrix(P, size)
6:              (ρ_max, expr_max) ← MaxSpearmanCorr(M)
7:              if |ρ_max| ≥ ρ_threshold then
8:                  value ← GetSeqThreshold(M, expr_max)
9:                  if ρ_max > 0.0 then
10:                     TExpr ← TExpr ∨ ((obj instanceof T) ∧ (expr_max ≥ value))
11:                 else
12:                     TExpr ← TExpr ∨ (( obj instanceof T) ∧ (expr_max ≤ value))
13:                 end if
14:                 break
15:             end if
16:         end for
17:     else if obj.f(a1..an) is parallel for type T then
18:         TExpr ← TExpr ∨ (obj instanceof T)
19:     end if
20: end for
```

m columns, where columns 1 to $m-1$ contains the data corresponding to the expressions and column m contains the work. MaxSpearmanCorr computes the Spearman's rank correlation between each of the columns $1..m-1$ and column m and returns the maximum correlation, ρ_{max} and the expression $expr_{max}$ having the highest correlation. If the absolute value of ρ_{max} is greater than or equal to $\rho_{threshold}$, we have found the threshold expression, else we continue the search with expressions of larger size. Lines 7-13 extends the threshold expression depending on whether the correlation between the expression and work is positive or negative. The method GetSequentialThreshold returns the minimum/maximum value of $expr_{max}$ for which the call site must be executed asynchronously, depending

on whether the work done by `f` increases or decreases as the value of $expr_{max}$ increases.

5.7 Final Future Synthesis

The last step in our parallelization tool is the generation of parallel code, in which pure method calls which are found to be beneficial by the analysis in Section 5.5 are executed asynchronously (and the others are executed sequentially). The inputs to the parallel code generation are the input sequential program, the set of parallel call sites, the set of conditional parallel call sites and the threshold expressions for each of the conditional parallel call sites computed by the algorithm in Section 5.6. (Note that this step uses the original sequential program as input, and not the parallel program from Section 5.4.)

The final future synthesis step includes the following steps:

1. Generate conditional statements at conditional parallel call sites using threshold expressions. The true branch represents the case where the work done by the method is greater than the sequential threshold and the false branch represents the case where the work done by the method is less than the sequential threshold. We annotate the method call in the true branch as an async expression.

2. Annotate all parallel call sites as async expressions

3. Clone all inputs to each pure method that may be modified in any of the continuations that follow each of the parallel and conditional parallel calls, and replace all references to those inputs in the pure method by references to the cloned data.

4. Synthesize futures in the async-annotated program resulting from the previous steps, using the synthesis algorithm presented in Section 5.4.1.

Source	Benchmark	Description	Input Size (Train)	Input Size (Ref)
JGF [36]	Series	Fourier coefficient analysis	size A	size B
SPECjvm2008 [55]	MPEGaudio	MPEG audio decoder	4 mp3 files	12 mp3 files
CLBG [49]	Binary Tree	Tree construction traversal	*depth* = 14	*depth* = 20
Jolden [56]	TreeAdd	Recursive depth-first traversal of a tree	*depth* = 15	*depth* = 24
BOTS	Nqueens	N Queens problem	$n = 9$	$n = 13$
HJ Bench	Fibonacci	Compute *n*th Fibonacci number	$n = 22$	$n = 38$
	MatrixEval	Matrix expression evaluation	200x200	500x500
	Mergesort	Mergesort	$n = 16000$	$n = 1000000$
	Quicksort	Quicksort	$n = 10000$	$n = 1000000$

Table 5.2 : List of benchmarks evaluated. **Input Size(Train)** is the input size used for profiling the parallel program for parallelism benefit analysis and **Input Size(Ref)** is the input size used for performance evaluation.

As discussed earlier, the result of final future synthesis for the program in Figure 5.1 can be seen in Figure 5.2.

5.8 Experimental Evaluation

5.8.1 Experimental Setup

In this section, we summarize the implementation and setup used in our experimental evaluation. The different components of our system were implemented in Habanero Java (HJ) [16] compiler and runtime as follows. The HJ compiler extends the Soot framework [35] for bytecode transformations. Inter-procedural future analysis is implemented as a new compiler analysis pass in Heros [57], a scalable, highly multi-threaded implementation of the IFDS framework, which can be invoked from Soot. Our inter-procedural analyses used the call graph provided by the Soot framework, which is obtained through

class hierarchy analysis and is sound. The insertion of future operations is implemented as a subsequent compiler transformation pass that updates Soot's Jimple [35] intermediate representation extended for HJ programs. Instrumentation of programs for WCG construction and computation of instruction counts of steps is also implemented as a bytecode-level transformation pass on Jimple. The instrumentation pass inserts callbacks to the HJ runtime at all future task creation, termination and synchronization points in the program and also inserts counters to compute the dynamic number of instructions executed during the program execution. For each method call that is executed as future task, we instrument the bytecode to collect the values of the arguments of the method, the type and the fields of the receiver object. This information is used for threshold expression synthesis. The instrumented program, during execution writes the profile information to a file. Parallelism benefit analysis is implemented as a compiler pass in the HJ compiler which reads the profile information, analyzes it and finds the set of method calls that are parallel and conditional parallel. We used the JGF ForkJoin microbenchmark to measure the overheads of task creation and task termination for our runtime and a given hardware platform. This information and the profile information is passed to threshold expression synthesis, which determines the threshold expressions for each of the conditional parallel method calls. Finally, the parallel code is generated by invoking the inter-procedural future analysis and the future transformation pass.

Our experiments were conducted on a 16-core Intel Ivybridge 2.6 GHz system with 48 GB memory, running Red Hat Enterprise Linux Server release 7.1, and Sun Hotspot JDK 1.7. To reduce the impact of JIT compilation, garbage collection and other JVM services, we report the steady state mean execution time of 30 runs repeated in the same JVM instance for each data point. In all our measurements, we only used 8 of the 16 cores to execute the application (by using HJ's "-places 1:8" option), so as to further reduce the

```
1 Integer f1 = fib(n-1);
2 Integer f2 = fib(n-2);
3 return f1+f2;
```

Figure 5.11 : Sequential recursive Fibonacci invocation

impact of system perturbations.

We evaluated the parallelization tool on a suite of nine benchmarks listed in Table 5.2. Our approach targets applications in which pure method calls perform significant amount of work. Therefore, we chose benchmarks in which at least 50% of the sequential work is performed in pure method calls. We did this by using ReImInfer to identify pure methods and by inserting timers around calls to pure methods. The fourth column of Table 5.2 shows the input size used for profiling the program ("Train"). The fifth column shows the input size used for performance evaluation of the parallelized programs ("Ref").

Benchmark	**#Candidate**	**#Serial**	**#Parallel**	**#Conditional**
Series	3	1	2	0
MPEGaudio	1	0	1	0
Binary Tree	12	9	1	2
TreeAdd	6	4	0	2
Fibonacci	3	2	0	1
MatrixEval	3	2	1	0
Mergesort	4	3	0	1
Nqueens	3	2	0	1
Quicksort	3	2	0	1

Table 5.3 : Number of call sites identified as serial, parallel, and conditional parallel by parallelism benefit analysis

5.8.2 Experimental Results

We now present experimental results for our automatic parallelization approach. Table 5.3 shows the results of parallelism benefit analysis. The second column shows the number

Benchmark	#Future	#MayFuture	#Gets	#Instanceof	#Typecasts
Series	2	3	2	4	4
MPEGaudio	3	0	1	0	1
Binary Tree	2	15	4	6	7
TreeAdd	0	10	4	8	8
Fibonacci	0	2	1	2	2
MatrixEval	8	0	3	0	3
Mergesort	0	4	1	2	2
Nqueens	0	4	2	4	4
Quicksort	0	2	1	2	2

Table 5.4 : Synthesis statistics. **#Future** gives the number of variables and fields whose type got changed to `future<T>`, **#MayFuture** gives the number of variables and fields whose type got changed to `mayfuture<T>`. **#Gets** , **#Instanceof** and **#Typecasts** are the number of `get()`, `instanceof` and cast operations inserted by future synthesis.

of call sites that were identified as candidates for future task creation by the algorithm in Section 5.4.3. The third, fourth, and fifth columns shows the number of call sites that were identified as serial, parallel, and conditional parallel respectively by parallelism benefit analysis.

A call site may be classified as serial if there is insufficient work done by the method or if there is insufficient parallelism. For example, in the Fibonacci program in Figure 5.11, the call to `fib` in line 2 will be classified as serial, since there is no benefit in executing it as a separate future task, where as the call in line 1 will be classified as conditional parallel, because 1) the work done by the function depends on the value of `n` and 2) if the call is executed as a future task, it can execute in parallel with the call in line 2.

Table 5.3 shows that a subset of methods identified by the algorithm in Section 5.4.3 benefit from asynchronous execution, thereby reinforcing the importance of parallelism benefit analysis in choosing method calls for parallelization.

Table 5.4 shows the statistics resulting from the final future synthesis step, which is performed after parallelism benefit analysis and threshold expression synthesis. It shows

Benchmark	Seq	Par(No PBA)	Par(No TES)	Par	Speedup (Seq/Par)
Series	25,973.35	4,175.49	4,203.84	4,201.22	6.18
MPEGaudio	6,691.52	2,839.76	2,838.51	2,835.21	2.36
Binary Tree	527.50	167,674.52	529.11	271.33	1.94
TreeAdd	586.54	293,431.26	602.21	259.49	2.26
Fibonacci	473.92	OOM	474.33	64.07	7.40
Quicksort	282.71	OOM	296.39	70.58	4.00
MatrixEval	1,853.83	884.11	359.23	358.86	5.17
Mergesort	151.31	OOM	151.57	40.96	3.69
Nqueens	4,242.57	OOM	4,211.21	1,199.07	3.54
GeoMean	-	-	-	-	3.69

Table 5.5 : Comparison of execution times in milliseconds of the sequential and parallel versions of the program. **Seq** is the sequential execution time, **Par(No PBA)** is parallel execution time without parallelism benefit analysis, **Par(No TES)** is parallel execution time without threshold expression synthesis and **Par** is the parallel execution time of the final program generated by our approach. Parallel versions were run on 8 cores. OOM (Out Of Memory) represents cases where execution did not complete because of insufficient heap memory.

the number of variable types that were changed and the number of `get()`, `instanceof` and cast operations that were inserted by future synthesis. Overall, these statistics indicate that significant programmer effort is required to manually parallelize a sequential program using futures, and that this effort may need to repeated for different platforms (due to differences in overheads and available hardware parallelism).

Table 5.5 compares the execution times of the sequential and automatically parallelized versions of each program in our benchmark suite. All programs were run with 16GB heap space. **Par(No PBA)** shows the parallel execution time without parallelism benefit analysis. All call sites that were identified as candidates for future task creation by the algorithm in Section 5.4.3 were executed as parallel tasks with no threshold expressions. OOM (Out Of Memory) represents cases where execution did not complete because of insufficient heap memory. **Par(No TES)** shows the parallel execution times with parallelism benefit

Benchmark	Threshold - 2	Threshold - 1	Threshold	Threshold + 1	Threshold + 2
Binary Tree	309.18	292.13	271.33	295.85	306.62
TreeAdd	349.83	279.53	259.49	259.26	256.93
Fibonacci	OOM	88.2	64.07	57.71	54.37
Quicksort	89.74	85.18	70.58	66.69	69.73
Mergesort	43.92	40.68	40.96	41.79	41.76
Nqueens	1,461.63	1,261.53	1,199.07	1,556.3	4,665.8

Table 5.6 : Comparison of execution times in milliseconds of the parallel versions of the programs with different threshold values for conditional parallel call sites. **Threshold** is the execution time with threshold expressions computed by TES. **Threshold - 1** and **Threshold - 2** are execution times of parallel programs with higher parallelism compared to **Threshold**. **Threshold + 1** and **Threshold + 2** are execution times of parallel programs with lower parallelism compared to **Threshold**.

analysis but without threshold expression synthesis. In this case, we used the parallel/serial frequency information at a call site to classify a method call as either serial or parallel. There are no conditional parallel method calls in this case. **Par** is the parallel execution time with parallelism benefit analysis and threshold expression synthesis. All the applications showed significant performance improvements with our approach.

Table 5.6 compares the execution times of the parallel versions of each program with conditional parallel sites when using different threshold values. **Threshold** shows the parallel execution time when using threshold values computed by our threshold expression synthesis algorithm. For this sensitivity analysis, we used two threshold values which allow higher parallelism and two threshold values which allow lower parallelism compared to the threshold value computed by threshold expression synthesis. **Threshold - 1** shows the parallel execution time when a threshold value that allows higher parallelism compared to **Threshold** is used. **Threshold - 2** shows the parallel execution time where the threshold value allows higher parallelism compared to **Threshold - 1**. Similarly, **Threshold + 1** and **Threshold + 2** shows the parallel execution times where the parallelism is lower compared to **Threshold**. For example, the threshold value computed by TES for the Binary Tree

benchmark is 12. Therefore, we used threshold values of 10 (**Threshold - 2**), 11 (**Threshold - 1**), 12 (**Threshold**), 13 (**Threshold + 1**) and 14 (**Threshold + 2**) for the sensitivity analysis. For Quicksort and Mergesort benchmarks, the threshold values are varied by a factor of two. The results indicate that the performance of the parallel program when using threshold values computed by threshold expression synthesis is better or comparable to the performance of the program when using a different threshold value.

In summary, these results indicate the importance of parallelism benefit analysis and threshold expression synthesis in automatic generation of task parallelism.

5.9 Related Work

5.9.1 Futures

Futures were introduced by Halstead as an explicit concurrency primitive for functional programming in Multilisp [18], an untyped language that did not need the synthesis capabilities introduced in our work. Flanagan and Felleisen [58] defined a whole program analysis to reduce runtime checks for futures in dynamically typed languages. In contrast, our work synthesizes future operations, synchronization and runtime checks, while also providing parallelism benefit analysis and threshold expression synthesis capabilities.

Safe futures [59] implement futures as software transactions so that safety violations (data races) can be avoided or corrected. Their approach requires a heavyweight runtime which supports object versioning, operation logging and other metadata. Performance results can vary widely depending on the number of safety violations detected at runtime. Navabi et al. [60] uses static analysis to insert barriers, which preserve sequential semantics with the help of a lightweight runtime. Swaine et al. [61] provides a way to add parallelism to legacy runtime systems using futures. These approaches require the programmer to par-

allelize the program and the framework handles conflicts due to shared data accesses. In contrast, our approach is fully automatic, targets pure method calls and has to deal with only anti-dependences on mutable data, which are preserved by copying the data.

Pratikakis et al. [50] present a framework for transparently executing programs with asynchronous calls. They employ a static analysis based on qualifier inference to identify the proxy variables in the program. Their analysis is flow-insensitive and context-insensitive, and does not differentiate *maybe proxy* variables from *mustbe proxy* variables. Due to these differences, their approach changes the types of all variables which could potentially be a proxy to `java.lang.Object`. Their framework also requires a type cast and an `instanceof` check at every potential proxy access. Their framework requires the programmer to annotate the async expressions, which can cause data races. The programmer also has to determine whether the annotated expressions have benefits from asynchronous execution.

Harris and Singh [62] presented a profile based parallelization for Haskell programs. Their approach selects thunks for parallel execution which are likely to be needed by the program and will run long enough to compensate the overheads. In contrast to our work, their approach does not require future synthesis and does not find threshold expressions which can dynamically determine if the work is of sufficient granularity to justify task creation overhead.

Directive-based Lazy Futures [63] require users to annotate declarations of all variables that store the return value from a function that can be potentially executed as futures with @future directives. Their approach does not allow the propagation of future objects across method boundaries, which can limit parallelism in many cases. In contrast, our approach automatically identifies the variables and fields which may hold references to future objects and inserts `get()`, without limiting the parallelism at method boundaries. Zhang et

al. [64] and Navabi et al. [65] presented approaches for precise exceptions in the presence of futures, which is orthogonal to our work which addresses automatic parallelization.

5.9.2 Parallelism and Performance Profiling

Past work has used profile information and critical path analysis to analyze the parallelism in a given application. Kulkarni et al. [66] used a critical path based analysis to bring insight into the parallelism inherent in irregular algorithms that exhibit amorphous data parallelism. Kremlin [67], given a serial version of a program, will make recommendations to the user as to what regions (e.g. loops or functions) of the program to parallelize first using a hierarchical critical path analysis. It also provides a ranked order of specific regions to the programmer that are likely to have the largest performance impact when parallelized. Cilkview [68] scalability analyzer is a software tool for profiling and estimating scalability of parallel Cilk++ applications. It monitors the logical parallelism during an instrumented execution of the application on a single core. As Cilkview executes, it analyzes logical dependencies within the computation to determine its work and critical path length. It uses these metrics to estimate parallelism and predict the scalability of the application. Unlike Cilkview, which analyzes only the whole-program scalability of a Cilk computation, Cilkprof [69] collects work and critical-path length for each call site in the computation to assess how much each call site contributes to the overall work and span.

Threshold expression synthesis finds an expression involving method arguments and receiver object fields which is monotonic with respect to the work done by the function. Past work has tried to model the performance of programs as function of the size of the input. Emilio Coppa et al. in [70] presented input-sensitive profiling, a method for automatically measuring how the performance of individual routines scales as a function of the size of the input. The key feature of their method is the ability to automatically measure

the size of the input given to a generic code fragment. The tool estimates the input size by using the amount of distinct memory first accessed by a routine or its descendants as reads. This work was extended in [71] which takes into account dynamic workloads produced by memory stores performed by other threads and by the OS kernel. As noted in [70], their approach fails to characterize pure functional computations such as Fibonacci where the running time (or work) is determined by the values of one or more arguments. Algorithmic profiling [72] is an approach to automatically infer approximations of the expected algorithmic cost functions of algorithm implementations. Our method of determining the inputs to a function is similar to their approach. Their implementation automatically infers the inputs to the program, but the fitting of cost functions is done by hand. In contrast, our approach does not try to find an exact cost function, but uses a search algorithm to find an expression which is monotonic with respect to the work done by the method.

Duran et al. [73] presented a runtime technique to adaptively coalesce OpenMP tasks by employing a dynamic profiler. The profiler estimates the work performed by a task as the average work performed by all previously profiled tasks at that particular level/depth of the spawn tree. The work estimation does not depend on the computation performed by the task or the arguments to the computation, whereas our approach computes a distinct threshold expression for every call site and takes into account the arguments to the computation.

Thoman et al. [74] presented a combined compiler and runtime approach that enables automatic granularity control. They generate multiple versions of a given task of increasing granularity by task unrolling at compile time and the runtime system selects a task version by estimating task demand. The number of generated versions depend on the granularity of the initial tasks, but the paper does not discuss how the task granularity is estimated. A runtime estimate of task demand could be combined with our approach to prevent task creation if the demand is low.

5.10 Summary and Future Work

We presented a novel approach for automatically parallelizing pure method calls by using futures as the primary parallel construct. Given a sequential program, our algorithm automatically generates a parallel program in which pure method calls that benefit from asynchronous execution are executed as future tasks. Our approach addresses the major drawbacks of manually parallelizing programs using futures. Section 5.4 contains our algorithm for synthesizing future tasks and their associated type declarations with more precision than in past work. Section 5.5 describes our approach to classifying each pure method call as sequential, parallel, or conditional parallel, based on computing critical path lengths in a weighted computation graph that takes task creation, termination, and synchronization overheads into account. Section 5.6 contains our algorithm for synthesizing threshold expressions that can be evaluated at runtime, to ensure that a future task is only created when it is profitable to do so. We implemented all three steps in our approach, and evaluated the complete tool chain on a range of applications written in Java. When using 8 processor cores, the evaluation shows that our approach can provide significant parallel speedups of up to 7.4× (geometric mean of 3.69×) for sequential programs with zero programmer effort, beyond providing test cases for parallelism benefit analysis.

There are many opportunities for future research to build on the results of this chapter. Future synthesis is inter-procedural in scope, and requires whole-program static analysis in general. A direction for future work is to make our approach modular by ensuring that there is no asynchronous information flow through future objects across components. Enhancements in alias analysis could further increase the precision of type information in the synthesized program. Alternatively, our approach can be applied to dynamically typed languages for which no type declarations need to be generated. There is also room to further study the impact of exception semantics on automatic parallelization with futures,

and to explore the use of runtime checks for potential exceptions in candidate future tasks. There is a promising opportunity for code motion to separate future task creation and the corresponding future `get()` operations as far as possible, so as to further increase the parallelism in the program (akin to global instruction scheduling). Yet another direction for future work is to extend our approach so that it can be applied to programs with explicit task parallelism, thereby using future tasks to further increase parallelism; it would also be interesting to perform parallelism benefit analysis and threshold expression synthesis for explicitly-parallel programs so as to aid the programmer in granularity control of their parallelism. Finally, it would be desirable (albeit challenging) to perform parallelism benefit analysis and threshold expression synthesis at runtime, so that they can be better tuned to the underlying platform.

Chapter 6

Putting It Together

Debugging, repair, and synthesis of parallelism are both synergistic and orthogonal problems. These three capabilities can be used together or in a standalone manner to improve the productivity of parallel software developers. We discuss below some different possible workflows using our tool chain, which are demonstrated in Figures 6.1- 6.5.

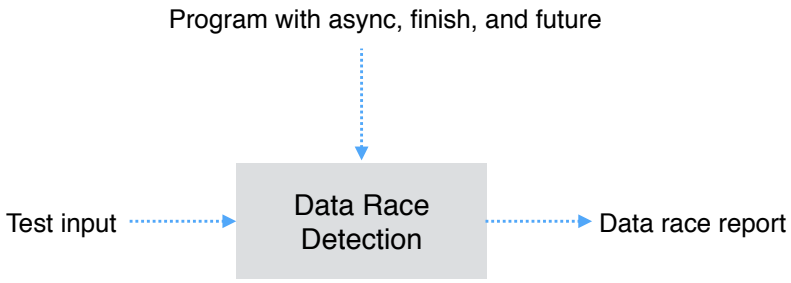

Figure 6.1 : Debugging of parallel programs with async, finish, and future

6.1 Data Race Detection and Manual Repair (DEBUG)

The programmer can use the data race detection algorithm presented in Chapter 3 to identify data races in the input program. They can then analyze the race report and manually repair the detected races by inserting additional synchronization as needed. This workflow is shown in figure 6.1.

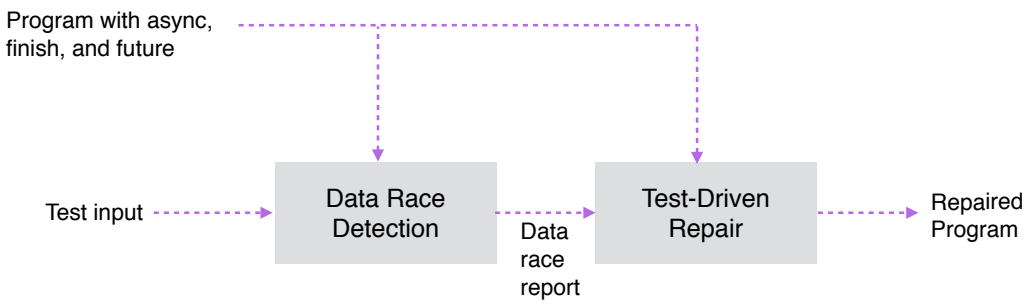

Figure 6.2 : Debugging and repair of parallel programs with async, finish, and future

6.2 Data Race Detection and Automatic Repair (DEBUG-REPAIR)

Instead of manually performing the repair, our repair algorithm presented in Chapter 4 can be used to fix the data races identified. The repair algorithm is independent of the race detection algorithm; therefore the programmer can use any race detection algorithm that produces precise race reports for parallel programs containing `async`, `finish`, and `future` constructs. This workflow is shown in Figure 6.2.

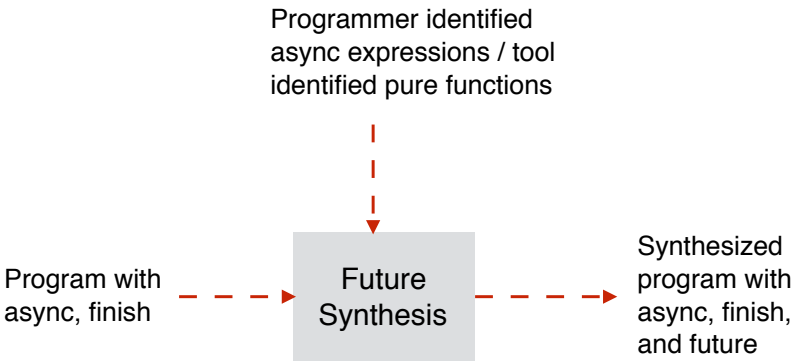

Figure 6.3 : Synthesis of futures in parallel programs

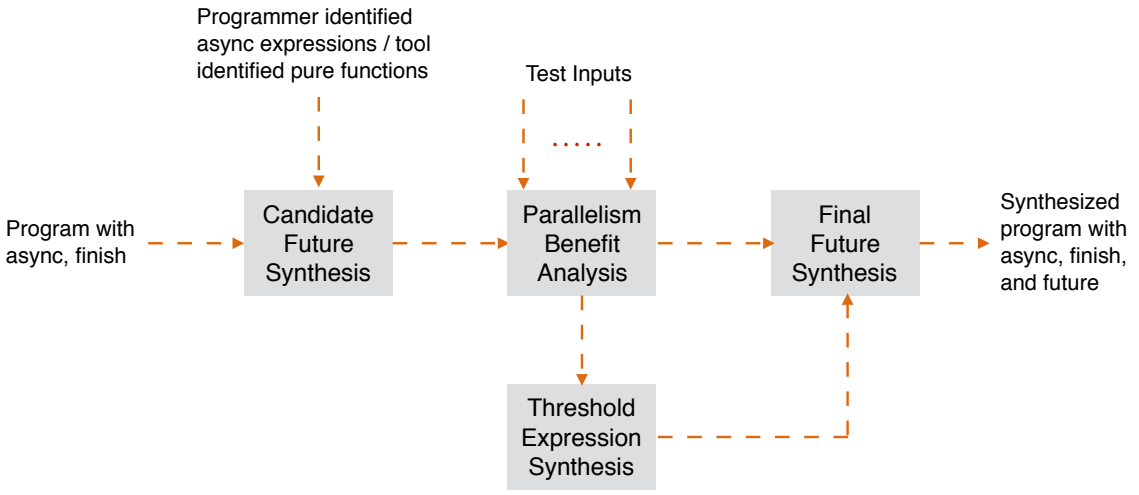

Figure 6.4 : Synthesis of futures with profitability analysis

6.3 Synthesis of Futures (SYNTHESIS)

The programmer can annotate expressions in the input program as `async` expressions, and direct our algorithm to automatically synthesize future operations and necessary type conversions using the approach presented in Chapter 5. This is shown in Figure 6.3. A synthesis workflow which includes profitability analysis is shown in Figure 6.4. Note that in this work flow, if the candidate async expressions are identified by the programmer, it is their responsibility to ensure that the candidate expressions do not contain any side effects which may cause data races. However, even for programmer specified future tasks, our approach will take care of cloning inputs to the future task to avoid violating any anti dependences on those inputs.

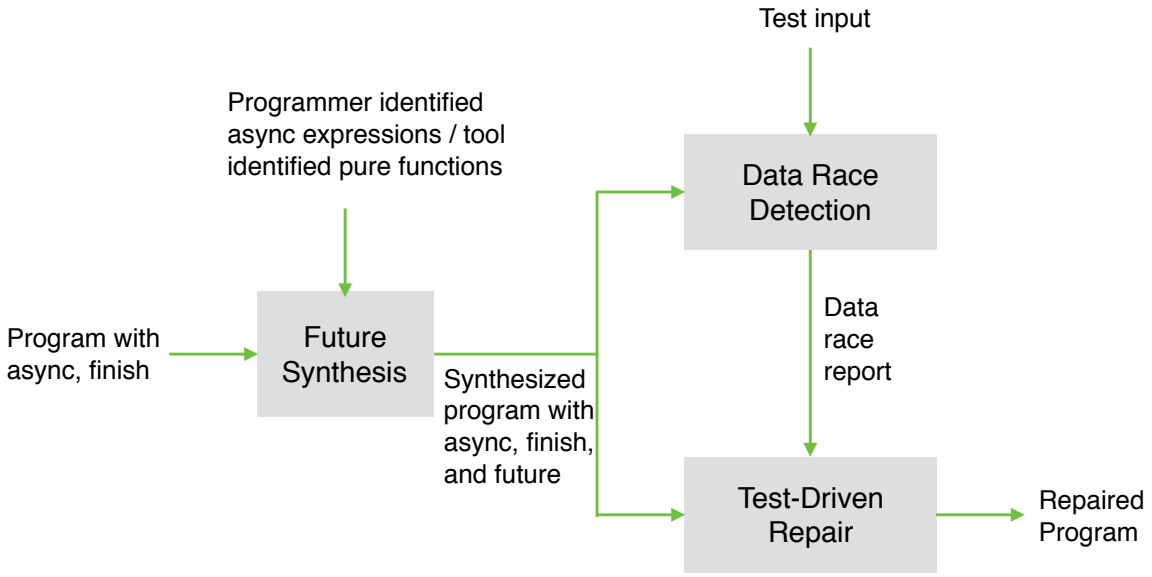

Figure 6.5 : A workflow which includes synthesis, debugging and repair of task-parallel programs

6.4 Synthesis, Data Race Detection and Automatic Repair (SYN-DEBUG-REPAIR)

A complete workflow (shown in Figure 6.5) that puts all these algorithms together would involve first annotating expressions and statements which should be asynchronously executed, synthesizing futures and type conversions using the future synthesis algorithm, and finally identifying and repairing data races by inserting necessary finish constructs. Although the repair algorithm presented in Chapter 4 targets programs with async and finish constructs, it can also be used to repair programs with async, finish, and future constructs by treating a future as an async.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Current approaches for debugging, repair, and synthesis have known limitations in functionality, performance or precision when applied to general task-parallel programs with non-strict computation graphs. In this dissertation, we presented efficient and precise algorithms for debugging, repair, and synthesis of task-parallel programs with async, finish and future constructs, using a combination of static and dynamic techniques.

We presented the first known determinacy race detector for dynamic task parallelism with futures. We presented a complexity analysis of our algorithm, and also discussed its correctness. We implemented the algorithm, and evaluated it on benchmarks which generate both strict and non-strict computations. The results indicate that the performance of our approach is comparable to that of other efficient algorithms for spawn-sync and async-finish programs and degrades gracefully in the presence of futures.

We presented a tool for test-driven repair of data races in structured parallel programs. The tool identifies static points in the program where additional synchronization is required to fix data races for a given set of input test cases. These static points obey the scoping constraints in the input program, and are selected with the goal of maximizing parallelism while still ensuring data race freedom. We evaluated an implementation of our tool on a wide variety of benchmarks which require different synchronization patterns. Our experimental results indicate that the tool is effective — for all benchmarks, the tool was able to

insert finishes to avoid data races and maximize parallelism. Further, the evaluation of the tool on student homeworks shows the potential for such tools in future offerings of parallel programming courses, including online offerings of such courses.

We presented a novel approach for automatically parallelizing pure method calls by using futures as the primary parallel construct. Given a sequential program, our algorithm automatically generates a parallel program in which pure method calls that benefit from asynchronous execution are executed as future tasks. Our approach addresses the major drawbacks of manually parallelizing programs using futures, including cloning of inputs to future tasks to break anti dependences.. When using 8 processor cores, the evaluation shows that our approach can provide significant parallel speedups of up to 7.4× (geometric mean of 3.69×) for sequential programs with zero programmer effort, beyond providing test cases for parallelism benefit analysis.

7.2 Future Work

There are many opportunities for future research to build on the results of this thesis. One direction for future work is to use a hybrid static+dynamic approach for computing the task reachability information, thereby reducing the runtime overhead of race detection. Another opportunity for future work is to support race detection in parallel programs with additional point-to-point synchronization constructs including task dependences [30, 75], doacross [76, 77], and phasers [78, 79].

One of the limitations of our repair algorithm is in analyzing long-running programs, which may lead to the creation of S-DPSTs that do not fit in memory. While this can be mitigated by using small test inputs to drive repair, one possible extension for the future is to enable garbage collection of parts of the S-DPST that do not exhibit race conditions. Some other directions for future work include generation of context sensitive finishes (where a

finish is conditionally executed only in contexts where a data race is observed), and test coverage analysis to evaluate the suitability of a given set of test cases for program repair.

Another direction for future work is to make future synthesis modular by ensuring that there is no asynchronous information flow through future objects across components. Enhancements in alias analysis could further increase the precision of type information in the synthesized program. Alternatively, our approach can be applied to dynamically typed languages for which no type declarations need to be generated. There is also room to further study the impact of exception semantics on automatic parallelization with futures, and to explore the use of runtime checks for potential exceptions in candidate future tasks. There is a promising opportunity for code motion to separate future task creation and the corresponding future `get()` operations as far as possible, so as to further increase the parallelism in the program (akin to global instruction scheduling). Yet another direction for future work is to extend our approach so that it can be applied to programs with explicit task parallelism, thereby using future tasks to further increase parallelism; it would also be interesting to perform parallelism benefit analysis and threshold expression synthesis for explicitly-parallel programs so as to aid the programmer in granularity control of their parallelism. Finally, it would be desirable (albeit challenging) to perform parallelism benefit analysis and threshold expression synthesis at runtime, so that they can be better tuned to the underlying platform.

Bibliography

[1] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen, "A theory of data race detection," in *PADTAD '06*, (New York, NY, USA), pp. 69–78, ACM, 2006.

[2] C. Flanagan and S. N. Freund, "FastTrack: efficient and precise dynamic race detection," in *PLDI '09*, (New York, NY, USA), pp. 121–133, ACM, 2009.

[3] J. Mellor-Crummey, "On-the-fly detection of data races for programs with nested fork-join parallelism," in *Supercomputing '91*, (New York, NY, USA), pp. 24–33, ACM, 1991.

[4] M. Feng and C. E. Leiserson, "Efficient detection of determinacy races in Cilk programs," in *SPAA '97*, (New York, NY, USA), pp. 1–11, ACM, 1997.

[5] 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, pp. 321–347, Dec. 2012.

[6] R. Surendran and V. Sarkar, "Dynamic determinacy race detection for task parallelism with futures," in *RV'16*, (Berlin, Heidelberg), Springer-Verlag, 2016.

[7] R. Surendran, R. Raman, S. Chaudhuri, J. Mellor-Crummey, and V. Sarkar, "Test-driven repair of data races in structured parallel programs," in *PLDI '14*, (New York, NY, USA), pp. 15–25, ACM, 2014.

[8] V. Raychev, M. T. Vechev, and E. Yahav, "Automatic synthesis of deterministic concurrency," in *SAS*, pp. 283–303, 2013.

[9] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *PLDI '11*, (New York, NY, USA), pp. 389–400, ACM, 2011.

[10] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, "Automated concurrency-bug fixing," in *OSDI'12*, (Berkeley, CA, USA), pp. 221–236, USENIX Association, 2012.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, pp. 319–349, July 1987.

[12] P. Feautrier and C. Lengauer, *Polyhedron Model*, pp. 1581–1592. Boston, MA: Springer US, 2011.

[13] L. Lamport, "The parallel execution of do loops," *Commun. ACM*, vol. 17, pp. 83–93, Feb. 1974.

[14] R. Surendran and V. Sarkar, "Automatic parallelization of pure method calls via conditional future synthesis," in *OOPSLA'16*, (New York, NY, USA), ACM, 2016.

[15] K. Ebcioğlu, V. Saraswat, and V. Sarkar, "X10: an experimental language for high productivity programming of scalable systems (extended abstract)," in *P-PHEC '05*, February 2005.

[16] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-java: the new adventures of old x10," in *PPPJ '11*, (New York, NY, USA), pp. 51–61, ACM, 2011.

[17] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 291–312, Aug. 2007.

[18] R. H. Halstead, Jr., "Multilisp: A language for concurrent symbolic computation," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 4, pp. 501–538, 1985.

[19] B. Liskov and L. Shrira, "Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems," in *PLDI '88*, (New York, NY, USA), pp. 260–267, ACM, 1988.

[20] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, pp. 720–748, Sept. 1999.

[21] S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. Yelick, "Deadlock-free scheduling of x10 computations with bounded resources," in *SPAA '07*, (New York, NY, USA), pp. 229–240, ACM, 2007.

[22] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[23] R. M. Karp and R. E. Miller, "Parallel program schemata," *J. Comput. Syst. Sci.*, vol. 3, pp. 147–195, May 1969.

[24] J. B. Dennis, G. R. Gao, and V. Sarkar, "Determinacy and repeatability of parallel program schemata," in *DFM '12*, (Washington, DC, USA), pp. 1–9, IEEE Computer Society, 2012.

[25] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall, "Dag-consistent distributed shared memory," in *IPPS '96*, (Washington, DC, USA), pp. 132–141, IEEE Computer Society, 1996.

[26] 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 *SPAA '04*,

(New York, NY, USA), pp. 133–144, ACM, 2004.

[27] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Scalable and precise dynamic datarace detection for structured parallelism," in *PLDI '12*, (New York, NY, USA), pp. 531–542, ACM, 2012.

[28] D. Dimitrov, M. Vechev, and V. Sarkar, "Race detection in two dimensions," in *SPAA '15*, (New York, NY, USA), pp. 101–110, ACM, 2015.

[29] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *PPoPP '95*, (New York, NY, USA), pp. 207–216, ACM, 1995.

[30] "OpenMP specifications." `http://www.openmp.org/specs`.

[31] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.

[32] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *J. ACM*, vol. 22, pp. 215–225, Apr. 1975.

[33] P. Dietz and D. Sleator, "Two algorithms for maintaining order in a list," in *STOC '87*, (New York, NY, USA), pp. 365–372, ACM, 1987.

[34] V. K. Nandivada, J. Shirako, J. Zhao, and V. Sarkar, "A transformation framework for optimizing task-parallel programs," *ACM Trans. Program. Lang. Syst.*, vol. 35, Apr. 2013.

[35] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *CASCON '99*, IBM Press, 1999.

[36] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey, "A benchmark suite for high performance java," *Concurrency: Practice and Experience*, vol. 12, no. 6, pp. 375–388, 2000.

[37] P. Virouleau, P. Brunet, F. Broquedis, N. Furmento, S. Thibãult, O. Aumage, and T. Gautier, "Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite," in *IWOMP 2014*, pp. 16–29, 2014.

[38] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Efficient data race detection for async-finish parallelism," in *RV'10*, (Berlin, Heidelberg), pp. 368–383, Springer-Verlag, 2010.

[39] A. Dinning and E. Schonberg, "An empirical comparison of monitoring algorithms for access anomaly detection," in *PPOPP '90*, (New York, NY, USA), pp. 1–10, ACM, 1990.

[40] M. Vechev, E. Yahav, and G. Yorsh, "Abstraction-guided synthesis of synchronization," in *POPL '10*, (New York, NY, USA), pp. 327–338, ACM, 2010.

[41] P. Černý, K. Chatterjee, T. A. Henzinger, A. Radhakrishna, and R. Singh, "Quantitative synthesis for concurrent programs," in *CAV'11*, (Berlin, Heidelberg), pp. 243–259, Springer-Verlag, 2011.

[42] A. Solar-Lezama, C. G. Jones, and R. Bodik, "Sketching concurrent data structures," in *PLDI*, pp. 136–148, 2008.

[43] D. Kelk, K. Jalbert, and J. S. Bradbury, "Automatically repairing concurrency bugs with ARC," in *Multicore Software Engineering, Performance, and Tools*, pp. 73–84, Springer, 2013.

[44] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 54–72, 2012.

[45] A. Griesmayer, R. Bloem, and B. Cook, "Repair of Boolean programs with an application to C," in *CAV*, pp. 358–371, Springer, 2006.

[46] F. Logozzo and T. Ball, "Modular and verified automatic program repair," in *OOPSLA '12*, (New York, NY, USA), pp. 133–146, ACM, 2012.

[47] A. Sălcianu and M. Rinard, "Purity and side effect analysis for Java programs," in *VMCAI'05*, (Berlin, Heidelberg), pp. 199–215, Springer-Verlag, 2005.

[48] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst, "Reim & Reiminfer: Checking and inference of reference immutability and method purity," in *OOPSLA '12*, (New York, NY, USA), pp. 879–896, ACM, 2012.

[49] J. Miettinen, "Computer language benchmarks game." `http://benchmarksgame.alioth.debian.org/u64q/program.php?test=binarytrees&lang=java&id=6`. Accessed: 2015-11-06.

[50] P. Pratikakis, J. Spacco, and M. Hicks, "Transparent proxies for Java futures," in *OOPSLA '04*, (New York, NY, USA), pp. 206–223, ACM, 2004.

[51] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *POPL '95*, (New York, NY, USA), pp. 49–61, ACM, 1995.

[52] A. Diwan, K. S. McKinley, and J. E. B. Moss, "Type-based alias analysis," in *PLDI '98*, (New York, NY, USA), pp. 106–117, ACM, 1998.

[53] C. Spearman, "The proof and measurement of association between two things," *American Journal of Psychology*, vol. 15, pp. 88–103, 1904.

[54] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *FMCAD 2013*, pp. 1–8, 2013.

[55] "Specjvm2008." `https://www.spec.org/jvm2008/`.

[56] B. Cahoon and K. S. McKinley, "Data flow analysis for software prefetching linked data structures in Java," in *PACT '01*, (Washington, DC, USA), pp. 280–291, IEEE Computer Society, 2001.

[57] E. Bodden, "Inter-procedural data-flow analysis with IFDS/IDE and Soot," in *SOAP '12*, (New York, NY, USA), pp. 3–8, ACM, 2012.

[58] C. Flanagan and M. Felleisen, "The semantics of future and its use in program optimization," in *POPL '95*, (New York, NY, USA), pp. 209–220, ACM, 1995.

[59] A. Welc, S. Jagannathan, and A. Hosking, "Safe futures for Java," in *OOPSLA '05*, (New York, NY, USA), pp. 439–453, ACM, 2005.

[60] A. Navabi, X. Zhang, and S. Jagannathan, "Quasi-static scheduling for safe futures," in *PPoPP '08*, (New York, NY, USA), pp. 23–32, ACM, 2008.

[61] J. Swaine, K. Tew, P. Dinda, R. B. Findler, and M. Flatt, "Back to the futures: Incremental parallelization of existing sequential runtime systems," in *OOPSLA '10*, (New York, NY, USA), pp. 583–597, ACM, 2010.

[62] T. Harris and S. Singh, "Feedback directed implicit parallelism," in *ICFP '07*, (New York, NY, USA), pp. 251–264, ACM, 2007.

[63] L. Zhang, C. Krintz, and P. Nagpurkar, "Language and virtual machine support for efficient fine-grained futures in Java," in *PACT '07*, (Washington, DC, USA), pp. 130–139, IEEE Computer Society, 2007.

[64] L. Zhang, C. Krintz, and P. Nagpurkar, "Supporting exception handling for futures in Java," in *PPPJ '07*, (New York, NY, USA), pp. 175–184, ACM, 2007.

[65] A. Navabi and S. Jagannathan, "Exceptionally safe futures," in *COORDINATION '09*, (Berlin, Heidelberg), pp. 47–65, Springer-Verlag, 2009.

[66] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval, "How much parallelism is there in irregular applications?," in *PPoPP '09*, (New York, NY, USA), pp. 3–14, ACM, 2009.

[67] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor, "Kremlin: Rethinking and rebooting gprof for the multicore age," in *PLDI '11*, (New York, NY, USA), pp. 458–469, ACM, 2011.

[68] Y. He, C. E. Leiserson, and W. M. Leiserson, "The cilkview scalability analyzer," in *SPAA '10*, (New York, NY, USA), pp. 145–156, ACM, 2010.

[69] T. B. Schardl, B. C. Kuszmaul, I.-T. A. Lee, W. M. Leiserson, and C. E. Leiserson, "The cilkprof scalability profiler," in *SPAA '15*, (New York, NY, USA), pp. 89–100, ACM, 2015.

[70] E. Coppa, C. Demetrescu, and I. Finocchi, "Input-sensitive profiling," in *PLDI '12*, (New York, NY, USA), pp. 89–98, ACM, 2012.

[71] E. Coppa, C. Demetrescu, I. Finocchi, and R. Marotta, "Estimating the empirical cost function of routines with dynamic workloads," in *CGO '14*, (New York, NY, USA),

pp. 230–239, ACM, 2014.

[72] D. Zaparanuks and M. Hauswirth, "Algorithmic profiling," in *PLDI '12*, (New York, NY, USA), pp. 67–76, ACM, 2012.

[73] A. Duran, J. Corbalán, and E. Ayguadé, "An adaptive cut-off for task parallelism," in *SC '08*, (Piscataway, NJ, USA), pp. 36:1–36:11, IEEE Press, 2008.

[74] P. Thoman, H. Jordan, and T. Fahringer, "Adaptive granularity control in task parallel programs using multiversioning," in *Euro-Par'13*, (Berlin, Heidelberg), pp. 164–177, Springer-Verlag, 2013.

[75] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC '12*, (Los Alamitos, CA, USA), pp. 66:1–66:11, IEEE Computer Society Press, 2012.

[76] J. Shirako, P. Unnikrishnan, S. Chatterjee, K. Li, and V. Sarkar, "Expressing DOACROSS Loop Dependencies in OpenMP," in *IWOMP'13*, Sep 2013.

[77] P. Unnikrishnan, J. Shirako, K. Barton, S. Chatterjee, R. Silvera, and V. Sarkar, "A practical approach to doacross parallelization," in *Euro-Par'12*, (Berlin, Heidelberg), pp. 219–231, Springer-Verlag, 2012.

[78] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, "Phasers: a unified deadlock-free construct for collective and point-to-point synchronization," in *ICS '08*, (New York, NY, USA), pp. 277–288, ACM, 2008.

[79] J. Shirako, K. Sharma, and V. Sarkar, "Unifying barrier and point-to-point synchronization in OpenMP with phasers," in *IWOMP'11*, (Berlin, Heidelberg), pp. 122–137, Springer-Verlag, 2011.