

RICE UNIVERSITY

**Dynamic Data Race Detection for Structured  
Parallelism**

by

**Raghavan Raman**

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

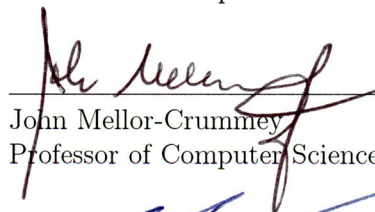
**Doctor of Philosophy**

APPROVED, THESIS COMMITTEE:



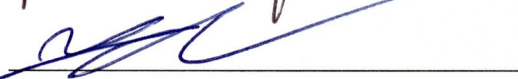
---

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



---

John Mellor-Crummey  
Professor of Computer Science



---

Lin Zhong  
Associate Professor of Electrical and  
Computer Engineering

Houston, Texas

August, 2012

## ABSTRACT

### Dynamic Data Race Detection for Structured Parallelism

by

Raghavan Raman

With the advent of multicore processors and an increased emphasis on parallel computing, parallel programming has become a fundamental requirement for achieving available performance. Parallel programming is inherently hard because, to reason about the correctness of a parallel program, programmers have to consider large numbers of interleavings of statements in different threads in the program. Though structured parallelism imposes some restrictions on the programmer, it is an attractive approach because it provides useful guarantees such as deadlock-freedom. However, data races remain a challenging source of bugs in parallel programs. Data races may occur only in few of the possible schedules of a parallel program, thereby making them extremely hard to detect, reproduce, and correct. In the past, dynamic data race detection algorithms have suffered from at least one of the following limitations: some algorithms have a worst-case *linear space and time overhead* [1], some algorithms are *dependent on a specific scheduling technique* [2], some algorithms generate *false positives and false negatives* [3, 4], some have *no empirical evaluation* as yet [2], and some require *sequential execution* of the parallel program [5, 6].

In this thesis, we introduce dynamic data race detection algorithms for structured parallel programs that overcome past limitations. We present a race detection algorithm called ESP-bags that requires the input program to be executed sequentially

and another algorithm called SPD3 that can execute the program in parallel. While the ESP-bags algorithm addresses all the above mentioned limitations except sequential execution, the SPD3 algorithm addresses the issue of sequential execution by scaling well across highly parallel shared memory multiprocessors. Our algorithms incur constant space overhead per memory location and time overhead that is independent of the number of processors on which the programs execute. Our race detection algorithms support a rich set of parallel constructs (including `async`, `finish`, `isolated`, and `future`) that are found in languages such as HJ, X10, and Cilk. Our algorithms for `async`, `finish`, and `future` are precise and sound for a given input. In the presence of `isolated`, our algorithms are precise but not sound. Our experiments show that our algorithms (for `async`, `finish`, and `isolated`) perform well in practice, incurring an average slowdown of under  $3\times$  over the original execution time on a suite of 15 benchmarks. SPD3 is the first practical dynamic race detection algorithm for `async-finish` parallel programs that can execute the input program in parallel and use constant space per memory location. This takes us closer to our goal of building dynamic data race detectors that can be “always-on” when developing parallel applications.

## Acknowledgments

I would like to express my deepest gratitude to my advisor, Prof. Vivek Sarkar, for his support and guidance throughout this dissertation. He has been a constant source of inspiration for me in many ways. His enthusiasm in solving problems is infectious. He gave me the freedom to express myself in my work all along while also guiding me in the right direction. He has always been ready to help in both academic and personal issues. I would like to thank him specifically for his support during the troubled times in my PhD. This dissertation would not have been possible without him. I could not have asked for a better advisor.

My sincere thanks to Prof. John Mellor-Crummey for agreeing to be on my thesis committee and for his feedback and suggestions at various stages during my dissertation. His suggestions to improve parts of my dissertation were very useful. I would also like to thank him for collaborating on extending the parallel data race detector for futures. I enjoyed all the technical discussions that I have had with him. The effort he puts in to get everything right is truly amazing. I have learned a lot from him.

I would like to extend my sincere thanks to Prof. Lin Zhong for agreeing to be on my thesis committee and for his enthusiasm in learning about my work. His insights and feedback were very useful in improving my thesis.

I express my sincere gratitude to Prof. Martin Vechev (ETH) and Prof. Eran Yahav (Technion) with whom I have had the pleasure of collaborating for the past three years. A major part of this dissertation was a result of my collaboration with them. I learned a lot from them right from my visit to IBM Research in 2009. I have enjoyed every bit of my interactions with them. It has always been fun working with them. Many thanks to Prof. Vivek Sarkar for introducing me to them.

I am grateful to Jisheng Zhao for all the collaborative work over the past few years. It was a pleasure working with him on all the publications we co-authored. A special thanks to him for implementing the static optimizations that are used in this

work.

I would like to thank my fellow graduate students David Peixotto and Rajkishore Barik for helping me during my early days at Rice. Being my office mate, Dave was forced to listen to everything I had to say and also to answer all my questions. A special thanks to him for putting up with me all along. I would also like to thank Prof. Keith Cooper whose frequent visits to our office came as a relief on stressful days. Dave and I enjoyed listening to many fun stories that he had.

My hearty thanks to every member (past and present) of the Habanero group for all the interactions that I have had with them. I have enjoyed every bit being part of the Habanero group. I would like to thank everyone who patiently listened and gave me feedback on some of my talks that were repeated multiple times.

A special thanks to my beloved wife, Vanitha Vijayaraghavan, for supporting me throughout and for putting up with my long night hours at work. I can never forget those innumerable visits to Starbucks well past midnight while writing my thesis. Thanks to her for staying up all night regularly giving me company in writing my thesis and also for all the yummy food served right at my work table. Her love, support, and encouragement were an important reason that I was able to complete my PhD sooner. Now that I am done, I hope to give her back in many-fold for all these. My sincere thanks to my parents who have been very supportive and encouraging all along during my graduate course. They have been patiently and eagerly waiting for me to complete my PhD. My deepest gratitude to them without whom this work would not have been possible. I would also like to thank my in-laws, uncles, and aunts for trusting and encouraging me and also for patiently waiting for me to cross the line.

Many thanks to my friends Jeyarama Ananta, Kaushik Kumar Ram, and Ramkumar Krishnamurthy for making my stay at Rice very enjoyable and for being there during my troubled times. I cant imagine my state had they not been around during those times.

I would also like to thank everyone who has helped me directly or indirectly during my graduate study at Rice.

Finally, I thank the Almighty for giving me the confidence to pursue my dreams and the strength to complete my journey.

# Contents

Abstract	ii
List of Illustrations	xi
List of Tables	xv
<b>1 Introduction</b>	<b>1</b>
1.1 Thesis Statement . . . . .	4
1.2 Research Contributions . . . . .	4
1.3 Thesis Organization . . . . .	5
<b>2 Background</b>	<b>7</b>
2.1 Parallel Programming Models . . . . .	7
2.1.1 Three Approaches to Parallel Programming . . . . .	8
2.1.2 Structured and Unstructured Models . . . . .	9
2.2 Habanero Java: A Structured Parallel Programming Model . . . . .	9
2.2.1 Parallel Constructs . . . . .	10
2.2.2 Comparison with Cilk . . . . .	14
2.3 Data Races . . . . .	17
2.3.1 Data Race Detection . . . . .	18
2.3.2 Guarantees of Data Race Detection Algorithms . . . . .	21
2.3.3 Data Race Detection for Structured Parallelism . . . . .	22
2.3.4 Past Work on Data Race Detection . . . . .	23
<b>3 Sequential Data Race Detection for HJ</b>	<b>26</b>

3.1	Background: SP-bags for Fully-Strict Computations . . . . .	27
3.2	ESP-bags: A Generalization of SP-bags for Terminally-Strict Computations . . . . .	32
3.2.1	ESP-bags for Async-Finish . . . . .	33
3.2.2	Extending ESP-bags for Isolated Blocks . . . . .	42
3.3	ESP-bags for Labeled-Finish . . . . .	48
3.4	ESP-bags for Single-Get Futures . . . . .	51
3.4.1	A Maximal Generalization of ESP-bags . . . . .	55
3.5	Summary . . . . .	56

## **4 SPD3: A Parallel Algorithm for Detecting Data Races in HJ** **58**

4.1	SPD3 for Async-Finish . . . . .	59
4.1.1	Dynamic Program Structure Tree . . . . .	59
4.1.2	Shadow Memory . . . . .	70
4.1.3	SPD3 Algorithm . . . . .	72
4.1.4	Soundness and Precision . . . . .	76
4.1.5	Space Overhead . . . . .	76
4.1.6	Time Overhead . . . . .	77
4.2	Extending SPD3 for Isolated Blocks . . . . .	78
4.2.1	Shadow Memory with Isolated . . . . .	79
4.2.2	Extended SPD3 Algorithm . . . . .	80
4.2.3	Soundness and Precision . . . . .	86
4.2.4	Space Overhead . . . . .	86
4.2.5	Time Overhead . . . . .	87
4.3	Extending SPD3 for Futures . . . . .	87
4.3.1	Dynamic Program Structure Tree with Futures . . . . .	88
4.3.2	Enhancing the DPST with Futures . . . . .	95



4.3.3	Shadow Memory with Futures . . . . .	101
4.3.4	Extended SPD3 Algorithm . . . . .	103
4.3.5	Soundness and Precision . . . . .	107
4.3.6	Space Overhead . . . . .	107
4.3.7	Time Overhead . . . . .	108
<b>5</b>	<b>Correctness Proofs</b>	<b>109</b>
5.1	ESP-bags for Async-Finish . . . . .	109
5.2	SPD3 for Async-Finish . . . . .	114
<b>6</b>	<b>Implementation</b>	<b>125</b>
6.1	Design of Data Race Detectors . . . . .	125
6.2	Disjoint-set in ESP-bags . . . . .	128
6.3	DPST in SPD3 . . . . .	129
6.4	Relaxing the Atomicity Requirement in SPD3 . . . . .	131
<b>7</b>	<b>Static Optimizations for Data Race Detection</b>	<b>135</b>
7.1	Main Task Check Elimination in Sequential Code Regions . . . . .	136
7.2	Read-only Check Elimination in Parallel Code Regions . . . . .	138
7.3	Escape Analysis . . . . .	140
7.4	Loop Invariant Check Motion . . . . .	140
7.5	Read/Write Check Elimination . . . . .	142
<b>8</b>	<b>Experimental Results</b>	<b>144</b>
8.1	Experimental Setup . . . . .	144
8.2	Data Races Observed . . . . .	146
8.3	Evaluation of ESP-bags for Async-Finish-Isolated . . . . .	147
8.3.1	Performance of ESP-bags . . . . .	147
8.3.2	Performance of Static Optimizations . . . . .	149

8.3.3	Comparison with Serialized Execution . . . . .	153
8.4	Evaluation of SPD3 for Async-Finish-Isolated . . . . .	155
8.4.1	Performance of SPD3 . . . . .	155
8.4.2	Comparison of SPD3 with ESP-bags . . . . .	163
8.4.3	Comparison of SPD3 with Eraser and FastTrack . . . . .	165
<b>9</b>	<b>Related Work</b>	<b>173</b>
9.1	Dynamic Data Race Detection . . . . .	173
9.1.1	Lockset based Algorithms . . . . .	174
9.1.2	Happens-Before based Algorithms . . . . .	175
9.2	Static Analysis for Data Race Detection and Avoidance . . . . .	183
9.2.1	Static Analysis to Improve Dynamic Race Detectors . . . . .	185
9.3	Data Race Detection in Hardware . . . . .	186
9.4	Determinism Checking . . . . .	188
<b>10</b>	<b>Conclusions and Future Work</b>	<b>191</b>
	<b>Bibliography</b>	<b>195</b>
<b>A</b>	<b>Anomaly in Crypt with Optimizations</b>	<b>212</b>

## Illustrations

2.1	An example HJ program and its computation graph. This code is the body of the main method in the program. The computation graph represents an execution where the <code>for</code> loop in line 5 executes only once.	13
3.1	Action to be taken on read and write of shared memory locations. This action applies to the SP-bags and the ESP-bags algorithm. . . .	30
3.2	A program where a data race would be missed if the <i>reader</i> field of the memory location “x” is updated to the new reader while the previous one was in a P-bag. . . . .	31
3.3	A sample HJ program with <code>future</code> (without the <code>final</code> restriction) for which sequential depth-first execution is not possible. . . . .	33
3.4	The computation graph from Figure 2.1. The S and P bags of all the tasks and finishes when statement 20 executes are also shown here. Statement 20 may execute in parallel with $T_2$ and $T_4$ . So, when statement 20 executes, both $T_2$ and $T_4$ are in a P-bag. But statement 20 can never execute in parallel with the statements already executed in $T_1$ and $T_3$ . So, when statement 20 executes, $T_1$ and $T_3$ are in an S-bag. . . . .	37
3.5	A sample HJ program with data races to show the lack of soundness and precision guarantees beyond the first data race. . . . .	40
3.6	Actions to be taken on isolated and non-isolated read, write operations on shared memory locations for the ESP-bags algorithm. .	43

3.7	An example HJ program that depicts a scenario in which ESP-bags is not <i>sound</i> in the presence of <i>isolated</i> . . . . .	46
3.8	A sample program to depict the usage of the <i>labeled-finish</i> construct. . . . .	49
3.9	A sample program to depict the usage of <i>single-get</i> futures. . . . .	52
4.1	A sample program in HJ with <i>async</i> , <i>finish</i> , statements. The statements <i>s1 - s13</i> are grouped in to steps <i>S1 - S6</i> . The tree on the right is the DPST corresponding to this program. . . . .	62
4.2	A part of a DPST. LCA is the Lowest Common Ancestor of steps <i>S1</i> and <i>S2</i> . <i>A</i> is the DPST ancestor of <i>S1</i> which is the child of LCA. <i>S1</i> and <i>S2</i> can execute in parallel if and only if <i>A</i> is an <i>async</i> node. . . . .	66
4.3	An example DPST with <i>async</i> , <i>finish</i> , and <i>future</i> nodes. <i>F1</i> and <i>F2</i> are <i>finish</i> nodes; <i>G1</i> , <i>G2</i> , and <i>G3</i> are <i>future</i> nodes; <i>A1</i> , <i>A2</i> , <i>A3</i> , and <i>A4</i> are non- <i>future async</i> nodes. The nodes <i>S1-S7</i> refer to the <i>step</i> nodes. The dotted lines denote the pointers from <i>future</i> nodes to their <i>get</i> operations. . . . .	92
4.4	An example of the enhanced DPST with <i>async</i> , <i>finish</i> , and <i>future</i> nodes. This is the enhanced version of the DPST in Figure 4.3. <i>F1</i> and <i>F2</i> are <i>finish</i> nodes; <i>G1</i> , <i>G2</i> , and <i>G3</i> are <i>future</i> nodes; <i>A1</i> , <i>A2</i> , <i>A3</i> , and <i>A4</i> are non- <i>future async</i> nodes. The nodes <i>S1-S7</i> refer to the <i>step</i> nodes. . . . .	99
6.1	HJ System Architecture with Race Detection . . . . .	126
7.1	An example HJ program with all read and write operations instrumented . . . . .	137
7.2	After applying the main task check elimination optimization on the program in Figure 7.1 . . . . .	137

7.3	After applying the read-only check optimization on the program in Figure 7.2 . . . . .	139
7.4	After applying the escape analysis and check elimination optimization on the program in Figure 7.3 . . . . .	139
7.5	After applying the loop invariant check elimination optimization on the program in Figure 7.4 . . . . .	141
7.6	After applying the read/write check elimination optimization on the program in Figure 7.5 . . . . .	141
8.1	Breakdown of the effect of static optimizations on the performance improvement of the ESP-bags algorithm. . . . .	151
8.2	Comparison of the slowdown of ESP-bags using the 1-thread execution with the slowdown of ESP-bags using the serialized execution	154
8.3	Relative slowdown of SPD3 for all benchmarks on 1, 2, 4, 8, and 16 threads. Relative slowdown on $n$ threads refers to the slowdown of the SPD3 version on $n$ threads compared to the HJ-Base version on $n$ threads. . . . .	156
8.4	Breakdown of the effect of static optimizations on the performance improvement of the SPD3 algorithm while executing on 16-threads. . . . .	160
8.5	Comparison of the performance of the SPD3 algorithm based on “Synchronized” with that of SPD3 based on “CompareAndSet” while executing on 16-threads. . . . .	162
8.6	Slowdown of ESP-bags and SPD3 relative to 16-thread HJ-Base version for all benchmarks. Note that the ESP-bags version runs on 1-thread while the SPD3 version runs on 16-threads. . . . .	164
8.7	Slowdown (relative to 16-threads RR-Base) of RR-Base, Eraser, FastTrack, HJ-Base, and SPD3 for Crypt benchmark (chunked version) on 1-16 threads . . . . .	168

8.8	Estimated heap memory usage (in MB) of RR-Base, Eraser, FastTrack, HJ-Base, and SPD3 for LUFact benchmark . . . . .	171
-----	--	-----

## Tables

3.1	SP-bags: Rules to update S and P bags at Task Boundaries . . . . .	29
3.2	ESP-bags for Async-Finish: Rules to update S and P bags at Task Boundaries . . . . .	34
3.3	Example of updating the S and P bags during ESP-bags algorithm on the program corresponding to the computation graph in Figure 3.4. ‘-N’ in the PC column indicates that the bags are updated before the execution of statement N. ‘N-’ in the PC column indicates that the bags are updated after the execution of statement N. ‘N’ in the PC column indicates that reader/writer field of a memory location is updated before the corresponding memory operation in statement N. ‘*N’ in the PC column indicates that a data race is signaled as a result of checking and updating the S and P bags due to a memory operation in N. . . . .	38
3.4	ESP-bags for Labeled-Finish: Rules to update S and P bags at Task Boundaries . . . . .	50
3.5	ESP-bags for Single-Get Future: Rules to update S and P bags at Task Boundaries . . . . .	53
8.1	List of Benchmarks Evaluated . . . . .	145
8.2	Slowdown of the ESP-bags algorithm relative to uninstrumented 1-thread execution time on a work-first work-stealing runtime . . . . .	148

8.3	Slowdown of the SPD3 algorithm on 1-thread and 16-threads relative to uninstrumented 1-thread and 16-thread execution times respectively on an adaptive work-stealing runtime . . . . .	158
8.4	Relative slowdown of Eraser, FastTrack and SPD3 for JGF benchmarks on 16 threads. The slowdown of Eraser and FastTrack was calculated relative to RR-Base while the slowdown of SPD3 was calculated relative to HJ-Base. For benchmarks marked with *, race-free versions were used for SPD3 but the original (buggy) versions were used for Eraser and FastTrack. . . . .	167
8.5	Peak heap memory usage of RR-Base, Eraser, FastTrack, HJ-Base, and SPD3 for JGF benchmarks on 16 threads. For benchmarks marked with *, race-free versions were used for SPD3 but the original versions were used for Eraser and FastTrack. . . . .	170
9.1	A comparison of related work. OTFDAA refers to “On The Fly Detection of Access Anomalies”, n refers to the number of threads executing the program and N refers to the maximum logical concurrency in the program. . . . .	181
A.1	Crypt execution repeated for 30 iterations . . . . .	213
A.2	Crypt execution repeated for 30 iterations with additional JVM parameters: <code>-XX:MaxPermSize=256m -XX:+UseParallelGC -XX:+UseParallelOldGC -XX:NewRatio=2 -XX:-UseGCOverheadLimit</code> . . . . .	215



# Chapter 1

## Introduction

Parallel processors have become a ubiquitous part of computer technology. Nowadays, parallel processors are present everywhere, from clusters and servers to tablets and mobile phones. With the increasing demand for parallelism, parallel processors are set to dominate the computer industry for the foreseeable future.

To exploit the potential of parallel processors, applications must run in parallel. While automatic parallelization of existing sequential programs seems an attractive option from the programmer's perspective, it is not very effective due to the limitations of dependence analysis of parallel programs. In effect, the gains offered by automatic parallelization across the spectrum of applications aren't enough to keep today's multicore processors busy. In the future, with processors expected to have hundreds of cores, it will be more difficult to utilize their full potential with automatic parallelization.

To address the concerns of automatic parallelization, many solutions have been proposed where programmers add annotations or pragmas to help the compiler in the parallelization process. At the other end of the spectrum are explicitly parallel programming models, which expect the programmer to specify parallelism in the program completely using explicit parallel constructs. This technique has great potential because it places less of a burden on the compiler and is limited only by the ability of the programmer to unearth and specify parallelism in the application.

Explicit parallel programming models range from low-level frameworks such as

**Pthreads** and **Java threads**, which are unstructured, to high-level languages with structured parallelism such as **Cilk** [7], **X10** [8] and **HabaneroJava (HJ)** [9]. Unstructured low-level frameworks give more power to the programmer than the structured high-level languages. On the other hand, it is easier to reason about a structured parallel program than an unstructured one. The main restriction placed on structured parallel programs is based on the way tasks can join with other tasks. Despite these restrictions, structured parallelism is expressive enough for a wide range of applications. The emergence of structured parallelism to enable better and more productive parallel programming as compared to the unstructured parallel programming models is analogous to the emergence of structured programming, a few decades ago, as a more productive programming paradigm over the unstructured programming using `gotos`.

It is often difficult for programmers to write correct and efficient parallel programs because it requires reasoning about the interleavings of statements in different threads in the program. The complexity of reasoning about interleavings of statements can increase exponentially with the number of threads in the program. Data races are a major cause for this difficulty. Data races may lead to unintended, undesirable, and non-deterministic behavior in parallel programs. Typically, data races occur only in some of the possible schedules of a parallel program and hence, it is very difficult to detect, reproduce, and correct data races. This is especially true in the case of structured parallelism where programs typically consist of a large number of lightweight tasks that are executed by a smaller number of worker threads and a data race between accesses in two tasks does not manifest unless they run on different worker threads.

Detecting data races correctly and efficiently helps programmers identify bugs in

their parallel programs quickly and hence, improves their productivity. The importance of data race detection is evident from the attention this problem has received in the literature for the past few decades. Data race detectors can be classified into *static* and *dynamic* based on the type of analysis they perform to detect data races. A static data race detector analyzes the parallel program statically at compile time to detect data races. In contrast, a dynamic data race detector analyzes an execution of the program to detect data races at runtime. We focus on dynamic data race detectors in this thesis.

In the past, dynamic data race detection algorithms have suffered from at least one of the following limitations: some algorithms have a worst-case *linear space and time overhead* [1], some algorithms are *dependent on a scheduling technique* [2], some algorithms generate *false positives and/or false negatives* [3, 4], some have *no empirical evaluation* as yet [2], and some require *sequential execution* of the parallel program [5, 6].

In this thesis, we present efficient and useful dynamic data race detection algorithms for structured parallel programs that overcome past limitations. We present a race detection algorithm called ESP-bags that requires the input program to be executed sequentially [10] and another algorithm called SPD3 that can execute the program in parallel [11]. While the ESP-bags algorithm addresses all the above mentioned limitations except sequential execution, the SPD3 algorithm addresses the issue of sequential execution as well. We also present some static optimizations to eliminate redundant instrumentation for race detection so as to reduce the overhead of these algorithms.

We target a rich set of parallel constructs present in today’s structured parallel programming languages: `async`, which is used to create tasks that can run in parallel,

`finish`, which specifies a join point for a subset of tasks, `isolated`, that is used for mutual exclusion, and `future`, which creates a task with a handle that can be used to wait for this task specifically. Our experiments show that our algorithms (for `async`, `finish`, and `isolated`) perform well in practice, incurring an average slowdown of under  $3\times$  over the original execution time on a suite of 15 benchmarks.

Our algorithms for `async`, `finish`, and `future` are precise and sound for a given input. In the presence of `isolated`, our algorithms are precise but not sound. Our algorithms are efficient because they use less space and time as compared to some state-of-the-art race detection algorithms. Specifically, our algorithms require only constant space for every memory location that is monitored (for `async`, `finish`, and `isolated` constructs) as compared to the linear space required for the state-of-the-art race detector [1]. The time overhead of our algorithms is a characteristic of the application and is independent of the number of processors (i.e., worker threads) on which the application executes. Hence our algorithms *scale* very well in space and time and are more practical and useful for highly parallel systems than past approaches.

## 1.1 Thesis Statement

Structured parallelism can enable dynamic data race detection to be performed efficiently in parallel on real-world multiprocessors with constant space overhead.

## 1.2 Research Contributions

This dissertation makes the following research contributions:

1. the ESP-bags algorithm, which is a sequential data race detection algorithm

for programs with `async` and `finish`. The ESP-bags algorithm is an adaptation of the SP-bags algorithm [5] that was designed for data race detection in Cilk programs. We also extend the ESP-bags algorithm for programs with `isolated`, `labeled-finish`, and a restricted form of `futures`.

2. the SPD3 algorithm, which is a parallel data race detection algorithm for programs with `async` and `finish`. This algorithm uses a new data structure called the Dynamic Program Structure Tree (DPST) to maintain parent-child relationships between `async` and `finish` instances during program execution and also uses a constant-size access summary for every memory location. We also extend the SPD3 algorithm for programs with `isolated` and `futures`.
3. a set of static optimizations to reduce the runtime overhead of dynamic data race detectors.
4. an implementation of the ESP-bags and the SPD3 algorithms for programs with `async`, `finish`, and `isolated`. This also includes a technique to relax the atomicity requirement for parallel updates to a shadow memory location in the SPD3 algorithm. We also present a comprehensive evaluation of our implementation and a comparison with other dynamic data race detectors.

### 1.3 Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2 describes the background on structured parallel programs, explains how data race detection for structured parallel programs is different, and also briefly discusses related work.

- Chapter 3 describes the ESP-bags algorithm, which is the sequential data race detection algorithm for async-finish, and also its extensions to isolated, labeled-finish, and a restricted form of futures.
- Chapter 4 describes the SPD3 algorithm, which is the parallel data race detection algorithm for async-finish, and also its extensions to isolated and futures.
- Chapter 5 proves the correctness of the ESP-bags and the SPD3 algorithms for async-finish programs.
- Chapter 6 explains the implementation details of these algorithms and the corresponding instrumentation needed to use these algorithms with structured parallel programs.
- Chapter 7 describes the static optimizations that can be performed to reduce the overhead of dynamic data race detection algorithms.
- Chapter 8 presents the evaluation of the ESP-bags and the SPD3 algorithms on a suite of benchmarks and also gives the implications of these results.
- Chapter 9 discusses the related work from the past and how this thesis compares with them.
- Chapter 10 presents the conclusions and also discusses some potential topics to be explored in future.

## Chapter 2

# Background

In this chapter, we introduce the topics that form the basis of this dissertation, along with the motivation for this work. First, we introduce different types of parallel programming models that are predominantly used in practice. Then, we describe the HabaneroJava (HJ) programming language and the parallel constructs in HJ which are the target of the algorithms in this dissertation. Then, we discuss data races, data race detection algorithms, and the soundness and completeness of these algorithms. Finally, we discuss about some related past work, along with their limitations, to motivate the need for better solutions.

### 2.1 Parallel Programming Models

With the advent of multicore processors, parallel programming has become an integral part of mainstream software development. Though automatic parallelization is appealing, it has not as yet shown signs of addressing a wide range of applications. Alternatively, many parallel programming models have been introduced. These models require that the programmers specify parallelism in their code explicitly.

In this section, we first describe three approaches to parallel programming and then discuss the difference between structured and unstructured parallel programming models.

### 2.1.1 Three Approaches to Parallel Programming

The parallel programming models can be broadly classified into three categories: the *directives* based approach, the *library* based approach, and the *language* based approach.

The *directives* based approach augments existing sequential programming language with directives (e.g., pragmas) and requires that programmers specify these directives in their programs which will act as hints/directions to the compiler for parallelizing the program. OpenMP [12] is an example of a directives based parallel programming model.

The *library* based approach provides a library that can be used by the programmers to specify parallelism in a sequential programming language. Pthreads and Java threads are examples of library based parallel programming models.

The *language* based models define a new programming language that include explicit parallel constructs. Generally, such parallel programming languages define parallel constructs on top of a sequential subset of some programming language. Programmers can use these parallel constructs to specify parallelism explicitly in their programs. Cilk [13], X10 [8], Habanero Java (HJ) [9], Unified Parallel C (UPC) [14], Coarray Fortran (CAF) [15] are examples of language based programming models. Since these programming models require that programmer explicitly specify the parallelism in their program, these are also referred to as explicitly parallel programming models.

The core task-parallel and loop-parallel constructs of these programming models can be viewed as semantic equivalents of *fork* and *join*, i.e., constructs to create a new task that can run in parallel with the parent and to wait for one or more tasks to complete. These programming models often include additional constructs



to augment fork-join parallelism with point-to-point synchronization, barriers, and mutual exclusion.

### 2.1.2 Structured and Unstructured Models

Parallel programming models can also be classified into *structured* and *unstructured* parallel programming models. Unstructured models do not impose any restriction on the structure of parallelism that their programs can generate. **Pthreads** and **Java threads** are examples of unstructured parallel programming models. On the other hand, structured models impose restrictions on the structure of parallelism that their programs can generate. Specifically, most of the structured models restrict the way tasks<sup>1</sup> can join with other tasks. For example, one restriction may be that a task can only wait for its descendants to complete. Such restrictions can help programmers easily reason about parallel programs. They also have benefits like deadlock freedom and enable simpler analysis of concurrency.

## 2.2 Habanero Java: A Structured Parallel Programming Model

In this section, we describe the **HabaneroJava** (HJ) programming language which has structured parallelism integrated in it. The race detection algorithms in this dissertation target some of the parallel constructs in the HJ language. Note that HJ started as an extension of **X10 v1.5** [8]. Since then, some new constructs that are not in X10, such as **phasers** [16], data-driven tasks, and hierarchical places, have been

---

<sup>1</sup>Here and elsewhere, we use *task/thread* to refer to application threads. We use *worker threads* explicitly to refer to the lower level system threads that the application executes on.

added to HJ. We now describe the parallel constructs in HJ which will be the target of our race detection algorithms.

### 2.2.1 Parallel Constructs

The core fork-join constructs in both HJ and X10 are `async` and `finish`. The `async` construct is used to create a new task and the `finish` construct is used to wait for a group of tasks to complete. These constructs are similar to *fork* and *join* with additional constraints on which tasks a task can join with, that come with semantic guarantees such as deadlock freedom. Our data race detection algorithms target these core constructs, as well as two other constructs; the `isolated` construct which is used for mutual exclusion, and the `future` construct which is used to create a task with a handle that can be used by other tasks to wait for this task specifically. Some higher-level constructs (e.g., `forall`) are translated to these constructs by the HJ compiler.

Now, we provide a brief description of the semantics of the parallel constructs that our race detection algorithms target. A complete description of all parallel constructs in HJ can be found in [9]. For formal operational semantics of `async` and `finish` constructs, please refer to [17].

When an HJ program begins execution, the *main task* starts executing the main method in the program.

- `async`: 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. In HJ, local variables are *private* to each task, whereas static and instance fields may be *shared* among tasks. An inner `async` is allowed to read a local variable declared in an outer scope. The value of the outer local variable is simply copied on entry to the `async`. However, an inner

`async` is not permitted to modify a local variable declared in an outer scope.

- **finish**: The statement `finish { s }` causes the parent task to execute `s` and then wait until all `async` tasks created within `s` have completed, including transitively spawned tasks.
- **isolated**: The statement `isolated { s }` guarantees that each instance of `s` will be performed in mutual exclusion with all other potentially parallel *interfering* instances of `isolated` statements.<sup>2</sup> Two instances of `isolated` statements are said to interfere with each other if both access the same shared location, such that at least one of the accesses is a write.
- **future**: The statement, `final future<T> f = async<T> Expr;` creates a new child task to evaluate `Expr` that is ready to execute immediately. In this case, `f` contains a **future handle** to the newly created task and the operation `f.get()` (also known as a **force operation**) 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. An important constraint in HJ is that all variables of type `future<T>` must be declared with a **final** modifier, thereby ensuring that the value of the reference cannot change after initialization. This rule ensures that no deadlock cycle can be created with **future** tasks.

Each dynamic instance  $T_A$  of an `async` task  $A$  has a unique *Immediately Enclosing Finish* (IEF) instance  $F$  of a `finish` statement during program execution, where  $F$  is

---

<sup>2</sup>As advocated in [18], HJ uses the `isolated` keyword instead of `atomic` to make explicit the fact that the construct supports *weak isolation* rather than *strong atomicity*.

the innermost dynamic `finish` scope containing  $A$ . There is an implicit `finish` scope surrounding the body of `main()` so program execution will terminate only after all `async` tasks in the program have completed.

We now define the Computation Graph (CG) of an HJ program execution.

**Definition 2.2.1.** A Computation Graph (CG)<sup>3</sup>,  $\Phi(N, E)$ , for a schedule  $\delta$  of an HJ program  $P$  is a directed acyclic graph (dag) where

1.  $N$  is the set of nodes such that each node  $n \in N$  corresponds to a statement instance in  $\delta$ .
2.  $E$  is the set of edges that connects the statement instances such that each edge  $e \in E$  belongs to one of the following types: *continue*, *async*, and *join* [7, 19]. There is a *continue* edge from every instance of a statement in a task to the instance of its next statement in the same task according to the program order. There is an *async* edge from every `async` statement instance to the instance of the first statement of the new task that it creates. There is a *join* edge from the instance of the last statement of every task to the statement instance that marks the end of its immediately enclosing `finish`. □

Figure 2.1 shows an example HJ program and the computation graph of the program for an execution where the `for` loop in line 5 executes only once. Each circle in the computation graph represents a statement instance in the program. The number in each circle indicates the program statement it represents. Each vertical sequence of circles denotes a task. The computation graph shows the four tasks,  $T_1$ ,  $T_2$ ,  $T_3$ ,

---

<sup>3</sup>A computation graph is sometimes also referred to as a dynamic computation graph because it corresponds to an execution of a program.

```

1  final int [] A, B;
2  ... ..
3  A[0] = 10;
4  finish {
5    for (int i=0; i<size; i++ ) {
6      final int ind = i;
7      async {
8        B[ind] += ind;
9        Foo q = new Foo();
10     for (int j=0; j<ind; j++) {
11       q.x += 1;
12       B[ind] = A[j] + ind;
13     } // for
14   } // async
15   finish {
16     async {
17       async {
18         B[ind] = A[ind];
19       } // async
20       B[ind+1] = A[ind+1] + 5;
21     } // async
22   } // finish
23 } // for
24 } // finish

```

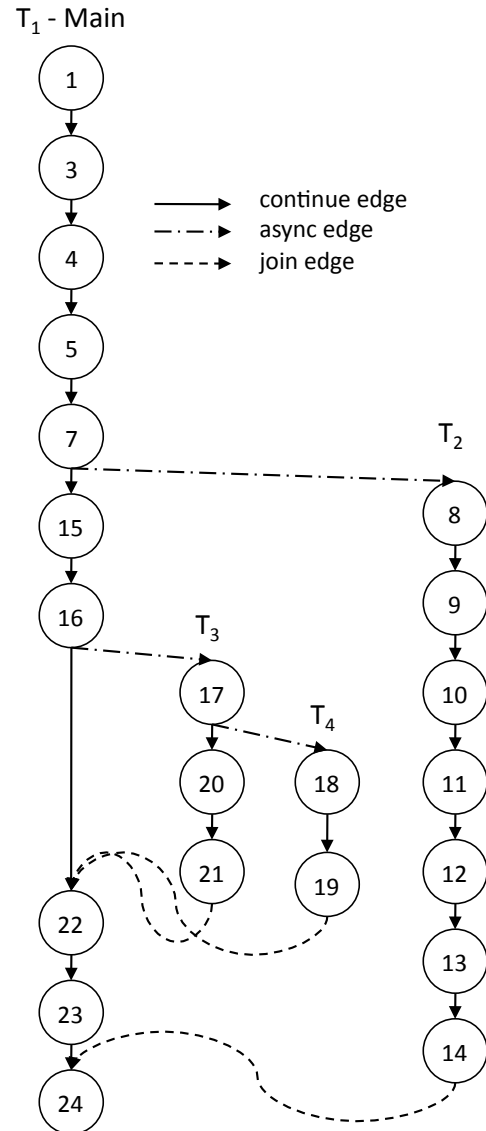


Figure 2.1 : An example HJ program and its computation graph. This code is the body of the main method in the program. The computation graph represents an execution where the for loop in line 5 executes only once.

and  $T_4$ , in the program. The solid edges are the *continue* edges, the alternate dashed and dotted edges are the *async* edges, and the dashed edges are the *join* edges.

### 2.2.2 Comparison with Cilk

Cilk is a programming language with integrated support for structured parallelism, like HJ and X10. The two core constructs in Cilk are `spawn` and `sync`. The `spawn` keyword can only be attached to a function call. It indicates that the function call can execute in parallel with the rest of the code in the caller. `Sync` is a statement that can be placed anywhere in a function. It indicates that the execution of the function cannot proceed until all the previously *spawned* functions complete.

The `async-finish` constructs of HJ supports a more relaxed concurrency model than the `spawn-sync` Cilk computations. The static lexical scope of `async-finish` subsumes all of `spawn-sync` excluding *conditional syncs*.<sup>4</sup> On the other hand, the dynamic computation graph of `async-finish` subsumes all of `spawn-sync` including *conditional syncs*. The key semantic relaxation lies in the way a task is allowed to join with other tasks. In Cilk, at any given (join) point of the task execution, a task must join with *all* of its immediate children tasks (and by transitivity, all its recursive descendant tasks) created in between the start of the task and the join point. The join is accomplished by executing the statement `sync`.

Most `spawn-sync` constructs can be translated to `async-finish` constructs as follows: each `spawn` construct can be directly replaced with an `async` construct. A Cilk function with unconditional `sync` statements can be directly translated to a sequence of `finish` blocks, where the start of the `finish` block is the start of the procedure or the previous

---

<sup>4</sup>We refer to a `sync` that is executed under some condition in a function body as a *conditional sync*.

`sync` and the end of the `finish` block is the label of the `sync` statement. It is not possible to directly translate the *conditional sync* to a `finish` because of the syntactic structure of `finish`.

To handle all programs that can be written with `spawn` and `sync`, we extend the HJ language with two keywords (or library calls), *beginFinish* and *endFinish*. The semantics of *beginFinish* is that it begins a `finish` block and the semantics of *endFinish* is that it completes a `finish` block. These dynamically specified *beginFinish* and *endFinish* scopes can be nested arbitrarily like the lexical `finish` construct. These constructs allow us to define the scope of the `finish` block *dynamically*. Note that while the programmer may use *beginFinish* and *endFinish* in an arbitrary order, the runtime system checks that they are properly nested: any *beginFinish* eventually completes with a matching *endFinish* (in the same task), and no *endFinish* is issued without a corresponding *beginFinish* already started (in the same task). The runtime system reports an error if they are not properly nested. As a high-level analogy, the relationship between *beginFinish* / *endFinish* and HJ's lexical `finish` construct is akin to that of *MonitorEnter* / *MonitorExit* bytecode instructions and Java's lexical *synchronized* statement (though bytecode verification rather than dynamic checking is used to check the proper nesting of *MonitorEnter* / *MonitorExit* instructions).

We can now translate all of the `sync` constructs of Cilk (including conditional syncs) into *beginFinish* and *endFinish* constructs as follows:

1. Generate a *beginFinish* on entry to every function.
2. Replace each occurrence of `sync` by *endFinish*; *beginFinish*.
3. Generate an *endFinish* on function exit to reflect Cilk's implicit `sync` on function exit.

This shows that the `async-finish` constructs subsume all of `spawn-sync` constructs. Our data race detection algorithms work by intercepting the start and end of `finish` and `async` constructs during the execution of a program. Hence, our algorithms can be applied directly to `spawn` and `sync` constructs of Cilk as well.

In contrast to Cilk, with the use of nested `finish` operations in HJ, it is possible for a task to join with *some* rather than all of its descendant tasks. This is not possible in Cilk because Cilk requires that every parallel function be *spawned*. The way these descendant tasks are specified at the language level in HJ is with the `finish` construct: upon encountering the end of a `finish` block, the task waits until all of the descendant tasks created inside the `finish` scope have completed.

The computation graph in Figure 2.1 illustrates the differences between Cilk and HJ. At node 22, the main task waits only for T3 and T4 and not for T2 in HJ, which is not possible using the `spawn-sync` semantics used in Cilk.

Further, another restriction in Cilk is that every task (spawned function) must execute a `sync` statement before its return. That is, a task cannot terminate unless all of its children (and transitively, descendants) have terminated. In contrast, in HJ, a task can outlive its parents, i.e., a task can complete even while its children are still alive. For instance, in the example of Figure 2.1, in Cilk, T3 would need to wait until T4 has terminated. That is, the edge from node 19 to 22 would change to an edge from 19 to 21. As we can see, this need not be the case in HJ: task T3 can terminate before task T4 has finished.

More generally, the class of computation graphs generated by the `spawn-sync` constructs is said to be *fully-strict* [20], while the computation graphs generated by `async-finish` constructs are called *terminally-strict* [21]. The set of terminally-strict computation graphs subsumes the set of fully-strict computation graphs. All these re-



laxations imply that it is not possible to rewrite all HJ programs using the `spawn-sync` constructs of Cilk, which in turn implies that we cannot use the data race detection algorithms designed for Cilk [5, 6, 2] to detect data races in all HJ programs.

## 2.3 Data Races

Data races are one of the most common form of bugs in explicitly parallel programs. The presence of data races may result in data corruption and may lead to non-deterministic behavior in parallel programs. Though data races may sometimes be benign, they are often indicative of errors that lead to unintended behaviors. Sometimes data races are intentional when implementing low-level synchronization operations. However, structured parallel programming models, which are the focus of this dissertation, target higher level of parallel programming where programmers are not expected to implement low-level synchronizations.

In this section, we define data races, the process of data race detection, the soundness and completeness (or precision) guarantees of data race detection algorithms, and also discuss data race detection in structured parallel programs.

**Definition 2.3.1.** Two accesses to a shared memory location by two different tasks result in a **data race** if:

- at least one access is a write, and
- there is no ordering between the two accesses. □

Note that this definition refers to *potential data races*, i.e., data races that occur in some schedule of the program for that input. The ordering between two memory accesses  $\alpha_1$  and  $\alpha_2$  that occur in tasks  $\tau_1$  and  $\tau_2$  could be one of the following:

- $\tau_2$  is guaranteed to begin only after  $\tau_1$  completes or vice-versa.
- There is some form of synchronization (e.g., post-wait, barrier, phaser) that guarantees that  $\alpha_2$  begins only after  $\alpha_1$  completes or vice-versa.
- There is some form of mutual exclusion which guarantees that  $\alpha_1$  and  $\alpha_2$  never execute in parallel.

In the absence of any such ordering between two conflicting memory accesses in concurrent tasks, an actual data race is guaranteed to occur in some schedule<sup>5</sup> of the program.

### 2.3.1 Data Race Detection

Data race detection is the process of identifying potential or actual data races in a program. It is important to identify data races correctly and efficiently because it helps programmers identify and fix bugs in their programs quickly. Thus, identifying data races could directly impact the productivity of programmers. The importance of data race detection is evident from the attention this problem has received in the literature for the past few decades.

Most data race detectors answer the following two questions for a given pair of memory accesses:

- Check if the two accesses are interfering, i.e., if the two accesses are to the same memory location and at least one of them is a write.
- Check if the two accesses may-happen-in-parallel.

---

<sup>5</sup>We use *schedule* and *execution* to refer to a particular schedule of a program.

Data race detectors maintain some data structures to answer these questions. Typically, they maintain some form of access history for every memory location to check if two accesses are interfering and some form of parallelism relationship among different threads in the program to check if two accesses belonging to two different threads may-happen-in-parallel.

Data race detectors can be classified into *static* and *dynamic* based on whether they perform *static* or *dynamic* analysis to detect races in a program. Also, there are some *hybrid* techniques that combine *static* and *dynamic* analysis to detect races in a given program.

### **Static Data Race Detection**

Static data race detection involves performing static analysis on the given program to identify data races. There has been a lot of work on static data race detection in the past [22, 23, 24, 25, 26, 27]. These techniques come with the usual advantages and disadvantages of static analysis. The advantage is that they can reason about the entire program and span all possible schedules of the program for all possible inputs. The downside is that the conservative nature of static analysis may result in false positives. Hence, static race detection may report false data races resulting in a negative impact on the productivity of programmers.

A variant of static race detection is to employ a race-free type system in the parallel programming language. These models augment the type system of a parallel programming language with synchronization relationships. All parallel programs that satisfy such type systems are guaranteed to be data race free [28, 29, 30, 31]. Hence, these models also statically prove data race freedom in parallel programs. But these models are very restrictive in the types of synchronizations that they allow and hence

affect programmability.

### Dynamic Data Race Detection

Dynamic data race detection involves performing dynamic analysis on the program to identify data races. Dynamic race detectors overcome some of the drawbacks of static race detectors. Since they perform the analysis at execution time they have the complete context of the execution through which they can eliminate false positives. But because they reason about data races by considering one particular schedule of a parallel program, their guarantees apply to the schedules which only contain the paths accessed in the examined schedule [4].

**Post-mortem vs On-the-fly analysis:** Some dynamic data race detectors perform *post-mortem* analysis on the execution traces of programs [32, 33, 34, 35]. In this model, a race detector monitors an execution of a program and logs all the necessary information during the execution. After the execution completes, it analyzes the logged information and detects data races in the program execution. The main drawback of this approach is that the execution logs can be prohibitively large even for small parallel programs that execute for a short duration.

The other class of dynamic data race detectors perform *on-the-fly* analysis to detect data races during an execution of a program [36, 37, 5, 6, 2, 38, 1]. These race detectors maintain information about the program and its memory accesses during the execution to identify conflicting accesses that result in a data race. In these models, since the race detectors can discard some information about the program and its memory accesses during execution (as and when they become unnecessary), they take much less space compared to the *post-mortem* analysis based approaches,

thereby making it possible to perform dynamic data race detection on longer program executions than *post-mortem* techniques.

The *on-the-fly* analysis based data race detectors can either be *summary* methods [4] which only report data races and not the conflicting accesses that cause those data races, or *access history* methods [37, 5, 6, 2, 1] which report data races along with the conflicting accesses that cause those data races. It is evident that the *access history* methods are more useful because they give enough information for the programmers to correct the data races unlike the *summary* methods.

### 2.3.2 Guarantees of Data Race Detection Algorithms

We now discuss about the *soundness* and *completeness* guarantees provided by race detection algorithms and also the levels at which the guarantees hold for different race detection algorithms.

**Definition 2.3.2.** A data race detection algorithm is **sound** if it does not report any **false negatives**; i.e., if there exists a data race, the algorithm will report it. In other words, if the algorithm does not report a data race, then there are no possible data races. □

**Definition 2.3.3.** A data race detection algorithm is **complete** if it does not report any **false positives**; i.e., if the algorithm reports a data race, then the data race really exists. □

The race detection algorithms may guarantee soundness and/or completeness at three different levels : *per-schedule*, *per-input*, or *per-program*. A race detection algorithm is said to be sound and (or) complete for a given schedule if it does not report any false positives and (or) false negatives for that particular schedule. Similarly, a

race detection algorithm is said to be sound and (or) complete for a given input, if it does not report any false positives and (or) false negatives for all possible schedules for a given input. Finally, a race detection algorithm is said to be sound and (or) complete for a given program, if it does not report any false positives and (or) false negatives for all possible schedules of the program for all possible inputs.

Note that *per-program* guarantee is stronger than *per-input* guarantee, which in turn is stronger than *per-schedule* guarantee. While static data race detection algorithms can provide *per-program* guarantee, the strongest guarantee that dynamic data race detection algorithms can provide is the *per-input* guarantee. In a dynamic data race detector with *per-input* guarantees, the data races reported may be for a schedule that is different from the examined schedule.

Since *completeness* is usually associated with static analysis, we instead use the term *precision* for dynamic analysis to avoid confusion. Hence, a dynamic race detection algorithm is **precise** if it does not report any **false positives**.

### 2.3.3 Data Race Detection for Structured Parallelism

Data race detectors that provide *per-schedule* guarantees have to report only those races that happen in that particular schedule. So, in this case, the race detectors have to look for conflicting accesses by different worker threads only. Specifically, they do not have to look for conflicting accesses by different tasks within the same worker threads. In such a situation, data race detectors are very similar for both unstructured and structured parallel programming models.

Data race detectors that provide *per-input* guarantees have to report those races that may happen in any schedule of the program for the given input. In this case, the race detectors have to look for conflicting accesses across different tasks in the

program which may or may not have been executed by the same worker thread in the examined schedule. In other words, they have to look for conflicting accesses in every pair of tasks that may execute in parallel with each other in some schedule of the program. Hence, *per-input* data race detectors for structured models have to be different from those for unstructured models.

Now that we have established the difference between race detection for structured and unstructured models, we focus on race detection for structured models a little more, since that is the focus of this dissertation. In structured parallel programs, data races manifest only in those schedules in which the tasks containing the conflicting accesses are executed by different worker threads. Since such programs typically contain a huge number of tasks, there are many possible schedules in which a particular data race is not apparent. Also, some schedulers may inadvertently execute the tasks containing conflicting accesses in the same worker thread, which in turn may hide some data races. In most cases, it is not useful to detect only those data races that occur in the schedule that is examined during data race detection.

In this dissertation, we focus on dynamic data race detectors that give *per-input* guarantees for structured parallel programming models. Hence, when our race detectors do not find any data races, it proves that there can be no data races in any schedule of the program for that input. In other words, our race detectors can prove *data-race freedom* in a program for a given input.

#### 2.3.4 Past Work on Data Race Detection

We now present some past work on data race detection that is very closely related to the problem we consider in this dissertation.

The Cilk paper [5] introduces the SP-bags algorithm for spawn-sync computations.

The SP-bags algorithm requires that the input parallel program is executed in a sequential depth first manner on a single worker thread. This algorithm requires only constant space per memory location and time per memory access that is proportional to the inverse Ackermann function. An extension to SP-bags was proposed by Cheng et al. [6] to handle locks in Cilk programs. Their approach includes a data race detection algorithm for programs that satisfy the *umbrella* locking discipline, i.e., each memory location is protected by the same lock in every parallel access. However, the slowdown factors reported in [6] were in the  $33\times$  -  $78\times$  range for programs that follow their locking discipline, and up to  $3700\times$  for programs that don't. The main drawback of these algorithms is that they require the input program to be executed sequentially, due to which they will not use the multiple cores available in today's processors. Also, since these algorithms apply only for `spawn-sync` programs, they cannot be directly applied to programs with `async-finish`.

Another related work on data race detection for structured parallel programs was also done as part of the Cilk project [2]. This work introduces an algorithm called SP-hybrid, which detects races in the program by executing it in parallel on multiple workers and with a constant space and time overhead. Their algorithm has constant overheads for both space and time. However, despite its good theoretical bounds, the SP-hybrid algorithm is very complex and incurs significant inefficiencies in practice. The original paper on SP-hybrid [2] provides no evaluation and subsequent evaluation of an incomplete implementation of SP-hybrid [39] was done only for a small number of processors. One indicator of the inefficiency of SP-hybrid is the fact that the CilkScreen race detector used in Intel Cilk++ [40] uses the sequential All-Sets algorithm [6] rather than the parallel SP-hybrid algorithm. Another drawback of their algorithm is that it is tightly coupled with Cilk's work-stealing scheduler.



Hence, their algorithm cannot be applied directly to other schedulers.

### **Objective**

We discussed couple of data race detectors from the past that are most related to our problem. This combined with the knowledge of other related work on data race detection, as discussed in Chapter 9, helps us define our objective very clearly.

Build a dynamic data race detector for terminally-strict (async-finish style) parallel programs that executes the input program in parallel, uses constant space per memory location, and is useful in practice.

## Chapter 3

### Sequential Data Race Detection for HJ

In this dissertation, we focus on dynamic data race detection for structured parallelism, specifically for parallel constructs in HJ. We begin by exploring a sequential dynamic data race detector for *async-finish* parallel programs. Among the various dynamic data race detection algorithms from the past, we start with the sequential SP-bags algorithm [5] because, the fully-strict computations that this algorithm targets are closely related to the terminally-strict computations that we focus on.

The SP-bags algorithm was designed to detect races in parallel programs with `spawn` and `sync` constructs, as part of the Cilk project. In this chapter, we describe techniques to generalize the SP-bags algorithm for the parallel constructs in HJ. First, we summarize the original SP-bags algorithm for `spawn` and `sync` constructs. Then, we describe a generalization of the SP-bags algorithm for programs with `async` and `finish` constructs of HJ. We also extend this algorithm for programs with `isolated` blocks. We refer to our race detection algorithm for the parallel constructs in HJ as the **ESP-bags** (Extended SP-bags) algorithm. Both the SP-bags and the ESP-bags algorithms execute the input program sequentially. Then, we explore further generalizations of the ESP-bags algorithm for *labeled-finish* constructs and a restricted form of `futures`, called *single-get futures*. We also show that the generalization of ESP-bags for *single-get futures* is a maximal generalization, in the sense that more general classes of parallel programs cannot be analyzed with the SP-bags approach.

### 3.1 Background: SP-bags for Fully-Strict Computations

This section summarizes the SP-bags algorithm that was developed for detecting data races in Cilk programs with `spawn` and `sync` constructs [5]. As discussed in Section 2.2.2, we can always translate `spawn-sync` computations into `async-finish` computations. Therefore, we present the operations of the original SP-bags algorithm in terms of `async` and `finish`, rather than `spawn` and `sync` constructs, so that the extensions that we describe later are easily understood.

The SP-bags algorithm is a sequential data race detection algorithm that performs a sequential depth-first execution of the parallel program.

Any dynamic data race detection algorithm has to identify parallel and interfering accesses to memory locations during program execution. The SP-bags algorithm uses two data structures, one to identify parallel accesses and the other to identify interfering accesses.

#### Identifying Parallel Accesses

First, we explain the data structure that the SP-bags algorithm uses to identify parallel accesses. Every dynamic task (`async`) instance is given a unique task id. Also, two “bags”, S and P, are attached to each dynamic task instance. Here, S stands for Serial and P for Parallel. Each bag contains a set of task id’s.

The following invariants are maintained by the algorithm throughout the execution of the program.

1. Every task id will always belong to at most one bag. Also, a task id must belong to at least one bag while the task is “live”.
2. When a statement E that belongs to a task A is being executed,

- (a) the S-bag of task A will hold all of the descendant tasks of A that always precede E in any execution of the program. The S-bag of A will also include A itself since any statement G in A that executes before E in the sequential depth first execution will always precede E in any execution of the program.
- (b) the P-bag of A holds all descendant tasks of A that may execute in parallel with E.

The implication of the invariant 1 is that the S and P bags of all the tasks are always disjoint during the execution of the program. Therefore, all these bags can be efficiently represented using a single disjoint-set data structure. The disjoint-set data structure maintains the entire collection of task id's with support for operations like *MakeSet*, *Union*, and *Find-Set*. Tarjan [41] proved that any  $m$  of these operations on  $n$  bags take a total of  $O(m \alpha(m, n))$  time, where  $\alpha(m, n)$  represents the inverse ackermann's function.

Invariant 2 forms the basis for identifying parallel accesses using the S and P bags. Table 3.1 shows the rules to update the S and P bags that are attached to each task. Note that the S and P bags need to be updated only at the start and end of tasks and at sync points (denoted by *EndFinish* in the table.).

When a task A is created, its S bag,  $S_A$ , is initialized to contain its own task id because no pair of accesses to a memory location in task A should conflict. The P bag of A is initialized to empty set because when A begins it has no descendants. When a task A returns to a task B during the depth-first execution, the contents of the S and P bags of A are moved to the P bag of B. This is because the code following task A in B can execute in parallel with A and hence, while executing this part of

Table 3.1 : SP-bags: Rules to update S and P bags at Task Boundaries

<b>Execution Point</b>	<b>Rules</b>
<i>Async A</i>	$S_A \leftarrow \{A\}$ $P_A \leftarrow \emptyset$
<i>Task A returns to Task B</i>	$P_B \leftarrow P_B \cup S_A \cup P_A$ $S_A \leftarrow \emptyset$ $P_A \leftarrow \emptyset$
<i>EndFinish F in a Task B</i>	$S_B \leftarrow S_B \cup P_B$ $P_B \leftarrow \emptyset$

the code in B, A and its descendants should be in a P bag. When a join point is encountered in a task A, the P bag of A is moved to its S bag. This is because the code after the join point in A can never execute in parallel with the descendants of A before the join and hence, while executing this part of the code in A, all descendants of A before the join should be in a S bag.

The intuition behind the algorithm is the following: when a program is executed sequentially in depth-first manner, a write  $W_1$  to a shared memory location  $L$  by a task  $\tau_1$  races with an earlier read/write to  $L$  by any task  $\tau_2$  which is in a P-bag when  $W_1$  occurs and it does not race with read/write to  $L$  by any task that is in an S-bag when  $W_1$  occurs. A read races with an earlier write in the same way. The SP-bags algorithm identifies parallel accesses using this idea.

```

1 Read location L by Task A:
2     If L.writer is in a P-bag then Data Race;
3     If L.reader is in a S-bag then L.reader = A;

1 Write location L by Task A:
2     If L.writer is in a P-bag or L.reader is in a P-bag
3         then Data Race;
4     L.writer = A;

```

Figure 3.1 : Action to be taken on read and write of shared memory locations. This action applies to the SP-bags and the ESP-bags algorithm.

### Identifying Interfering Accesses

We now discuss the data structure used in the SP-bags algorithm to identify interfering memory accesses in a program. Each memory location is augmented to contain two additional fields: a *reader* task id and a *writer* task id. These two fields keep track of the “most recent” tasks that accessed a given memory location.

In addition to updating the S and P bags at task boundaries, during the depth-first execution of a program, the SP-bags algorithm requires that action is taken on every read and write of a shared variable. Figure 3.1 shows the required actions for *read* and *write* operations.

On a read of a memory location L by a task A, the algorithm checks if the task stored in the *writer* field of L is in a P-bag, which implies that the task A may execute in parallel with *writer* task. Hence these two accesses conflict with each other and result in a data race. If not, then there are no conflicting accesses for this read by A. The *reader* field of L is updated to A only if the previous *reader* is in an S-bag. If the previous *reader* is in a P-bag, then it should not be updated to this task A

```

1  int main() {
2    // Task T1
3    // T1 will be in a S-bag
4    finish {
5      async { // T2
6        t = x; // x.reader = T2
7      } // T2 will be moved to a P-bag
8      y = x; // x.reader is in a P-bag => x.reader = T1 => Correct?
9      async {
10       x = 10; // x.reader is a S-bag => missed data race
11     }
12   }
13 }

```

Figure 3.2 : A program where a data race would be missed if the *reader* field of the memory location “x” is updated to the new reader while the previous one was in a P-bag.

because doing so may lead to false negatives in the algorithm. For example, during the execution of the program in Figure 3.2, the task T1 will be in its own S-bag. The task T2 will be moved to the P-bag of T1 after it completes execution in line 7. At this point the *reader* field of the memory location “x” will point to T2. Now, on the read of “x” in line 8 by T1, the algorithm does not update the *reader* field of “x” to T1 because T2 is in a P-bag. When the write to “x” happens in line 10, the algorithm will find the previous reader, T2, in a P-bag and hence, report a race between accesses in lines 6 and 10. Suppose the *reader* field of “x” was update to T1 instead. Then, on the write of “x” in line 10, the algorithm will miss the data race since the previous reader, T1, is in an S-bag.

On a write of a memory location  $L$  by a task  $A$ , the algorithm checks if the task stored in the *writer* field of  $L$  is in a P-bag and reports a data race. It repeats the same for the task stored in the *reader* field of  $L$ . Then, the *writer* field of  $L$  is updated to point to this task  $A$ .

Thus, for each operation on a shared memory location  $L$ , the algorithm checks those fields of  $L$  that interfere with the current operation and updates the field corresponding to the current operation.

### 3.2 ESP-bags: A Generalization of SP-bags for Terminally-Strict Computations

In this section, we describe a generalization of the SP-bags algorithm for terminally-strict computations. First, we present our generalization of that algorithm, ESP-bags, for detecting data races in programs with `async` and `finish` constructs. Then, we present an extension of the ESP-bags algorithm for programs with `isolated` blocks.

The basic requirement for SP-bags and ESP-bags algorithms is that a sequential depth-first execution of the input program is possible. This is trivially satisfied for all programs with `spawn` and `sync` constructs, as well as for all programs with `async` and `finish` constructs, because with these constructs every task can wait only for its descendants. This requirement is satisfied even for programs with `isolated` constructs because we require a sequential execution and also, no blocking operation is allowed within `isolated` blocks.

For programs with `future` constructs, this requirement is satisfied because of the restriction in HJ that every `future` must be declared `final`. Suppose this restriction was not present, then we could have programs for which a sequential depth-first



```

1  int main() {
2      future<int> f2 = null; // non-final variable
3      final future<int> f1 = async<int> {
4          while (f2 != null) ; // wait until f2 is set
5          int x = f2.get();
6          return x * 4;
7      };
8      f2 = async<int> {
9          return 10;
10     };
11 }

```

Figure 3.3 : A sample HJ program with `future` (without the `final` restriction) for which sequential depth-first execution is not possible.

execution may not be possible. The HJ program in Figure 3.3 is one example where a sequential depth-first execution is not possible because `future` “f1” waits for `future` “f2” to complete and “f2” will execute only after “f1” completes in the depth-first execution. However, a single-threaded execution (not depth-first) is still possible for the HJ program in Figure 3.3 where “f1” begins executing, suspends on line 4, then “f2” executes followed by the rest of “f1”.

### 3.2.1 ESP-bags for Async-Finish

Now, we present our extension to the SP-bags algorithm for the parallel constructs in the HJ language. We refer to this extended algorithm as the ESP-bags algorithm. We start with the ESP-bags algorithm for `async` and `finish` constructs in HJ.

Recall that the key difference between `async-finish` and `spawn-sync` lies in the flexi-

Table 3.2 : ESP-bags for Async-Finish: Rules to update S and P bags at Task Boundaries

Execution Point	Rules
<i>Async A</i> - fork a new task A	$S_A \leftarrow \{A\}$ $P_A \leftarrow \emptyset$
<i>Task A returns to Parent B</i>	$P_B \leftarrow P_B \cup S_A \cup P_A$ $S_A \leftarrow \emptyset$ $P_A \leftarrow \emptyset$
<b>StartFinish F</b>	$P_F \leftarrow \emptyset$
<b>EndFinish F in a Task B</b>	$S_B \leftarrow S_B \cup P_F$ $P_F \leftarrow \emptyset$

bility of selecting which of its descendants a given task can join with. Table 3.2 shows the updated rules at task boundaries for the ESP-bags algorithm. The modifications to SP-bags are highlighted in **bold**. The S and P bags need to be updated only at the start and end of *asyncs* and *finishes* in the program.

The key extension lies in attaching a P bag, not only to tasks, but also to identifiers of *finish* blocks. At the start of a *finish* block F, its P bag is initialized to empty set because it has no descendants yet. When a *finish* block F ends in a task B, the contents of the P bag of F are moved to the S bag of B. This is because at the end of the *finish* block F, all the tasks within the scope of F are guaranteed to complete. The code following the end of F in B can never execute in parallel with any task in F and hence, while executing this part of the code in B, all the descendants of F must be in a S bag. Further, during the depth-first execution, when a task A returns to its parent B, B may be either a task *or* a *finish* block. The actual operations on the S

and P bags in that case are identical to SP-bags.

The need for this extension comes from the fact that at the end of a `finish` block, only the tasks created inside the `finish` block are guaranteed to complete and will precede the tasks that follow the `finish` block. Therefore, only the tasks created inside the `finish` block need to be added to the S-bag of the parent task when the `finish` completes and those tasks created before the `finish` block began need to stay in the P-bag of the parent task.

Note that the rest of the SP-bags algorithm, i.e., the data structure used to identify the interfering memory accesses during program execution and the actions taken on read and write of every memory location, remains the same in the ESP-bags algorithm as well.

This extension generalizes the SP-bags algorithm from [5]. This means that the ESP-bags algorithm can be applied directly to `spawn-sync` programs as well, by first translating them to `async-finish` as shown earlier, and then applying the algorithm. Of course, if we know that the `finish` blocks have a particular structure, and we know that translated `spawn-sync` programs do, then we can safely optimize away the P bag for the `finish` id's and directly update the bag of the parent task (as was done in the original SP-bags algorithm).

## Discussion

In summary, the ESP-bags algorithm works by updating the *reader* and *writer* fields of a shared memory location whenever that memory location is read or written by a task. On each such read/write operation, the algorithm also checks to see if the previously recorded task in these fields (if any) can conflict with the current task, using the S and the P bags of the current task. We now show an example of how the

algorithm works for the code in Figure 2.1.

Figure 3.4 repeats the computation graph from Figure 2.1. The statements are numbered according to the order in which they will be executed in the sequential depth-first execution of ESP-bags. Figure 3.4 also shows the S and P bags of all the tasks and finishes when statement 20 executes. Statement 20 may execute in parallel with  $T_2$  and  $T_4$ . So, when statement 20 executes, both  $T_2$  and  $T_4$  are in a P-bag. But statement 20 can never execute in parallel with the statements already executed in  $T_1$  and  $T_3$ . So, when statement 20 executes,  $T_1$  and  $T_3$  are in an S-bag. Table 3.3 shows how the S and P bags of the tasks and finishes are modified by ESP-bags as the program corresponding to the computation graph is executed.

Each row in Table 3.3 shows the status of these S and P bags before or after the execution of a particular statement. The PC refers to the statement number that is executed. ‘-N’ in the PC column indicates that the bags are updated before the execution of statement N. ‘N-’ in the PC column indicates that the bags are updated after the execution of statement N. ‘N’ in the PC column indicates that reader/writer field of a memory location is updated before the corresponding memory operation in statement N. ‘\*N’ in the PC column indicates that a data race is signaled as a result of checking and updating the S and P bags due to a memory operation in N. The table also tracks the contents of the writer field of the memory location  $B[0]$  which is written in statements 8 and 18. The P bags of the tasks  $T_1$ ,  $T_2$ , and  $T_4$  are omitted here since they remain empty through this execution.

In the first three steps in the table, the S and P bags of  $T_1$ ,  $F_1$ , and  $T_2$  are initialized appropriately. When statement 8 is executed, the writer field of  $B[0]$  is set to the current task,  $T_2$ . Then, on completion of  $T_2$  in statement 14, the contents of its S and P bags are moved to the P bag of  $F_1$ . When the write to  $B[0]$  in statement

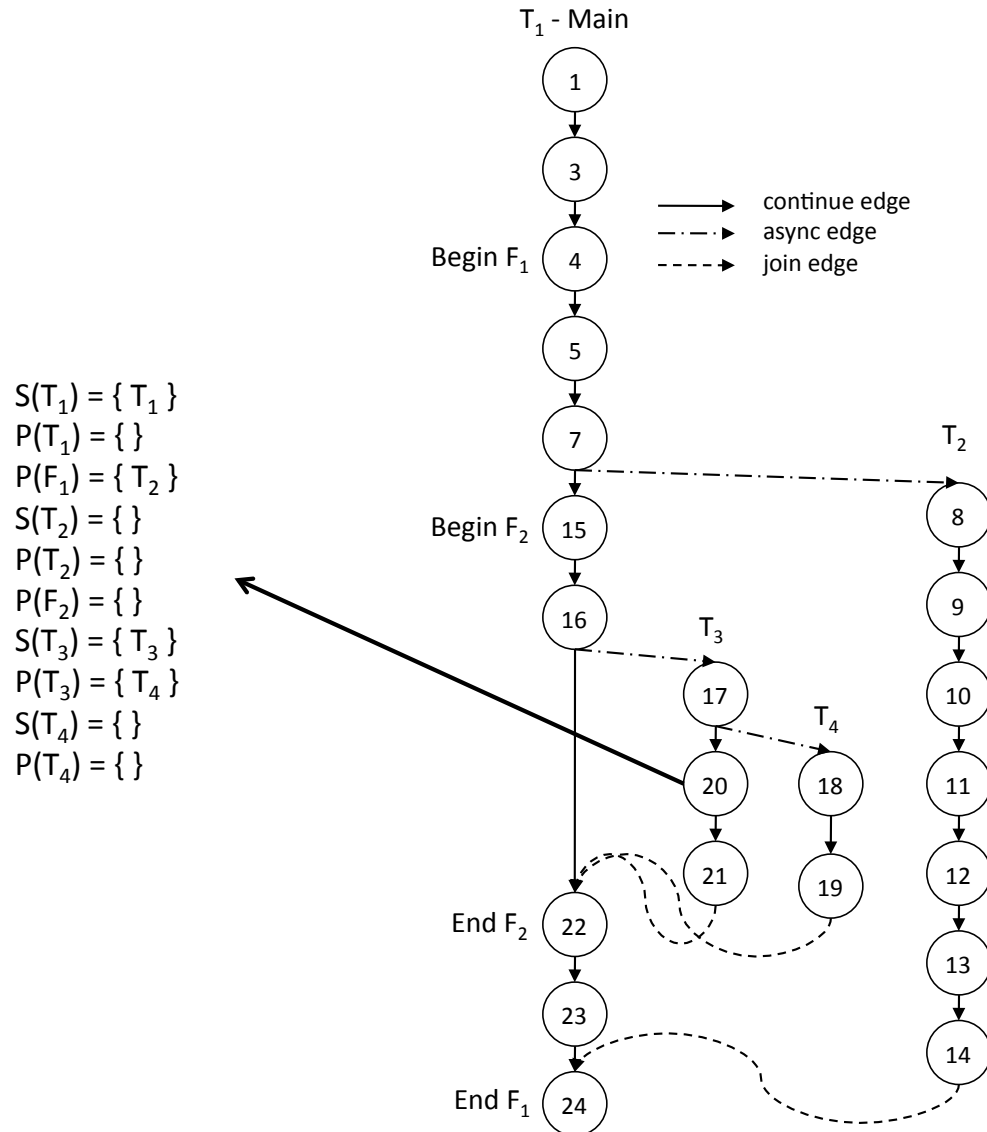


Figure 3.4 : The computation graph from Figure 2.1. The S and P bags of all the tasks and finishes when statement 20 executes are also shown here. Statement 20 may execute in parallel with  $T_2$  and  $T_4$ . So, when statement 20 executes, both  $T_2$  and  $T_4$  are in a P-bag. But statement 20 can never execute in parallel with the statements already executed in  $T_1$  and  $T_3$ . So, when statement 20 executes,  $T_1$  and  $T_3$  are in an S-bag.



18 (in Task  $T_4$ ) is executed, the algorithm finds the task in its writer field,  $T_2$ , in a P bag (P bag of  $F_1$ ). Hence this is reported as a data race. Further, when  $T_4$  completes in statement 19, the contents of its S and P bags are moved to the P bag of its parent  $T_3$ . Similarly, when  $T_3$  completes in statement 21, the contents of its S and P bags are moved to the P bag of its parent  $F_2$ . When the finish  $F_2$  completes in statement 22, the contents of its P bag are moved to the S bag of its parent  $T_1$ .

### Soundness and Precision

The ESP-bags (and SP-bags) algorithm detects a data race in a program,  $P$ , for a given input if and only if a data race exists for a given input. Hence, both the algorithms are *precise* and *sound* for a given input, as proved in Section 5.1. If the ESP-bags algorithm does not detect a data race in  $P$  for a given input, then it is guaranteed that there is no data race in any schedule of  $P$  for the given input. On the other hand, if a race is found, then it is guaranteed that there is some schedule of  $P$  with the given input for which the reported race is the first one encountered. There may be other schedules of  $P$  with the given input which may encounter different races in different orders, but all schedules are guaranteed to encounter a data race.

Note that the *precision* and *soundness* guarantees of the ESP-bags algorithm do not hold beyond the first data race. So, there may be false positives and false negatives among the data races that are reported after the first one if execution is continued after the first data race. This is true for all dynamic data race detectors because the code following the first race may be conditioned on the conflicting memory location and hence, that conditional code may not execute when the first race is removed. For example, the HJ program in Figure 3.5 has two data races, one on memory location “t” between accesses in statements 4 and 6, and the other on memory location “x”

```
1 int main() {
2   finish {
3     async {
4       t = 10;
5     }
6     t = 20;
7   }
8   if (t == 20) {
9     async {
10      x = 1;
11    }
12    x = 2; // a data race
13  }
14 }
```

Figure 3.5 : A sample HJ program with data races to show the lack of soundness and precision guarantees beyond the first data race.



between accesses in statements 10 and 12. Our algorithm will report both the races but the second race may not occur if the first race is fixed. If the condition in statement 8 was “`t == 10`” instead, then our algorithm will not report the second data race on the variable “`x`”, though this race may occur when the first race is fixed.

As an analogy, the lack of *precision* and *soundness* beyond the first data race in dynamic data race detectors is similar to the lack of *precision* and *soundness* beyond the first error in the list of syntax errors given by a compiler on compiling a given program.

### Space Overhead

The space overhead of the ESP-bags algorithm is  $O(1)$  for each memory location, since we just need to store the reader and writer task ids for each memory location. In addition to this, we need space to store all the task ids in the form of a disjoint set data structure. Note that we need to store the ids of completed tasks as well, since there might be a need to look up that task to check if it is in a S or a P bag as part of some memory access. The space overhead for storing all the task ids is  $O(a+f)$ , where ‘*a*’ is the number of async instances and ‘*f*’ is the number of finish instances in the program. But this space is generally insignificant compared to the space needed for each memory location.

### Time Overhead

In this algorithm, there are up to two look-ups for every memory access in the program. Note that some of these may be removed by the compiler optimizations described in Section 7. Also there are two union operations for each task instance in the program and one union operation for each finish instance. All these operations,

look-ups and unions, are performed on the disjoint-set data structure that contains all the tasks in the program. Tarjan et.al., showed that the worst case time taken for any operation on a disjoint-set structure is bounded by the inverse Ackermann function of the size of the data structure [41, 42]. Hence, each of these operations (look-up and union) will take time proportional to the inverse Ackermann function of the total number of tasks in the program. Note that the Ackermann function grows so fast that we can take the value of the inverse of Ackermann function to be bounded by a small constant (4) for all practical purposes. Since the number of memory accesses dominates the number of tasks in most programs, the worst-case time complexity of the algorithm is proportional to 4 times the number of memory accesses in the program.

### 3.2.2 Extending ESP-bags for Isolated Blocks

In this section, we describe an extension to the ESP-bags algorithm to handle **isolated** blocks in HJ programs [9, 43]. **Isolated** blocks are useful since they allow the programmer to write *data-race-free* parallel programs in which multiple tasks interact and update shared memory locations.

When a HJ program contains **isolated** blocks, the additional burden on the data race detector is to check for conflicts between isolated and non-isolated accesses to the same memory location that may execute in parallel and to ignore conflicts between two **isolated** accesses. If an access  $a_1$  to a memory location  $L$  in an **isolated** block conflicts with another access  $a_2$  to  $L$  in a non-isolated block, then it is a data race. Whereas, if an access  $a_1$  to a memory location  $L$  in an **isolated** block conflicts with another access  $a_2$  to  $L$  in another **isolated** block, then it does not constitute a data race. This is because, according to the semantics of **isolated**, every **isolated** block will

```

1 Isolated Read of location L by Task A:
2     If L.writer is in a P-bag then Data Race;
3     If L.isolatedReader is in a S-bag then L.isolatedReader = A;

1 Isolated Write of location L by Task A:
2     If L.writer is in a P-bag or L.reader is in a P-bag
3         then Data Race;
4     If L.isolatedWriter is in a S-bag then L.isolatedWriter = A;

1 Read location L by Task A:
2     If L.writer is in a P-bag or L.isolatedWriter is in a P-bag
3         then Data Race;
4     If L.reader is in a S-bag then L.reader = A;

1 Write location L by Task A:
2     If L.writer is in a P-bag or L.reader is in a P-bag
3         or L.isolatedWriter is in a P-bag
4         or L.isolatedReader is in a P-bag
5         then Data Race;
6     L.writer = A;

```

Figure 3.6 : Actions to be taken on isolated and non-isolated read, write operations on shared memory locations for the ESP-bags algorithm.

execute in mutual exclusion with respect to other *isolated* blocks. Note that though the conflicting operations in two *isolated* blocks may never execute in parallel, they may execute in one order in one schedule of the program and in some other order in another schedule of the program.

The extension to ESP-bags for handling *isolated* blocks includes checking that *isolated* and non-*isolated* accesses that may execute in parallel do not interfere. For this, we extend the ESP-bags algorithm as follows: two additional fields are added to every memory location, *isolatedReader*, and *isolatedWriter*. These fields are used to record the ids of the tasks that perform an *isolated* read or write on the memory location. The additional fields need only be added to memory locations that are accessed within *isolated* blocks.

In this extended ESP-bags algorithm, we need to handle reads and writes in *isolated* blocks differently as compared to *non-isolated* operations. Figure 3.6 shows the steps that need to be performed for each of the operations: *read*, *write*, *isolated-read*, and *isolated-write*.

An *isolated* read can never conflict with an *isolated* read, an *isolated* write, or a non-*isolated* read. It can only conflict with a non-*isolated* write. So, on an *isolated* read of a memory location L by task A, the algorithm checks if the previous writer is in a P-bag and reports a data race. The *isolatedReader* field of L is updated to A only if the previous *isolatedReader* is in an S-bag, for the same reasons as described earlier for *read*.

Similarly, an *isolated* write can never conflict with an *isolated* read or an *isolated* write. But it conflicts with a non-*isolated* read and write. So, on an *isolated* write of a memory location L by task A, the algorithm checks if the previous writer or reader is in a P-bag and reports a data race. Again the *isolatedWriter* field of L is updated

to  $A$  only if the previous *isolatedWriter* is in an S-bag.

The only addition on the actions taken on read and write of memory locations is that there are checks with additional fields for conflicts. On a read, the algorithm additionally checks if it conflicts with the previous isolated writer and on a write, the algorithm additionally checks if it conflicts with the previous isolated reader or writer.

### Soundness and Precision

With the extension to support `isolated`, the ESP-bags algorithm loses *soundness* (i.e., there may be false negatives) for a given input: there are example programs with `isolated` construct that contain races for a given input for which ESP-bags fails to find the race. Note that the ESP-bags algorithm is *precise* (i.e., there are no false positives) for a given input even in the presence of `isolated` blocks.

The problem is that with `isolated` blocks, there may be cases when the sequential depth-first execution does not execute certain paths of the code that may be executed in some parallel schedule for the same input. This happens when the `isolated` blocks in the program do not commute. In this case, for the same input, the `isolated` blocks may produce a different result in some parallel schedule as compared to the result produced in a depth-first execution and there may be some code conditioned on this result which has a data race. The ESP-bags algorithm does not report this data race because the code with the data race is never executed during the sequential depth-first execution of the algorithm.

Figure 3.7 shows an example HJ program that depicts a scenario in which the ESP-bags algorithm is not sound in the presence of `isolated` blocks. In this example, during the depth-first execution of our algorithm, the `isolated` block in line 3 executes

```

1  finish {
2    async {
3      isolated { t = 0; }
4    } // async
5    isolated { t = 1; }
6  } // finish
7  if (t == 0) {
8    async { x = 20; }
9    x = 10; // a data race
10 } // if

```

Figure 3.7 : An example HJ program that depicts a scenario in which ESP-bags is not *sound* in the presence of *isolated*.

before the *isolated* block in line 5. Hence, in such an execution, the *if* statement in line 7 evaluates to *false*, due to which the code in lines 8 and 9 do not execute and our algorithm reports no data races. However, there is a parallel schedule of this program for the same input in which the execution happens such that the *isolated* block in line 5 executes first, followed by the *isolated* block in line 3. In this schedule, the *if* in line 7 will evaluate to *true*, the code in lines 8 and 9 will execute and there will be a data race. This happens because the *isolated* blocks in lines 3 and 5 do not commute and hence they produce different results based on the order in which they are executed.

However, if the *isolated* blocks in the input program commute, the sequential depth-first execution is sufficient. In such cases, the ESP-bags algorithm does not miss data races for the given input. In practice, *isolated* blocks are used only with very small scopes and it is easy to show that they commute (for instance, they use only commutative operations like addition, to increment a counter).

In summary, when the `isolated` blocks commute, the ESP-bags algorithm is precise and sound for the given input, otherwise, our algorithm does not guarantee soundness.

### **Space Overhead**

The extension to ESP-bags to support the `isolated` construct involves two additional fields for every memory location that is monitored by the algorithm. But these two additional fields can be restricted to only those memory locations that are accessed from within the `isolated` blocks. The locations that are accessed from within `isolated` blocks can be identified dynamically as and when the accesses occur. They can be mapped to their additional fields using an explicit map. Typically, the `isolated` blocks in a program are very small and hence only a few memory locations are accessed within the `isolated` blocks. Hence, these additional fields do not add much to the space overhead of the algorithm (and hence, the size of the map is also typically small) and the asymptotic space overhead remains the same.

### **Time Overhead**

The time overhead of the extended ESP-bags algorithm increases with support for `isolated` because of the look-ups on the two additional fields in the memory location. Again, since these additional fields are restricted to only those few memory locations that are accessed from within the `isolated` blocks, the extra time spent looking up these additional fields will be small too. In the worst case, the time overhead may double for the extended ESP-bags algorithm, but there will be no change in the asymptotic overhead.

### 3.3 ESP-bags for Labeled-Finish

The SP-bags algorithm was designed to detect data races for fully-strict computations generated by `spawn-sync` constructs of Cilk. We generalized it to the ESP-bags algorithm to detect data races for terminally-strict computations generated by `async-finish` constructs of HJ. In this section, we describe a generalization of ESP-bags for the *labeled-finish* constructs.

The *labeled-finish* construct is a generalized form of the `finish` construct in HJ.<sup>1</sup> With *labeled-finish*, there is a *label* associated with every `finish` statement in the program. Now, every `async` in the program can choose to report to any of its ancestor finishes. An `async` reports to a particular ancestor `finish` by specifying the *label* associated with that `finish`. Note that this is a generalization of the original `finish` construct where every `async` reports to its immediately enclosing `finish` in the program. When we refer to an `async` A reporting to a `finish` F, we actually mean that the task performing F waits at the end of the `finish` F for A to complete before proceeding further.

Figure 3.8 shows a sample program to depict the usage of the *labeled-finish* construct. The `asyncs` A1 and A2 report to their immediately enclosing finishes, F1 and F2 respectively, according to the semantics of the original `finish` construct. But `async` A3 reports to `finish` F1 though its immediately enclosing `finish` is F2. This is because `async` A3 specifies a label on its creation, `label1`, which in turn specifies that A3 reports to the `finish` associated with that label, i.e., F1.

Note that, even with *labeled-finish*, every `async` has to report to one of its ancestor finishes. It is not possible for an `async` to report to some arbitrary `finish` which is not in its ancestor chain.

---

<sup>1</sup>We thank the X10 team at IBM Research for introducing us to the *labeled-finish* concept.



```
1 int main() {
2   label1: finish { // F1
3     async { // A1
4       finish { // F2
5         async { // A2
6           async (label1) { // A3
7             s4; s5;
8           } // async A3
9             s6;
10          } // async A2
11          s7;
12         } // finish F2
13         s8;
14        } // async A1
15        s9; s10; s11;
16      } // finish F1
17 }
```

Figure 3.8 : A sample program to depict the usage of the *labeled-finish* construct.

Table 3.4 : ESP-bags for Labeled-Finish: Rules to update S and P bags at Task Boundaries

<b>Execution Point</b>	<b>Rules</b>
<i>Async A (Label L)</i>	$S_A \leftarrow \{A\}$ $P_A \leftarrow \emptyset$
<b>Task A (with label L) returns to its Parent</b> <b>(F - finish associated with label L)</b>	$P_F \leftarrow P_F \cup S_A \cup P_A$ $S_A \leftarrow \emptyset$ $P_A \leftarrow \emptyset$
<i>StartFinish F</i>	$P_F \leftarrow \emptyset$
<i>EndFinish F in a Task B</i>	$S_B \leftarrow S_B \cup P_F$ $P_F \leftarrow \emptyset$

To detect data races using the ESP-bags algorithm on programs with labeled-finish constructs, we need to modify the rules to update the S and P bags at task boundaries. Table 3.4 shows the modified rules to update the S and P bags at task boundaries. The modifications to ESP-bags are highlighted in **bold**. Here again, the S and P bags need to be updated only at the start and end of **asyncs** and **finishes**.

The only change to the update rules for S and P bags from the ESP-bags algorithm for **async-finish** is in the rule when a task returns to its parent. The modified rules state that, when a task returns to its parent, the S and P bags of the task are emptied on to the P bag of the finish corresponding to the label associated with the task. This is different from the original algorithm where, when a task returns to its parent, the S and P bags of the task are emptied on to the P bag of the task's immediately enclosing **finish**. The rest of the ESP-bags algorithm remains the same for data race detection with labeled-finish constructs.

The soundness and precision guarantees of the ESP-bags algorithm (without `isolated`) remain the same even with the generalization to support labeled-finish constructs. In other words, even with the generalization to support labeled-finish constructs, the ESP-bags algorithm is precise and sound for a given input, i.e., it has no false positives or false negatives. This is because the labeled-finish constructs do not lead to non-deterministic executions, unlike `isolated`. Also, the only change in the ESP-bags algorithm is in the rules to update the S and P bags at task boundaries. This does not affect the soundness and precision guarantees of the algorithm.

The space overhead of the ESP-bags algorithm remains the same because the program still has the same number of task ids, one per task, and they are maintained using a disjoint-set data structure. Also, there is no change in the fields needed for every memory location. The time overhead of the algorithm also does not change because the algorithm performs exactly the same number of operations on the disjoint-set data structure at task boundaries.

### 3.4 ESP-bags for Single-Get Futures

In this section, we describe a generalization of the ESP-bags algorithm for a restricted subset of `futures`. A *single-get future* is a `future` on which exactly one instance of `get` operation is performed during the entire program execution. We show a generalization of the ESP-bags algorithm for programs with *single-get futures*. Also, we explain why the *single-get* restriction on `futures` is necessary for data race detection with the ESP-bags algorithm. We also show that when we remove the *single-get* restriction on `futures` it becomes impossible to generalize ESP-bags to detect data races.

Figure 3.9 shows a sample program that uses *single-get futures*. This program has two `futures`, `f1` and `f2`, and there is exactly one `get` operation on both these

```

1  int main() {
2      final future<void> f1 = async<void> {
3          s2; s3;
4      };
5      final future<void> f2 = async<void> {
6          s5;
7          f1.get();
8          s7;
9      }
10     s8;
11     f2.get();
12 }

```

Figure 3.9 : A sample program to depict the usage of *single-get* futures.

futures. Note that a *finish* can be modeled as a sequence of *single-gets* on all *asyncs* created in its scope. Also, *single-get futures* can be combined with *finish* and *async* in a program. For the sake of simplicity, we consider programs with only *single-get futures* in this section. Also, in this section, we assume that every *future* has exactly one *get* instance, though there can be *futures* with no *get* instances when they are combined with *finishes* in the program.

We need to modify the rules to update the S and P bags at task boundaries in the ESP-bags algorithm to detect data races in programs with *single-get futures*. The modified set of rules to update the S and P bags are given in Table 3.5. Here, the S and P bags need to be updated only at the start and end of tasks and on *get* operations.

When a *future* A is created, its S bag,  $S_A$ , is initialized to contain its own task

Table 3.5 : ESP-bags for Single-Get Future: Rules to update S and P bags at Task Boundaries

Execution Point	Rules
<i>Future A</i>	$S_A \leftarrow \{A\}$ $P_A \leftarrow \emptyset$
<b>Task A returns to its Parent</b>	$P_A \leftarrow P_A \cup S_A$ $S_A \leftarrow \emptyset$
<b>Task B executes A.get()</b>	$S_B \leftarrow S_B \cup P_A$ $P_A \leftarrow \emptyset$

id and its P bag,  $P_A$ , is initialized to empty set. When a **future** task A returns to its parent, the contents of the S bag of A are emptied on to its own P bag. This is because the **future** task A can execute in parallel with the code following it in its parent until a **get** operation is performed on A. Hence, during the execution of this part of the code in its parent, A and its descendants should be a in a P bag. Moving the contents of the S bag of A in to its own P bag is different from the original ESP-bags algorithm in which the contents of the S and P bags of A would be emptied on the P bag of the immediately enclosing **finish** of A. This is because, with **futures**, unlike **finishes**, every task can be waited on and joined with individually (using **get** operation on the **future**). When this **get** happens, the contents of the particular **future** and its descendants alone must be moved to an S-bag. Hence, they have to be kept separate from the contents of S and P bags of other tasks.

When task B performs a **get** operation on task A, the contents of the P bag of A are emptied on to the S bag of B. This is because the code in B after A.get() will never execute in parallel with A. Hence, during the execution of that part of B, A

and its descendants should be in an S-bag. Note that beyond the `A.get()` point, A and its descendants are merged with the contents of B. Hence, beyond this point, if B can execute in parallel with any part of the program, then A and its descendants also execute in parallel with those parts. This property holds because every **future** is guaranteed to have at most one **get** operation performed on it.

The ESP-bags algorithm is precise and sound for a given input even with the generalization to support *single-get futures*, i.e., it has no false positives or false negatives. This is because the use of **futures** in programs do not introduce non-deterministic behavior, unlike **isolated**. Also, the only change that is needed in the algorithm is in updating the S and P bags at task boundaries.

The space overhead of the algorithm remains the same because the algorithm still has the same number of task ids and they are represented using a disjoint-set data structure. Though the average number of bags that are alive at various points during the execution of the algorithm increases with this generalization, the set of bags on the whole store the same set of task ids. Also, there is no change needed in the fields associated with the memory locations. The time overhead of the algorithm at every task boundary remains the same because the algorithm still does the same number of operations on the disjoint-set data structure. But, with the presence of **get** operations on futures, there would be more task boundary points than in the case of **finishes**. Hence, the overall time overhead of the algorithm increases. But still, there would be no change in the asymptotic time overhead because there can be at most as many **get** operations as the number of **futures** in the program, due to the *single-get* restriction.

### 3.4.1 A Maximal Generalization of ESP-bags

Now, we describe why this generalization of ESP-bags for *single-get* futures is a maximal generalization of the ESP-bags algorithm.

When the *single-get* restriction on **futures** is relaxed, there can be more than one dynamic instance of a **get** operation on every **future** in the program. Now, whenever there is **get** operation on a **future** A performed by a task B, the code following A.get() in B can never execute in parallel with A. Hence, while executing this part of the code in B, A and its descendants must be in an S-bag. According to the rules defined above, when B performs A.get(), the contents of the P bag of A are emptied on to the S bag of B. At this point, the contents of A are merged on to the contents of B.

Now, suppose there is another task C that may happen in parallel with both A and B. When C performs A.get(), the code in C following A.get() can never execute in parallel with A, but may happen in parallel with B. Hence, while executing this part of the code in C, A and its descendants must be in an S-bag, but B and its descendants must still be in a P-bag. Since the contents of A and B are in one bag now, there is no way to split the contents of A from the bag to move them on to an S-bag. This happens because there is more than one **get** operation on the **future** A during the execution of a program.

The only other option is to not merge the P-bag of A on to the S-bag of B when B performs A.get(). In that case, the contents of A have to be kept separate. Similarly, the contents of every task has to be kept separate from other bags. Then, this degenerates to a case where the S and P bags of a task will only contain its own task id (at different points during execution) and no other task ids. Also, we will have to maintain a set of tasks that every task joins with, to move the contents from the S-bag to the P-bag of every task and vice-versa. Hence, in such a scenario, there

is no use of S and P bags for data race detection. Instead, we could associate a flag with every task to indicate whether the task is in *parallel* or *serial* with the current task and set the flag appropriately at various `get` points and task boundaries.

This clearly shows that, when the *single-get* restriction on `futures` is removed, the ESP-bags algorithm can no longer be used to detect data races. This is even true if the number of `get` instances on a `future` is any constant  $\geq 2$ . Hence, the computations generated by *single-get futures* are a maximal set of computations for which the ESP-bags algorithm can be generalized to detect data races. In other words, one maximal possible generalization of the ESP-bags algorithm is for programs with *single-get futures*. Note that there may be other maximal generalizations of ESP-bags for computation graphs generated by a different set of constructs.

### 3.5 Summary

The `async-finish` constructs of HJ subsume the `spawn-sync` constructs of Cilk. In other words, `async-finish` is more flexible than `spawn-sync`. Hence, any `spawn-sync` program can be converted to an `async-finish` program but not vice versa. Similarly, *single-get futures* subsume `async-finish` because all `async` and `finish` constructs can be replaced by *single-get futures* but not vice versa.

The SP-bags algorithm was designed for data race detection in Cilk programs with `spawn-sync` constructs. First, we presented an extension to the SP-bags algorithm, called ESP-bags, for detecting data races in HJ programs with `async-finish` constructs. The ESP-bags algorithm for `async-finish` is a generalization of the SP-bags algorithm for a more relaxed programming model without any additional cost in terms of space and time overhead. The ESP-bags algorithm for *single-get futures* further extends this algorithm to support an even more relaxed programming model without any



additional cost. Also, we showed that it is not possible to extend the ESP-bags algorithm for a more relaxed programming model than *single-get* futures.

This clearly shows the importance of the structure of parallelism in building race detectors with good space and time overheads.

## Chapter 4

### SPD3: A Parallel Algorithm for Detecting Data Races in HJ

Data race detectors play an important role in identifying bugs in parallel programs, thereby increasing the productivity of programmers. In Chapter 3, we presented a sequential algorithm called ESP-bags for detecting data races in HJ programs with some of its parallel constructs. The ESP-bags algorithm helps detect data races correctly and also eliminates most of the limitations of existing dynamic data race detectors. The only drawback is that it is a sequential algorithm which requires that the input program is executed in a sequential depth-first manner. Hence, the parallel hardware resources available in today's processors will not be used by this algorithm. In future, with hardware trending towards more parallel processors, this will become a serious bottleneck.

We solve this problem by introducing a new dynamic data race detection algorithm that has all the nice properties of the ESP-bags algorithm but can also execute the input program in parallel. This new parallel algorithm detects data races in HJ programs with some of its parallel constructs like `async`, `finish`, `isolated`, and `future`. We refer to this new parallel data race detection algorithm for HJ as SPD3. SPD3 stands for Scalable and Precise Dynamic Data race Detection.

In this chapter, we first present the SPD3 algorithm for `async` and `finish` constructs. We then extend SPD3 for `isolated` and `future` constructs in HJ programs.

## 4.1 SPD3 for Async-Finish

As we discussed earlier, most dynamic data race detection algorithms provide mechanisms that answer two questions: for any pair of memory accesses (with at least one write):

- determine whether the accesses can execute in parallel.
- determine whether they access the same location.

SPD3 maintains two different data structures to answer these two questions. First, we introduce the Dynamic Program Structure Tree (DPST), a data structure which can be used to answer the first question.

### 4.1.1 Dynamic Program Structure Tree

The DPST is an ordered rooted tree that is built at runtime to capture parent-child relationships among `async`, `finish`, and `step` (defined below) instances of a program. The internal nodes of a DPST represent `async` and `finish` instances. The leaf nodes of a DPST represent the steps of the program. Note that though we define the DPST in terms of `async` and `finish` constructs as in HJ in this dissertation, it can also be adapted to support dynamic analysis of structured parallel programs written in languages such as Cilk and OpenMP 3.0.

**Definition 4.1.1** (Step). A step is a maximal sequence of statement instances such that no statement instance in the sequence includes the start or end of an `async` or the start or end of a `finish` operation. □

**Definition 4.1.2** (DPST). The Dynamic Program Structure Tree (DPST) for a given execution is a tree in which all leaves are steps, and all interior nodes are `async` and `finish` instances. The parent relation is defined as follows:

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

There is a left-to-right ordering of all DPST siblings that reflects the left-to-right sequencing of computations belonging to their common parent task. Further, the tree has a single root that corresponds to the implicit top-level finish construct in the main program.  $\square$

### Building a DPST

Next we discuss how to build the DPST during program execution. When the main task begins, the DPST will contain a root finish node  $F$  and a step node  $S$  that is the child of  $F$ .  $F$  corresponds to the implicit finish enclosing the body of the main function in the program and  $S$  represents the starting computation in the main task.

**Task creation** When a task  $T$  performs an async operation and creates a new task  $T_{child}$ :

1. An async node  $A_{child}$  is created for task  $T_{child}$ . If the immediately enclosing finish (IEF)  $F$  of  $T_{child}$  exists within task  $T$ , then  $A_{child}$  is added as the rightmost child of  $F$ . Otherwise,  $A_{child}$  is added as the rightmost child node of (the async) node corresponding to task  $T$ .
2. A step node representing the starting computations in task  $T_{child}$  is added as the child of  $A_{child}$ .

3. A step node representing the computations that follow task  $T_{child}$  in task  $T$  is added as the right sibling of  $A_{child}$ .

Note that there is no explicit node in a DPST for the main task because everything done by the main task will be within the implicit finish in the main function of the program and hence all of the corresponding nodes in a DPST will be under the root finish node.

**Start Finish** When a task  $T$  starts a finish instance  $F$ :

1. A finish node  $F_n$  is created for  $F$ . If the immediately enclosing finish  $F'$  of  $F$  exists within task  $T$  (with corresponding finish node  $F'_n$  in the DPST), then  $F_n$  is added as the rightmost child of  $F'_n$ . Otherwise,  $F_n$  is added as the rightmost child of the (async) node corresponding to task  $T$ .
2. A step node representing the starting computations in  $F$  is added as the child of  $F_n$ .

**End Finish** When a task  $T$  ends a finish instance  $F$ , a step node representing the computations that follow  $F$  in task  $T$  is added as the right sibling of the node that represents  $F$  in the DPST.

Note that the DPST operations described thus far only take  $O(1)$  time. Thus, the DPST for a given program run grows monotonically as program execution progresses and new async, finish, and step instances are added to the DPST. Note that since all data accesses occur in steps, it follows that all tests for whether two accesses may happen in parallel will only take place between two leaves in a DPST.

```

1: finish { // F1
2:   s1;   } S1
3:   s2;   }
4:   async { // A1
5:     s3;   } S2
6:     s4;   }
7:     s5;   }
8:     async { // A2
9:       s6; } S3
10:    } // async A2
11:    s7;   } S4
12:    s8;   }
13:  } // async A1
14:  s9;   } S5
15:  s10;  }
16:  s11;  }
17:  async { // A3
18:    s12;  } S6
19:    s13;  }
20:  } // async A3
21: } // finish F1

```

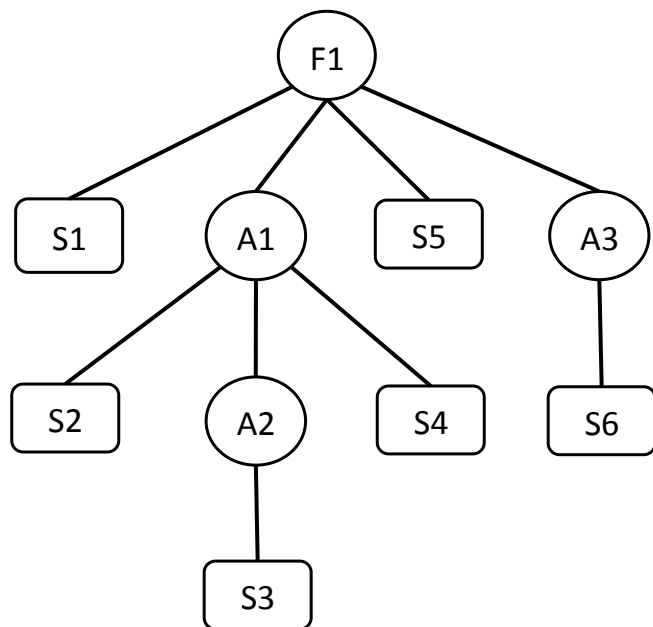


Figure 4.1 : A sample program in HJ with `async`, `finish`, statements. The statements `s1` - `s13` are grouped in to steps `S1` - `S6`. The tree on the right is the DPST corresponding to this program.

**Example** Let us now consider the example HJ program in Figure 4.1. This program consists of statements  $s1 - s13$ , asyncs  $A1 - A3$ , and a finish  $F1$ . Let us consider an execution in which the entire program is executed only once. So, every async, finish, and statement in the program will result in exactly only async instance, finish instance, and statement instance during this execution. Thus, we consider each of them as dynamic instances from now on. Note the way statement instances are grouped into steps  $S1 - S6$ .

When the main task starts executing finish  $F1$ , a node corresponding to  $F1$  is added as the root node of the DPST, and a step node  $S1$  is added as the child of  $F1$ ;  $S1$  represents the starting computations in  $F1$ , i.e., instances of statements  $s1$  and  $s2$ . When the main task forks the task  $A1$ , an async node corresponding to  $A1$  is added as the right-most child of  $F1$  (since the immediately enclosing finish of  $A1$  is  $F1$  and it is within the main task), a step node  $S2$  is added as the child of  $A1$ , and a step node  $S5$  is added as the right sibling of  $A1$ .  $S2$  represents the starting computations in  $A1$  (i.e., instance of statements  $s3, s4$ , and  $s5$ ) and  $S5$  represents the computation that follows  $A1$  in the main task (i.e., instances of statements  $s9, s10$ , and  $s11$ ). After this point, the main task and the task  $A1$  can execute in parallel. Eventually, the DPST grows to the form shown in the figure.

### Properties of a DPST

In this section, we briefly summarize some key properties of a DPST.

- For a given input that leads to a data-race-free execution of a given async-finish parallel program, all executions of that program with the same input will result in the same DPST.

- Let  $F$  be the DPST root (finish) node. Each non-root node  $n_0$  is uniquely identified by a finite path from  $n_0$  to  $F$ :

$$n_0 \xrightarrow{r_0} n_1 \xrightarrow{r_1} n_2 \xrightarrow{r_2} \dots \xrightarrow{r_{k-1}} n_k$$

where  $k \geq 1$ ,  $n_k = F$ , and for each  $0 \leq i < k$ ,  $n_i$  is the  $r_i^{\text{th}}$  child of node  $n_{i+1}$ . The path from  $n_0$  to  $F$  stays invariant as the tree grows. For a given statement instance, its path to the root is unique regardless of which execution is explored (as long as the executions start with the same state). This property holds up to the point that a data race (if any) is detected.

- The DPST is amenable to efficient implementations in which nodes can be added to the DPST in parallel without any synchronization in  $O(1)$  time. Also, the DPST is amenable to garbage collection, i.e., the nodes in the DPST can be garbage collected as and when they become obsolete.

**Definition 4.1.3.** A node  $A$  is said to be to the left of a node  $B$  in a DPST if  $A$  appears before  $B$  in the depth first traversal of the tree.  $\square$

The function  $Left(A, B)$ , where  $A \neq B$ , evaluates to  $A$  if  $A$  is to the left of  $B$  in a DPST and to  $B$  otherwise. Similarly, the function  $Right(A, B)$ , where  $A \neq B$ , evaluates to  $A$  if  $B$  is to the left of  $A$  in a DPST and to  $B$  otherwise.

As mentioned above, even though the DPST changes during program execution, the path from a node to the root does not change and the left-to-right ordering of siblings does not change. Hence, even though the depth first traversal of the DPST is not fully specified during program execution, the *left* relation between any two nodes in the current DPST is well-defined.



**Definition 4.1.4.** The Lowest Common Ancestor (LCA)<sup>1</sup> of two nodes  $S_1$  and  $S_2$  in a DPST, denoted by  $LCA(S_1, S_2)$ , is the node  $\lambda$  that is an ancestor<sup>2</sup> of both  $S_1$  and  $S_2$  with the greatest depth.<sup>3</sup>  $\square$

**Definition 4.1.5.** Two steps,  $S_1$  and  $S_2$ , in a DPST  $\Gamma$  that corresponds to a program  $P$  with input  $\psi$ , may execute in parallel if and only if there exists at least one schedule  $\delta$  of  $P$  with input  $\psi$  in which  $S_1$  executes in parallel with  $S_2$ .  $\square$

The predicate  $DMHP(S_1, S_2)$  evaluates to *true* if steps  $S_1$  and  $S_2$  can execute in parallel in at least one schedule of a program and to *false* otherwise ( $DMHP$  stands for “Dynamic May Happen in Parallel” to distinguish it from the MHP relation used by static analysis). Note that the relation  $DMHP$  is *symmetric*, i.e., for every  $S_1$  and  $S_2$ ,  $DMHP(S_1, S_2) = DMHP(S_2, S_1)$ . We now state a key theorem that will be important in enabling our approach to data race detection.

**Theorem 4.1.1.** *Consider two leaf nodes (steps)  $S_1$  and  $S_2$  in a DPST, where  $S_1 \neq S_2$  and  $S_1$  is to the left of  $S_2$  as shown in Figure 4.2. Let  $LCA$  be the node denoting the lowest common ancestor of  $S_1$  and  $S_2$  in the DPST. Let node  $A$  be the ancestor of  $S_1$  that is a child of  $LCA$ . Then,  $DMHP(S_1, S_2) = true$  if and only if  $A$  is an async node.*

*Proof. if:  $A$  is an async node.* Let us consider a schedule  $\delta$  of  $P$  with input  $\psi$  such that one worker executes the subtree under  $A$  and the other worker executes all the subtrees under  $LCA$  that are to the right of  $A$ . This is possible because, according to the semantics of an async,  $A$  is not guaranteed to complete before any of its peers on

---

<sup>1</sup>LCA is sometimes referred to as Least Common Ancestor

<sup>2</sup>In a DPST, a node is considered both an ancestor and a descendant of itself.

<sup>3</sup>The depth of a node in a DPST is the length of the path from the root to the node.

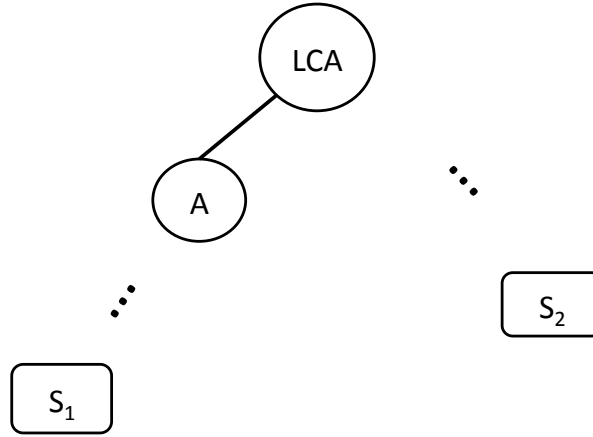


Figure 4.2 : A part of a DPST. LCA is the Lowest Common Ancestor of steps  $S_1$  and  $S_2$ .  $A$  is the DPST ancestor of  $S_1$  which is the child of LCA.  $S_1$  and  $S_2$  can execute in parallel if and only if  $A$  is an async node.

the right.  $A$  is guaranteed to complete only by the end of its immediately enclosing finish  $F$ . Note that  $F$  may be the node LCA or any of its ancestors. Now, in this schedule,  $\delta$ , the subtree under  $A$  will execute in parallel with the subtrees to the right of  $A$  under LCA. Hence,  $S_1$  will execute in parallel with  $S_2$  in  $\delta$ . Thus,  $DMHP(S_1, S_2) = true$ .

*only if:*  $DMHP(S_1, S_2) = true$ . In general, node  $A$  can be an async node, finish node or the same node as  $S_1$ . Let  $A_2$  be the ancestor of  $S_2$  which is the child of LCA.

Case 1: Assume  $A$  is a finish node.  $A_2$  must be disjoint from, and to the right of  $A$ . According to the semantics of finish, the subtree under  $A$  must complete before any peers to the right of  $A$  (including  $A_2$ ) start execution. Hence, it is guaranteed in all possible schedules of  $P$  with input  $\psi$  that  $S_1$  will complete execution before  $S_2$  can start executing. This contradicts our assumption that  $DMHP(S_1, S_2) = true$ .

Case 2: Assume  $A = S_1$ . Due to the left-to-right sequencing of computations within a task, step  $S_1$  must complete before any peers to the right of  $S_1$  (including  $A_2$ ) start execution. Hence, it is guaranteed in all possible schedules of  $P$  with input  $\psi$  that  $S_1$  will complete execution before  $S_2$  can start executing. This contradicts our assumption that  $DMHP(S_1, S_2) = true$ .

Thus  $A$  has to be an async node. □

### Computing DMHP

Now, we describe the algorithms to compute the DMHP relation for any given pair of steps. We assume that every node in the DPST has a *depth* field which specifies the depth of the node in the DPST and a *parent* field which gives the parent of the node in the DPST.

Algorithm 1 computes the lowest common ancestor of two steps  $S_1$  and  $S_2$ . If step  $S_1$  is at a greater depth than step  $S_2$ , then the loop in line 5 traverses the chain of ancestors from  $S_1$  until it reaches a node at the same depth as  $S_2$ . The loop in line 9 does the same for  $S_2$  if its depth is greater than the depth of  $S_1$ . At the start of loop in line 13, both nodes will be at the same depth. The algorithm then traverses up the ancestor chains of both the nodes until they reach a common node  $L$ . This common node  $L$  is the lowest common ancestor of the steps  $S_1$  and  $S_2$ . The time overhead of this algorithm is linear in the length of the longer of the two paths,  $S_1 \rightarrow L$  and  $S_2 \rightarrow L$ .

Algorithm 2 computes the DMHP relation for two steps,  $S_1$  and  $S_2$ . It returns *true* if the given two steps  $S_1$  and  $S_2$  may happen in parallel and *false* otherwise. This algorithm first computes the lowest common ancestor  $L$  of the given two steps using Algorithm 1. If the step  $S_1$  is to the left of  $S_2$ , then the algorithm returns *true* if the

---

**Algorithm 1:** Lowest Common Ancestor (LCA)
 

---

**Input:** DPST  $\Gamma$ , Step  $S_1$ , Step  $S_2$

**Output:**  $N_{lca}$ : the node which is the lowest common ancestor of  $S_1$  and  $S_2$  in  $\Gamma$

```

1  $node_1 \leftarrow S_1$ 
2  $node_2 \leftarrow S_2$ 
3  $depth_1 \leftarrow node_1.depth$ 
4  $depth_2 \leftarrow node_2.depth$ 
5 while  $depth_1 > depth_2$  do
6    $node_1 \leftarrow node_1.parent$ 
7    $depth_1 \leftarrow node_1.depth$ 
8 end
9 while  $depth_2 > depth_1$  do
10   $node_2 \leftarrow node_2.parent$ 
11   $depth_2 \leftarrow node_2.depth$ 
12 end
13 while  $node_1 \neq node_2$  do
14   $node_1 \leftarrow node_1.parent$ 
15   $node_2 \leftarrow node_2.parent$ 
16 end
17  $N_{lca} \leftarrow node_1$ 
18 return  $N_{lca}$ 

```

---

---

**Algorithm 2:** Dynamic May Happen in Parallel (DMHP)
 

---

**Input:** DPST  $\Gamma$ , Step  $S_1$ , Step  $S_2$

**Output:** true/false

```

1  $N_{lca} \leftarrow \text{LCA}(\Gamma, S_1, S_2)$ 
2  $A_1 \leftarrow$  Ancestor of  $S_1$  in  $\Gamma$  which is the child of  $N_{lca}$ 
3  $A_2 \leftarrow$  Ancestor of  $S_2$  in  $\Gamma$  which is the child of  $N_{lca}$ 
4 if  $\text{Left}(A_1, A_2) = A_1$  then
5   | if  $A_1$  is an Async then
6   |   | return true
7   | else
8   |   | return false ; //  $S_1$  happens before  $S_2$ 
9   | end
10 else
11   | if  $A_2$  is an Async then
12   |   | return true
13   | else
14   |   | return false ; //  $S_2$  happens before  $S_1$ 
15   | end
16 end

```

---

ancestor of  $S_1$  (which is the child of  $L$ ) is an async and *false* otherwise. If the step  $S_2$  is to the left of  $S_1$ , then the algorithm returns *true* if the ancestor of  $S_2$  which is the child of  $L$  is an async and *false* otherwise. The time overhead of this algorithm is same as that of the *LCA* function, since it only takes constant time to find the node which is the ancestor of the left step that is the child of LCA node and then check if that node is an async.

**Example** Let us now look at the *DMHP* relation for some pairs of steps in the example program in Figure 4.1. First, let us consider  $DMHP(S2, S5)$ . Here  $S2$  is to the left of  $S5$ , since  $S2$  will appear before  $S5$  in the depth first traversal of the DPST. The lowest common ancestor of  $S2$  and  $S5$  is the node F1. The node A1 is the ancestor of  $S2$  (the left node) that is the child of F1. Since A1 is an async node,  $DMHP(S2, S5)$  will evaluate to *true* indicating that  $S2$  and  $S5$  can execute in parallel. This is indeed *true* for this program:  $S2$  is within A1, while  $S5$  follows A1 and is within A1's immediately enclosing finish.

Now, let us consider  $DMHP(S6, S5)$ . Here  $S5$  is to the left of  $S6$ , since  $S5$  will appear before  $S6$  in the depth first traversal of the DPST. Their lowest common ancestor is F1, and the ancestor of  $S5$  which is the child of F1 is  $S5$  itself. Since  $S5$  is not an async instance,  $DMHP(S6, S5)$  evaluates to *false*. This is consistent with the program because  $S6$  is in task A3 and A3 is created only after  $S5$  completes.

#### 4.1.2 Shadow Memory

A key novelty of our algorithm is that it requires constant space to store the access history of a memory location, while still guaranteeing that no data races are missed. We now describe the shadow memory mechanism that supports this constant space

guarantee.

Our algorithm maintains a shadow memory  $M_s$  for every monitored memory location  $M$ .  $M_s$  is designed to store the relevant parts of the access history to  $M$ . Our algorithm uses this shadow memory to determine whether two accesses in a program are to the same memory location. The shadow memory,  $M_s$ , contains the following three fields, which are all initialized to null:

- $w$  : a reference to a step that wrote  $M$ .
- $r_1$  : a reference to a step that read  $M$ .
- $r_2$  : a reference to another step that read  $M$ .

The following invariants are maintained on the shadow memory throughout the execution of the program until the first data race is detected.

- $M_s.w$  refers to the step that last wrote  $M$ . This is the only step that has accessed  $M$  since the last synchronization (end finish).
- $M_s.r_1$  &  $M_s.r_2$  refer to the steps that last read  $M$ .  $M_s.r_1$  and  $M_s.r_2$  may happen in parallel, i.e.,  $DMHP(M_s.r_1, M_s.r_2) = true$ . There have only been reads to  $M$  in parallel since the last synchronization (end finish). All the steps  $(a_1, a_2, \dots, a_k)$  that have read  $M$  since the last synchronization are in the subtree rooted at  $LCA(M_s.r_1, M_s.r_2)$ .

One of the important aspects of our algorithm is that it stores only three fields for every monitored memory location irrespective of the number of steps that access that memory location. This is similar to the constant size access history used in the Offset-Span labeling algorithm [37].

The intuition behind this is as follows: it is only necessary to store the last write to a memory location because all the writes before the last one must have completed at the end of the last synchronization. This is assuming no data races have been observed yet during the execution. Note that though synchronization due to finish may not be global, two writes to a memory location have to be ordered by some synchronization to avoid constituting a data race. Among the reads to a memory location,  $(a_1, a_2, \dots, a_k)$ , since the last synchronization, it is only necessary to store two reads,  $a_i, a_j$ , such that the subtree under  $LCA(a_i, a_j)$  includes all the reads  $(a_1, a_2, \dots, a_k)$ . This is because every future read,  $a_n$ , which is in parallel with any discarded step will also be in parallel with at least one of  $a_i$  or  $a_j$ . Thus, the algorithm will not miss any data race by discarding these steps.

The fields of the shadow memory  $M_s$  are updated *atomically* by different tasks that access  $M$ .

### 4.1.3 SPD3 Algorithm

Our race detection algorithm involves executing the given program with a given input and monitoring every dynamic memory access in the program for potential data races. The algorithm maintains a DPST and the shadow memory for each shared memory location as described earlier. The algorithm performs two types of actions:

- Task actions: these involve updating the DPST with a new node for each *async*, *finish*, and *step* instance.
- Memory actions: on every shared memory access, the algorithm checks if the access conflicts with the access history for the relevant memory location. If a conflict is detected, the algorithm reports a race. Then, the memory location



is updated to include the memory access in its access history.

**Definition 4.1.6.** In a DPST, a node  $n_1$  is dpst-greater than a node  $n_2$ , denoted by  $n_1 >_{dpst} n_2$ , if  $n_1$  is an ancestor of  $n_2$  in the DPST. Note that, in this case,  $n_1$  is higher in the DPST (closer to the root) than  $n_2$ .  $\square$

---

**Algorithm 3:** Write Check

---

**Input:** Memory location  $M$ , Step  $S$  that writes to  $M$

```

1 if  $DMHP(M_s.r_1, S)$  then
2   | Report a read-write race between  $M_s.r_1$  and  $S$ 
3 end
4 if  $DMHP(M_s.r_2, S)$  then
5   | Report a read-write race between  $M_s.r_2$  and  $S$ 
6 end
7 if  $DMHP(M_s.w, S)$  then
8   | Report a write-write race between  $M_s.w$  and  $S$ 
9 else
10  |  $M_s.w \leftarrow S$ 
11 end

```

---

Algorithms 3 and 4 show the checking that needs to be performed on write and read accesses to monitored memory locations. When a step  $S$  writes to a memory location  $M$ , Algorithm 3 checks if  $S$  may execute in parallel with the reader in  $M_s.r_1$  by computing  $DMHP(S, M_s.r_1)$ . If they can execute in parallel, the algorithm reports a read-write data race between  $M_s.r_1$  and  $S$ . Similarly, the algorithm reports a read-write data race between  $M_s.r_2$  and  $S$  if these two steps can execute in parallel. Then,

Algorithm 3 reports a write-write data race between  $M_s.w$  and  $S$ , if these two steps can execute in parallel. Finally, it updates the writer field,  $M_s.w$ , with the current step  $S$  indicating the latest write to  $M$ . Note that this happens only when the write to  $M$  by  $S$  does not result in data race with any previous access to  $M$ .

When a step  $S$  reads a memory location  $M$ , Algorithm 4 reports a write-read data race between  $M_s.w$  and  $S$  if these two steps can execute in parallel. Then, it updates the reader fields of  $M_s$  as follows: if  $S$  is the first step that reads  $M$ , then  $M_s.r_1$  is set to  $S$ . If  $M_s.r_2$  is null, then  $M_s.r_2$  or  $M_s.r_1$  is set to  $S$  depending on whether  $S$  may execute in parallel with  $M_s.r_1$  or not. If  $S$  can never execute in parallel with either of the two readers,  $M_s.r_1$  and  $M_s.r_2$ , then both these readers are discarded and  $M_s.r_1$  is set to  $S$ . If  $S$  can execute in parallel with both the readers,  $M_s.r_1$  and  $M_s.r_2$ , then the algorithm stores two of these three steps, whose  $LCA$  is the highest in the DPST, i.e., if  $LCA(M_s.r_1, S)$  or  $LCA(M_s.r_2, S)$  is dpst-greater than  $LCA(M_s.r_1, M_s.r_2)$ , then  $M_s.r_1$  is set to  $S$ . Note that in this case  $S$  is outside the subtree under  $LCA(M_s.r_1, M_s.r_2)$  and hence,  $LCA(M_s.r_1, S)$  will be the same as  $LCA(M_s.r_2, S)$ .

If  $S$  can execute in parallel with one of the two readers and not the other, then the algorithm does not update the readers because, in that case,  $S$  is guaranteed to be within the subtree under the  $LCA(M_s.r_1, M_s.r_2)$ .

### Atomicity Requirements

A memory action for an access to a memory location  $M$  involves reading the fields of  $M_s$ , checking the predicates, and possibly updating the fields of  $M_s$ . Every such memory action has to execute atomically with respect to other memory actions for accesses to the same memory location.

---

**Algorithm 4:** Read Check
 

---

**Input:** Memory location  $M$ , Step  $S$  that reads  $M$

```

1 if  $DMHP(M_s.w, S)$  then
2   | Report a write-read data race between  $M_s.w$  and  $S$ 
3 end

4 if  $M_s.r_1 = null$  then
5   |  $M_s.r_1 \leftarrow S$ 
6 else if  $M_s.r_2 = null$  then
7   | if  $DMHP(M_s.r_1, S)$  then
8     |  $M_s.r_2 \leftarrow S$ 
9   | else
10    |  $M_s.r_1 \leftarrow S$ 
11   | end
12 else if  $\neg DMHP(M_s.r_1, S) \wedge \neg DMHP(M_s.r_2, S)$  then
13   |  $M_s.r_1 \leftarrow S$ 
14   |  $M_s.r_2 \leftarrow null$ 
15 else if  $DMHP(M_s.r_1, S) \wedge DMHP(M_s.r_2, S)$  then
16   |  $lca_{12} \leftarrow LCA(M_s.r_1, M_s.r_2)$ 
17   |  $lca_{1s} \leftarrow LCA(M_s.r_1, S)$ 
18   |  $lca_{2s} \leftarrow LCA(M_s.r_2, S)$ 
19   | if  $lca_{1s} >_{dpst} lca_{12} \vee lca_{2s} >_{dpst} lca_{12}$  then
20     |  $M_s.r_1 \leftarrow S$ 
21   | end
22 end

```

---

#### 4.1.4 Soundness and Precision

Like the ESP-bags algorithm, the SPD3 algorithm also detects a data race in a program for a given input if and only if a data race exists. Hence, SPD3 is precise and sound for a given input, which will be proved in Section 5.2. The other guarantees regarding soundness and precision of SPD3 are also exactly the same as that for the ESP-bags algorithm.

If the SPD3 does not report a data race during an execution of a program  $P$  for a given input, then it is guaranteed that there is no data race in any schedule of  $P$  for the given input. On the other hand, if a race is found, then it is guaranteed that there is some schedule of  $P$  with the given input for which the reported race is the first one encountered. There may be other schedules of  $P$  with the given input which may encounter a different set of races in a different order, but all schedules are guaranteed to encounter a data race. Also, the soundness and precision guarantees of the SPD3 algorithm hold only until the first data race.

#### 4.1.5 Space Overhead

Now, we discuss the space overhead of our SPD3 algorithm. First, we define *live steps* and *live internal nodes* in a DPST.

**Definition 4.1.7.** A step  $S$  in a DPST is said to be a *live step* at a point  $P$  during program execution if  $S$  is referenced by at least one shadow memory at the point  $P$ .

**Definition 4.1.8.** An internal node  $N$  in a DPST is said to be a *live internal node* at a point  $P$  during program execution if there is a *live step* in the subtree rooted at  $N$  at the point  $P$ .

Note that, though the entire DPST is built during program execution, we only

need to retain those leaf and internal nodes that are *live* at any point during program execution. The *dead* nodes (nodes that are not *live*) can be removed from the DPST because they will never participate in checking for data races in the program from that execution point.

Suppose that a maximum of  $v$  shared memory locations are active at any point during the execution of a program. The maximum number of shadow memory during the program execution is also  $v$ . Each of these  $v$  shadow memory may refer to up to three steps in the DPST. Assuming each field in every shadow memory refers to a unique step in the DPST, the total number of *live steps* is  $v * 3$ . Hence the maximum number of leaf nodes in the DPST at any point during program execution is  $v * 3$ .

The number of *live internal nodes* at any point during program execution is bounded by the number of `async` and `finish` instances in the program execution. Theoretically, the number of `async` and `finish` instances in a program execution could be arbitrarily large. However, in practice, the number of `async` and `finish` instances in a program execution is much smaller than the maximum number of active memory locations  $v$  in that program execution.

The space overhead for every memory location is  $O(1)$ , since we only need to store a writer step and two reader steps in the shadow memory of every memory location.

#### 4.1.6 Time Overhead

The time overhead at task boundaries is  $O(1)$ , which is the time needed to add/update a node in the DPST. The worst case time overhead on every memory access is same as that of Algorithm 2, which is the length of the longer of the two paths involved in the LCA computation. This is bounded by the height,  $H$ , of the DPST.

The height,  $H$ , of the DPST corresponds to the nesting depth of `async` and `finish`

instances in the program execution. Though, in theory, the nesting depth of `async` and `finish` instances in a program execution could be arbitrarily large, most practical applications do not have large nesting depth of `async` and `finish` instances. This is due to the high space requirements of parallel programs with large nesting depths. Narlikar and Blelloch [44] show that the space requirement of a parallel schedule of a program with a nesting depth  $D$  is linearly proportional to  $D * p$ , where  $p$  is the numbers of processors on which the program executes. This clearly shows that the space requirements of parallel programs with large nesting depth become intractable when running on highly parallel systems.

In the SPD3 algorithm, when the nesting depth of `async` and `finish` instances is large, the LCA computation can be optimized in two ways: 1) by maintaining a cache of the LCA of pairs of nodes and looking up the cache before computing new LCAs, and 2) by deleting the internal nodes in the subtree rooted at any *live* node whose execution is complete (path compression).

Note that the time overhead on memory access is not proportional to the number of processors (underlying worker threads) on which the program executes. Hence, the overhead is not expected to increase as we increase the number of processors on which the program executes. This is an important property as future hardware will likely have many cores.

## 4.2 Extending SPD3 for Isolated Blocks

In this section, we describe an extension to the SPD3 algorithm to handle HJ programs with isolated blocks. For this we generalize the shadow memory used in the SPD3 algorithm to handle operations other than reads and writes. Recollect that the SPD3 algorithm stores one *writer* and two *readers* in the shadow memory of every memory

location.

We need to store only one *writer* in the shadow memory of every memory location because two or more writes to the same memory location in parallel will result in a data race. In the case of reads, two or more reads to the same memory location in parallel will not result in a data race. Hence, we need to store two *readers* to cover an entire subtree of readers. This can be generalized to any operation as follows:

- Let us consider an operation  $\chi_1$  such that two or more of  $\chi_1$  on the same memory location in parallel will result in a data race. We only need one field in the shadow memory of every memory location to represent  $\chi_1$ .
- Let us consider an operation  $\chi_2$  such that two or more of  $\chi_2$  on the same memory location in parallel will not result in a data race. We need two fields in the shadow memory of every memory location to represent  $\chi_2$ .

#### 4.2.1 Shadow Memory with Isolated

To handle *isolated* blocks, the SPD3 algorithm has to check that the isolated and non-isolated accesses to memory locations do not conflict. This is done by treating isolated reads and writes to memory locations as distinct operations compared to the non-isolated reads and writes. When two or more isolated-reads to the same memory location happen in parallel, they do not result in a data race. Hence, according to the above generalization, we need two fields in the shadow memory to handle isolated-reads. Similarly, since two or more isolated-writes to the same memory location happening in parallel do not result in a data race, we need two fields for isolated-writes in the shadow memory. Thus, we add the following four fields to the shadow memory to handle isolated-reads and isolated-writes.

- $ir_1$  : a reference to a step that performed an isolated-read of M.
- $ir_2$  : a reference to another step that performed an isolated-read of M.
- $iw_1$  : a reference to a step that performed an isolated-write on M.
- $iw_2$  : a reference to another step that performed an isolated-write on M.

Note that, as in the case of the extended ESP-bags algorithm to support `isolated` blocks, these additional fields need only be added to memory locations that are accessed within `isolated` blocks.

#### 4.2.2 Extended SPD3 Algorithm

Now we describe the extension to the SPD3 algorithm to support `isolated` blocks. The change needed in the SPD3 algorithm is to handle isolated read and write operations differently as compared to non-isolated reads and writes. Algorithms 5 and 6 show the modified Read-Check and Write-Check algorithms with support for `isolated` blocks. In other words, these two algorithms give the steps that need to be performed on read and write operations when the additional four fields are present in the shadow memory to support isolated read and isolated write operations. Algorithms 7 and 8 give the steps that need to be performed on isolated read and isolated write operations on memory locations.

Algorithm 5 shows a modified version of Algorithm 4 with support for handling isolated reads and writes. Since a non-isolated read could conflict with an isolated write, this algorithm includes steps in lines 4-9 to check if the current reader conflicts with the previous isolated writers, if any. These are the only modifications to Algorithm 5 compared to Algorithm 4.



---

**Algorithm 5:** Read Check w/ support for Isolated
 

---

**Input:** Memory location  $M$ , Step  $S$  that reads  $M$

```

1 if  $DMHP(M_s.w, S)$  then
2   | Report a write-read data race between  $M_s.w$  and  $S$ 
3 end

4 if  $DMHP(M_s.iw_1, S)$  then
5   | Report an isolated write - read data race between  $M_s.iw_1$  and  $S$ 
6 end

7 if  $DMHP(M_s.iw_2, S)$  then
8   | Report an isolated write - read data race between  $M_s.iw_2$  and  $S$ 
9 end

10 if  $M_s.r_1 = null$  then  $M_s.r_1 \leftarrow S$ 

11 else if  $M_s.r_2 = null$  then
12   | if  $DMHP(M_s.r_1, S)$  then  $M_s.r_2 \leftarrow S$ 
13   | else  $M_s.r_1 \leftarrow S$ 

14 else if  $\neg DMHP(M_s.r_1, S) \wedge \neg DMHP(M_s.r_2, S)$  then
15   |  $M_s.r_1 \leftarrow S$ 
16   |  $M_s.r_2 \leftarrow null$ 

17 else if  $DMHP(M_s.r_1, S) \wedge DMHP(M_s.r_2, S)$  then
18   |  $lca_{12} \leftarrow LCA(M_s.r_1, M_s.r_2)$ 
19   |  $lca_{1s} \leftarrow LCA(M_s.r_1, S)$ 
20   |  $lca_{2s} \leftarrow LCA(M_s.r_2, S)$ 
21   | if  $lca_{1s} >_{dpst} lca_{12} \vee lca_{2s} >_{dpst} lca_{12}$  then  $M_s.r_1 \leftarrow S$ 

22 end

```

---

---

**Algorithm 6:** Write Check w/ support for Isolated
 

---

**Input:** Memory location  $M$ , Step  $S$  that writes to  $M$

```

1 if  $DMHP(M_s.r_1, S)$  then
2   | Report a read-write race between  $M_s.r_1$  and  $S$ 
3 end

4 if  $DMHP(M_s.r_2, S)$  then
5   | Report a read-write race between  $M_s.r_2$  and  $S$ 
6 end

7 if  $DMHP(M_s.ir_1, S)$  then
8   | Report an isolated read - write race between  $M_s.ir_1$  and  $S$ 
9 end

10 if  $DMHP(M_s.ir_2, S)$  then
11  | Report an isolated read - write race between  $M_s.ir_2$  and  $S$ 
12 end

13 if  $DMHP(M_s.iw_1, S)$  then
14  | Report an isolated write - write race between  $M_s.iw_1$  and  $S$ 
15 end

16 if  $DMHP(M_s.iw_2, S)$  then
17  | Report an isolated write - write race between  $M_s.iw_2$  and  $S$ 
18 end

19 if  $DMHP(M_s.w, S)$  then
20  | Report a write-write race between  $M_s.w$  and  $S$ 
21 else
22  |  $M_s.w \leftarrow S$ 
23 end

```

---

Algorithm 6 shows a modified version of Algorithm 3 with support for handling isolated reads and writes. Since a non-isolated write could conflict with both isolated reads and isolated writes, this algorithm includes steps in lines 7-18 to check if the current writer conflicts with the previous isolated readers and isolated writers, if any. These are the only modifications to Algorithm 6 compared to Algorithm 3.

Algorithm 7 gives the steps that need to be performed on isolated reads of shared memory locations. An isolated read conflicts with non-isolated writes but does not conflict with isolated writes and non-isolated reads. So, the algorithm only checks if the current step that is performing an isolated read conflicts with the previous writer. Then, the algorithm updates the isolated reader fields with the new step,  $S$ , as follows: if  $S$  is the first step that performs an isolated read on  $M$ , then  $M_s.ir_1$  is set to  $S$ . If  $M_s.ir_2$  is null, then  $M_s.ir_2$  or  $M_s.ir_1$  is set to  $S$  depending on whether  $S$  may execute in parallel with  $M_s.ir_1$  or not. If  $S$  can never execute in parallel either of the two isolated readers,  $M_s.ir_1$  and  $M_s.ir_2$ , then both these isolated readers are discarded and  $M_s.ir_1$  is set to  $S$ . If  $S$  can execute in parallel with both the isolated readers,  $M_s.ir_1$  and  $M_s.ir_2$ , then the algorithm stores two of these three steps, whose LCA is the highest in the DPST. If  $S$  can execute in parallel with one of the isolated readers and not the other, then the algorithm does not update the isolated readers because, in that case,  $S$  is guaranteed to be within the subtree under the  $LCA(M_s.ir_1, M_s.ir_2)$ .

Algorithm 8 gives the steps to be performed on isolated writes of shared memory locations. An isolated write conflicts with non-isolated reads and non-isolated writes but does not conflict with isolated reads. Hence, the algorithm checks if the current step that is performing an isolated write conflicts with the previous writer or any of the previous readers. Then, the algorithm updates the isolated writer fields with the

---

**Algorithm 7:** Isolated Read Check w/ support for Isolated
 

---

**Input:** Memory location  $M$ , Step  $S$  that reads  $M$

```

1 if  $DMHP(M_s.w, S)$  then
2   | Report a write - isolated read data race between  $M_s.w$  and  $S$ 
3 end
4 if  $M_s.ir_1 = null$  then  $M_s.ir_1 \leftarrow S$ 
5 else if  $M_s.ir_2 = null$  then
6   | if  $DMHP(M_s.ir_1, S)$  then  $M_s.ir_2 \leftarrow S$ 
7   | else  $M_s.ir_1 \leftarrow S$ 
8 else if  $\neg DMHP(M_s.ir_1, S) \wedge \neg DMHP(M_s.ir_2, S)$  then
9   |  $M_s.ir_1 \leftarrow S$ 
10  |  $M_s.ir_2 \leftarrow null$ 
11 else if  $DMHP(M_s.ir_1, S) \wedge DMHP(M_s.ir_2, S)$  then
12  |  $lca_{12} \leftarrow LCA(M_s.ir_1, M_s.ir_2)$ 
13  |  $lca_{1s} \leftarrow LCA(M_s.ir_1, S)$ 
14  |  $lca_{2s} \leftarrow LCA(M_s.ir_2, S)$ 
15  | if  $lca_{1s} >_{dpst} lca_{12} \vee lca_{2s} >_{dpst} lca_{12}$  then
16  | |  $M_s.ir_1 \leftarrow S$ 
17  | end
18 end

```

---

---

**Algorithm 8:** Isolated Write Check w/ support for Isolated

---

**Input:** Memory location  $M$ , Step  $S$  that reads  $M$

```

1 if  $DMHP(M_s.w, S)$  then
2   | Report a write - isolated write data race between  $M_s.w$  and  $S$ 
3 end

4 if  $DMHP(M_s.r_1, S)$  then
5   | Report a read - isolated write data race between  $M_s.r_1$  and  $S$ 
6 end

7 if  $DMHP(M_s.r_2, S)$  then
8   | Report a read - isolated write data race between  $M_s.r_2$  and  $S$ 
9 end

10 if  $M_s.iw_1 = null$  then  $M_s.iw_1 \leftarrow S$ 

11 else if  $M_s.iw_2 = null$  then
12   | if  $DMHP(M_s.iw_1, S)$  then  $M_s.iw_2 \leftarrow S$ 
13   | else  $M_s.iw_1 \leftarrow S$ 

14 else if  $\neg DMHP(M_s.iw_1, S) \wedge \neg DMHP(M_s.iw_2, S)$  then
15   |  $M_s.iw_1 \leftarrow S$ 
16   |  $M_s.iw_2 \leftarrow null$ 

17 else if  $DMHP(M_s.iw_1, S) \wedge DMHP(M_s.iw_2, S)$  then
18   |  $lca_{12} \leftarrow LCA(M_s.iw_1, M_s.iw_2)$ 
19   |  $lca_{1s} \leftarrow LCA(M_s.iw_1, S)$ 
20   |  $lca_{2s} \leftarrow LCA(M_s.iw_2, S)$ 
21   | if  $lca_{1s} >_{dpst} lca_{12} \vee lca_{2s} >_{dpst} lca_{12}$  then  $M_s.iw_1 \leftarrow S$ 

22 end

```

---

new step, S, in the same way as the isolated reader fields are updated in Algorithm 7.

### 4.2.3 Soundness and Precision

The extended SPD3 algorithm with support for `isolated` is precise but not sound, i.e., the algorithm has no false positives but may have some false negatives. With `isolated` blocks, there may be cases where the execution that is monitored during the SPD3 algorithm does not execute certain parts of code which may execute in other executions of the program for the same input. This happens when the `isolated` blocks in the program do not commute. This is exactly the same reason why the ESP-bags algorithm is precise but not sound in the presence of `isolated` blocks, as described in Section 3.2.2.

Note that when the `isolated` blocks in the given program commute, the SPD3 algorithm is both precise and sound for a given input.

### 4.2.4 Space Overhead

The extended SPD3 algorithm with support for `isolated` uses four additional fields in the shadow memory location to store the isolated readers and writers. These additional fields can be restricted to only those memory locations that accessed within `isolated` blocks, in the same way as described in Section 3.2.2. Typically, the `isolated` blocks in programs are very small and hence, only a few memory locations are accessed within `isolated` blocks. Thus, the increase in the space overhead is very small in most programs. Since we only add constant number of fields to the shadow memory of some memory locations, the asymptotic space overhead remains the same as in the original SPD3 algorithm for `async` and `finish`.

#### 4.2.5 Time Overhead

The time overhead for the SPD3 increases in the presence of `isolated` blocks because of the extra checks that need to be performed for every operation on a memory location. In the worst case, we need to check if the new step conflicts with all the 7 fields in the shadow memory location. Thus, we may have to do 7 DMHP computations for every memory access in the worst case. But the additional checks for conflicts need to be done only for those memory locations that are accessed within `isolated` blocks. Since these memory locations are typically small in number, the extra time spent on these checks is typically less. Since we only have to do a constant number of DMHP computations in addition to the original algorithm, the asymptotic time overhead remains the same as the original SPD3 algorithm for `async` and `finish`.

### 4.3 Extending SPD3 for Futures

We now describe an extension to the SPD3 algorithm to support HJ programs with `futures`. Though this algorithm can be used for `futures` along with `isolated`, for the sake of simplicity we restrict our presentation in this section to programs with `async`, `finish`, and `futures`. Recall that the `future` construct is used to create a task with a handle which can be used to wait on this task, specifically. The `get` operation on a future, `G`, denoted by `G.get()`, waits for the task pointed to by the handle `G` to complete.

For the purpose of data race detection, we consider every `future` as an `async` with a special property. Hence, all futures are asyncs but not all asyncs are futures.

In this section, we first describe the modifications needed on the Dynamic Program Structure Tree (DPST) used in the SPD3 algorithm to support `futures`. Then, we

present an enhanced DPST which encodes the information regarding **futures** in a sophisticated manner so as to make the DMHP computation on the DPST faster in certain scenarios. Then, we present the extensions needed in the shadow memory of memory locations to support **futures**. Finally, we present the changes needed in the SPD3 algorithm to support **futures**. We also discuss about the soundness and precision guarantees of SPD3 in the presence of **futures** and also about how **futures** affect the space and time overhead of the algorithm.

#### 4.3.1 Dynamic Program Structure Tree with Futures

Every instance of a **future** will get a node in the DPST, just like an **async**. Since every **future** is also an **async**, any test for an **async** node will succeed on a **future** node as well. But since every **async** is not a **future**, a test for a **future** node will not succeed on a non-**future** **async** node.

Every instance of a **get** operation on a **future**,  $G$ , in the program is represented by a new leaf node,  $G_g$ . Now, the DPST contains two kinds of leaf nodes, **get** and **step** nodes. Also, every **future** node,  $G$ , in the DPST maintains a list of pointers,  $G_{gets}$ , which point to the  $G_g$  nodes corresponding to the **get** operations performed on this **future**.

In this extended DPST, a **step** is redefined as follows:

**Definition 4.3.1.** A **step** is a maximal sequence of statement instances such that no statement instance in the sequence includes the start or end of an **async**, the start or end of a **finish** or a **get** operation. □

Now, **get** operations, in addition to the start and end of **async** and **finish** operations, define the boundary of a **step**. When a program completes a **get** operation on a **future**  $G$ , a corresponding **get** node is added to the DPST as follows: suppose a **step**  $S$



performs a **get** on a **future**  $G$ , we split  $S$  around the **get** operation. Let  $S'$  denote the part of  $S$  before the **get** and  $S''$  denote the part of  $S$  after the **get**. Now, we insert  $S'$  in the place of  $S$  in the DPST. The **get** operation following  $S'$  gets a node,  $G_g$ , which is added as the right sibling of  $S'$ . Then,  $S''$  is added as the right sibling of  $G_g$ . Also, a pointer to this **get** node,  $G_g$ , is added to the list  $G_{gets}$ .

We now look at some definitions and notations that will be used in the rest of section.

**Definition 4.3.2.** The path to the root of a node,  $N$ , in the DPST, is the path from  $N$  to the root of the DPST. The set  $P(N)$  denotes the set of all nodes in the path from  $N$  to the root of the DPST.

$$P(N) = \{\text{Nodes in the path from } N \text{ to root}\} \quad \square \quad (4.1)$$

**Definition 4.3.3.** Parallelism Defining Nodes (PDNs) for a node,  $N$ , in a DPST, denoted by  $\text{PDN}(N)$ , is the set of all nodes,  $K$ , along the path from  $N$  to the root of the DPST, such that the child of  $K$  along this path is not an **async**, i.e.,

$$\begin{aligned} \text{PDN}(N) = \{K : K \in P(N) \text{ and } K_c \text{ is not an } \text{async}, \\ \text{where } K_c \text{ is the child of } K \text{ in } P(N)\} \quad \square \quad (4.2) \end{aligned}$$

**Definition 4.3.4.** Parallelism Defining Asyncns (PDAs) for a node,  $N$ , in a DPST, denoted by  $\text{PDA}(N)$ , is the set of all **async** nodes in  $\text{PDN}(N)$ .

$$\text{PDA}(N) = \{K : K \in \text{PDN}(N) \text{ and } K \text{ is an } \text{async}\} \quad \square \quad (4.3)$$

**Definition 4.3.5.** Parallelism Defining Futures (PDFs) for a node,  $N$ , in a DPST, denoted by  $\text{PDF}(N)$ , is the set of all **future** nodes in  $\text{PDA}(N)$ .

$$\text{PDF}(N) = \{K : K \in \text{PDA}(N) \text{ and } K \text{ is a } \text{future}\} \quad \square \quad (4.4)$$

### Dynamic May Happen in Parallel (DMHP) on the extended DPST

---

**Algorithm 9:** Dynamic May Happen in Parallel (DMHP) on DPST extended with Futures

---

**Input:** DPST  $\Gamma$ , Step  $S$ , Step  $S'$

**Output:** true/false

```

1  $S_1 \leftarrow \text{Left}(S, S')$ 
2  $S_2 \leftarrow \text{Right}(S, S')$ 
3  $N_{lca} \leftarrow \text{LCA}(\Gamma, S_1, S_2)$ 
4  $A_1 \leftarrow \text{Ancestor of } S_1 \text{ which is the child of } N_{lca}$ 
5 if  $A_1$  is not an async then
6   | return false
7 end
8 for  $G \in \text{PDF}(S_1)$  do
9   | for  $G_g \in G_{\text{gets}}$  do
10  |   | if  $\text{DMHP}(G_g, S_2) = \text{false}$  and  $\text{Left}(G_g, S_2) = G_g$  then
11  |   |   | return false
12  |   |   end
13  |   end
14 end
15 return true

```

---

With the DPST extended to support futures, we now describe the computation of DMHP on this extended DPST. Algorithm 9 computes the DMHP relation for two steps,  $S$  and  $S'$ , when the DPST contains `async`, `finish`, and `future` constructs. The algorithm first stores the left of the two input steps in  $S_1$  and the right of the two

steps in  $S_2$ . It then computes the Lowest Common Ancestor (LCA),  $N_{lca}$ , of the steps  $S_1$  and  $S_2$  using Algorithm 1. Suppose  $A_1$  denotes the ancestor of  $S_1$  which is the child of  $N_{lca}$ . If  $A_1$  is not an `async`, then the algorithm returns false. This follows directly from the original DMHP algorithm, Algorithm 2.

When the node  $A_1$  is an `async` node, we need to check if any of the *parallelism defining futures* of  $S_1$  (i.e., any node in the set  $\text{PDF}(S_1)$ ) is guaranteed to happen before  $S_2$ . Suppose a node,  $K$ , in  $\text{PDF}(S_1)$  is guaranteed to happen before  $S_2$ . Since  $K$  is in  $\text{PDF}(S_1)$ , the child node of  $K$  along the path from  $S_1$  to root,  $K_c$ , is not an `async` (so,  $K_c$  is either  $S_1$  or a `finish` node whose subtree contains  $S_1$ ). Hence, it is guaranteed that  $S_1$  happens before  $S_2$ . To check if any node in  $\text{PDF}(S_1)$  happens before  $S_2$ , the algorithm recursively calls itself for every node in  $\text{PDF}(S_1)$  along with  $S_2$ . If any of the nodes in  $\text{PDF}(S_1)$  is guaranteed to never happen in parallel with  $S_2$  and if that node is to the left of  $S_2$  in the DPST, then that node happens before  $S_2$ , in which case the algorithm returns false to denote that  $S_1$  can never happen in parallel with  $S_2$ . The intuition behind this is to check if there exists a happens before ordering between any `get` operation and  $S_2$  which could transitively impose a happens before ordering between  $S_1$  and  $S_2$ .

Note that non-future nodes (other than  $A_1$ ) along the path from  $S_1$  to the root of the DPST do not impose an ordering between  $S_1$  and  $S_2$  because  $A_1$  being an `async` node guarantees that  $S_1$  and  $S_2$  may execute in parallel unless there is an explicit ordering imposed through a `get`. The future nodes along the path from  $S_1$  to the root, other than those in  $\text{PDF}(S_1)$ , also do not impose an ordering between  $S_1$  and  $S_2$ . This is because the child of each such future node along the path from  $S_1$  to the root is an `async` node and the subtree under this `async` node is not guaranteed to complete even when the future is guaranteed to complete.

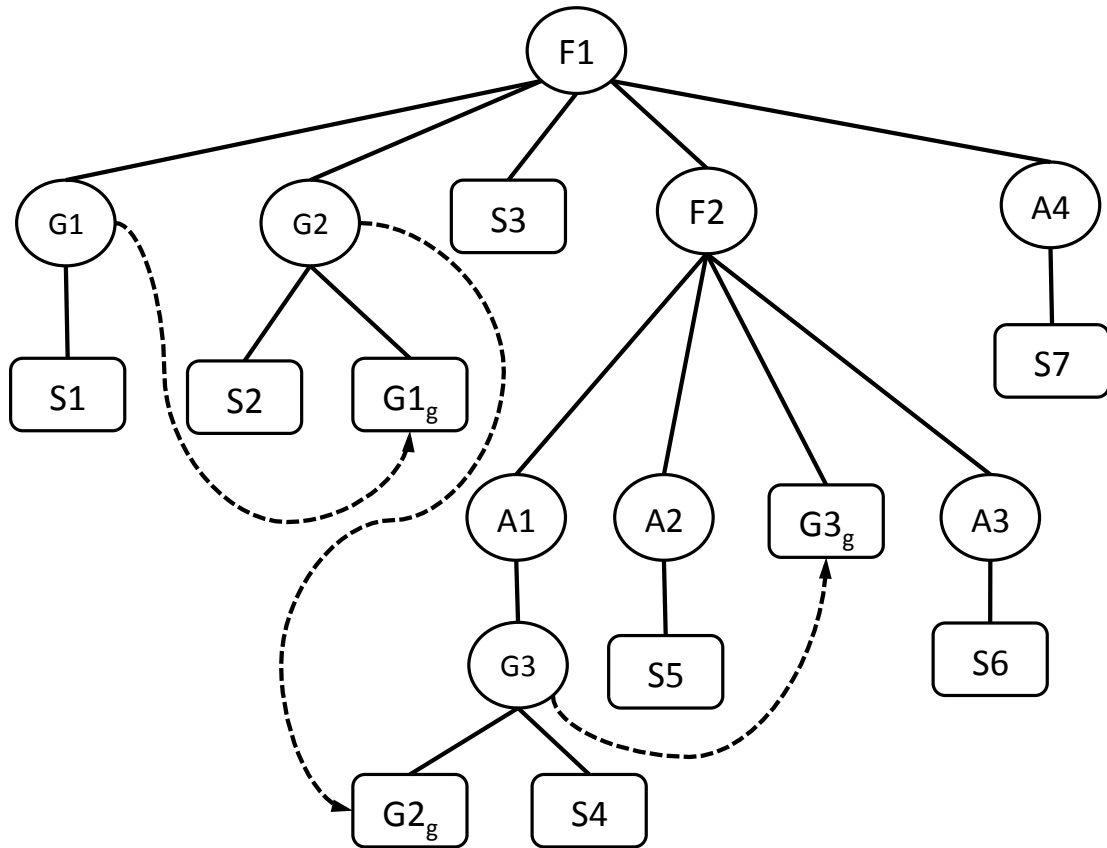


Figure 4.3 : An example DPST with async, finish, and future nodes. F1 and F2 are finish nodes; G1, G2, and G3 are future nodes; A1, A2, A3, and A4 are non-future async nodes. The nodes S1-S7 refer to the step nodes. The dotted lines denote the pointers from future nodes to their get operations.

In summary, the algorithm returns false when there is a node,  $G$ , in  $\text{PDF}(S_1)$ , such that for one of its get nodes,  $G_g$ ,  $\text{DMHP}(G_g, S_2)$  is false and  $G_g$  is to the left of  $S_2$  in the DPST, i.e.,  $G_g$  is guaranteed to happen before  $S_2$ . When no such node exists, the algorithm returns true implying that the steps  $S_1$  and  $S_2$  may execute in parallel.

### Example

Now, we show an example of DPST with futures and gets in Figure 4.3. This DPST contains **finish** nodes (F1 and F2), non-future **async** nodes (A1, A2, A3, and A4), **future** nodes (G1, G2, and G3), and **step** nodes (S1 - S7). The pointers from **future** nodes to their corresponding **get** nodes are denoted by dotted lines in Figure 4.3. Note that the only difference in this extended DPST are the leaf nodes for **get** operations and pointers from **future** nodes to their corresponding **get** nodes.

We now sketch a couple of DMHP computations on the DPST in Figure 4.3. First, let us consider the computation of DMHP(S4, S5). S4 is to the left of S5 and A1 is the ancestor of S4 which is the child of the LCA(S4, S5). Since A1 is an **async** node, the algorithm investigates further.  $PDF(S4) = \{G3\}$ . The algorithm then follows G3's pointer to its **get** nodes (only one **get** node for G3). This results in recursively calling DMHP with  $G3_g$  and S5 as input nodes. Now, S5 is the left node and A2 is the ancestor of S5 which is the child of LCA( $G3_g$ , S5). Since A2 is an **async** and  $PDF(S5)$  is empty, DMHP( $G3_g$ , S5) returns true. Thus DMHP(S4, S5) also returns true which correctly reflects the fact that S4 and S5 may execute in parallel.

Let us consider the computation of DMHP(S6, S4). The computation proceeds as in the case above until the recursive call to DMHP, which will now be with inputs  $G3_g$  and S6. Now,  $G3_g$  is the left node and also the node which is the child of LCA( $G3_g$ , S6). Since it is not an **async** node, DMHP( $G3_g$ , S6) returns false. Hence, DMHP(S6, S4) also returns false. This correctly reflects the fact that S4 and S6 can never execute in parallel due to ordering imposed by the **get** node of G3,  $G3_g$ .

## Discussion

To check if a `get` operation on a future,  $G_g$ , *happens before* a given step,  $S$ , Algorithm 9 checks that  $G_g$  and  $S$  can never happen in parallel (by computing the DMHP relation on them) and that  $G_g$  is to the left of  $S$  in the DPST. These two checks are necessary to confirm that  $G_g$  happens before  $S$ , on a complete DPST. But, during the execution of the program, while the DPST is still growing, the DMHP computation will be performed only when one of the two steps is being executed. So, when Algorithm 9 is being called to compute the DMHP of steps  $S_1$  and  $S_2$ , one of  $S_1$  or  $S_2$  must be executing. If  $S_1$  was executing, then none of the `get` operations on any of its PDFs would have completed. Hence, in this case, Algorithm 9 will not enter the loop in line 9. If  $S_2$  was executing, then the fact that  $\text{DMHP}(G_g, S_2)$  is *false* is enough to ascertain that  $G_g$  happens before  $S_2$ . Hence, the check for Left relation in line 10 of Algorithm 9 is not necessary, if the algorithm is used to compute DMHP when one of the two steps involved is being executed.

## Space and Time Overhead

The number of nodes in the DPST increases with nodes added for `get` operations. So, the DPST now has as many additional nodes as the number of `get` instances in the program. Every `future` node also stores a list of pointers to their `get` nodes in the DPST. The total size required to store these lists over all the `future` nodes is equal to the number of `get` instances in the program. Hence, the total size of the DPST is  $O(a + f + g)$ , where  $a$  refers to the number of `async` instances (including `future` instances),  $f$  refers to the number of `finish` instances, and  $g$  refers to the number of `get` instances in the program. Note that there is no need for a term to represent the step instances because the number of step instances is proportional to the number of

async, finish, and get instances in the program.

The algorithm to compute the DMHP relation between two **steps** in the DPST may now involve recursive calls to the DMHP algorithm. In the worst case, every DMHP computation between two **steps** could recursively call DMHP once for every **get** instance in the program. Hence, the time required to compute the DMHP relation between two **steps** in the DPST is  $O(L * g)$ , where  $L$  is the average length of the longer path in the LCA computation and  $g$  is the number of **get** instances in the program. This is bounded by  $O(h * g)$ , where  $h$  is the height of the DPST.

#### 4.3.2 Enhancing the DPST with Futures

We now describe an enhancement to the DPST with futures to reduce the time overhead of DMHP computation in certain scenarios. This enhancement involves replacing the pointers from **future** nodes to their corresponding **get** nodes by a summarized information about the **get** nodes in the DPST. The downside is that the space overhead of the DPST increases due to this enhancement.

The enhancement is done by associating sequence numbers with every node in the DPST. The sequence number of a node  $N$  in the DPST, referred to as  $N_{seq}$ , represents the ordering among the children of a node. Specifically, a value of  $k$  in  $N_{seq}$  for a node  $N$  means that  $N$  is the  $k^{th}$  child of its parent in the DPST.

In this enhanced DPST, there will be no pointers from every **future** node to its corresponding **get** nodes. Instead, every internal node,  $N$ , will maintain a list,  $N_{gets}$ , which stores a list of entries of the form *future:seq*. There will be at most one entry per **future** in  $N_{gets}$ , for any node  $N$ . An entry  $G:k$  in  $N_{gets}$  indicates that  $G.get()$  (and hence the **future**  $G$ ) is guaranteed to complete at the end of the execution of the  $k^{th}$  child of  $N$ .

Now, we describe the steps to update the *gets* list associated with every internal node in the DPST. When a program completes a **get** operation on a **future**  $G$ , a corresponding **get** node is added to the DPST as described in Section 4.3.1. Additionally, the following steps are performed to update the *gets* list:

1.  $G_g \leftarrow$  Node in the DPST corresponding to  $G.get()$
2. For every  $N$  in  $PDN(G_g)$ :
  - (a)  $C \leftarrow$  child( $N$ ) in  $P(G_g)$
  - (b) If  $N_{gets}$  does not contain “ $G:s$ ” (for some  $s$ ), then add “ $G:C_{seq}$ ” to  $N_{gets}$
3. Repeat the above step for every entry in  $G_{gets}$ .

When the program completes a **get** operation,  $G_g$ , on a **future**  $G$ , the *gets* list of all the nodes in the set of *parallelism defining nodes*,  $PDN(G_g)$ , are updated to include an entry for  $G$  corresponding to this **get**. The entry will include the sequence number of the child through which this **get** operation is guaranteed to complete. Also, all the entries in the set  $G_{gets}$  are added to the *gets* list of all the nodes in  $PDN(G_g)$ . This is because all the **futures** that complete within the **future**  $G$  are also guaranteed to complete at the end of  $G$ .

### **Dynamic May Happen in Parallel (DMHP) on the Enhanced DPST**

Algorithm 10 computes the DMHP relation on the enhanced DPST. This algorithm is exactly the same as Algorithm 9 until line 7, i.e., until checking if the node  $A_1$ , which is the ancestor of the left node and the child of the LCA of the two nodes, is an async node. After that point, this algorithm investigates the path from the node  $S_2$  (the right node among the two input nodes) to  $N_{lca}$  (the LCA of  $S_1$  and  $S_2$ ). The



---

**Algorithm 10:** Dynamic May Happen in Parallel (DMHP) on the enhanced DPST

---

**Input:** DPST  $\Gamma$ , Step  $S$ , Step  $S'$

**Output:** true/false

- 1  $S_1 \leftarrow \text{Left}(S, S')$
- 2  $S_2 \leftarrow \text{Right}(S, S')$
- 3  $N_{lca} \leftarrow \text{LCA}(\Gamma, S_1, S_2)$
- 4  $A_1 \leftarrow \text{Ancestor of } S_1 \text{ which is the child of } N_{lca}$
- 5 **if**  $A_1$  *is not an async* **then**
- 6 **return** false
- 7 **end**
- 8  $P_2 \leftarrow \{ N : N \text{ is in the path from } S_2 \text{ to } N_{lca} \}$
- 9 **for**  $N \in P_2$  **do**
- 10  $C \leftarrow \text{child}(N) \text{ in } P_2$
- 11 **if**  $N_{\text{gets}}$  *contains "G:n"* and  $G \in \text{PDF}(S_1)$  **then**
- 12 **if**  $n < C_{\text{seq}}$  **then**
- 13 **return** false
- 14 **end**
- 15 **end**
- 16 **end**
- 17 **return** true

---

algorithm checks the *gets* list of every node in this path and looks for an entry for one of the futures in  $\text{PDF}(S_1)$ . If any future in  $\text{PDF}(S_1)$  is guaranteed to complete before any node in the path from  $S_2$  to  $N_{lca}$ , then the DMHP algorithm returns false indicating that the steps  $S_1$  and  $S_2$  cannot run in parallel. The algorithm returns true when none of the *gets* lists contains such an entry. Note that the algorithm traverses the path from  $S_2$  to  $N_{lca}$  only once. But it has to search the *gets* list of every node in this path.

### Example

An example of the enhanced DPST with *async*, *finish*, and *future* nodes is shown in Figure 4.4. This is the enhanced DPST for the example in Figure 4.3. Note that this DPST does not contain pointers from the *future* nodes to their corresponding *get* nodes. Instead every internal node maintains the list, *gets*, of all the futures that are guaranteed to complete along some path in its subtree.

Let us now sketch the computations of DMHP on this enhanced DPST. First, let us consider the computation of  $\text{DMHP}(S_4, S_5)$ .  $S_4$  is the left node among the two inputs and  $A_1$  is the ancestor of  $S_4$  and a child of  $F_2$  (which is the LCA of  $S_4$  and  $S_5$ ). Since  $A_1$  is an *async*, the algorithm proceeds to investigate further. It then looks at the path from  $S_5$  to  $F_2$ . Though the *gets* list of  $F_2$  contains  $A_4$  which is in  $\text{PDN}(S_4)$ , the sequence number associated with that entry in  $F_2$  is 3. This means that  $G_3$  is guaranteed to complete only after the 3<sup>rd</sup> child of  $F_2$ . Since the path from  $S_5$  to  $F_2$  goes through the 2<sup>nd</sup> child of  $F_2$ , DMHP returns false.

Now, let us consider the computation of  $\text{DMHP}(S_6, S_4)$ . The computation proceeds as in the above case. But the entry  $G_3:3$  in the *gets* list of  $F_2$  is good enough to ensure that  $S_4$  completes before  $S_6$  begins. This is because  $G_3$  is guaranteed to

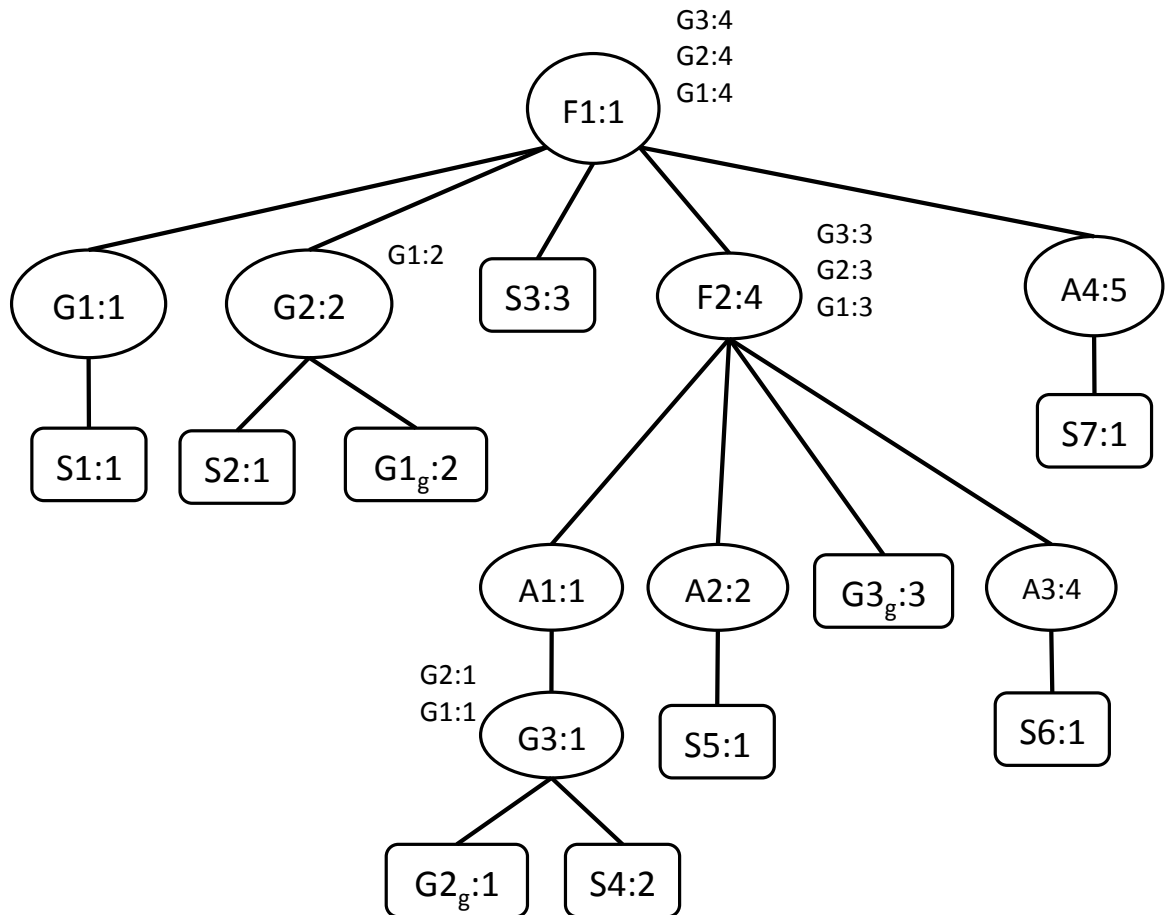


Figure 4.4 : An example of the enhanced DPST with *async*, *finish*, and *future* nodes. This is the enhanced version of the DPST in Figure 4.3. F1 and F2 are *finish* nodes; G1, G2, and G3 are *future* nodes; A1, A2, A3, and A4 are non-*future async* nodes. The nodes S1-S7 refer to the *step* nodes.

complete after the 3<sup>rd</sup> child of F2 and S6 is created only along the 4<sup>th</sup> child of F2.

### Space and Time Overhead

The enhanced DPST does not include the pointers from **future** nodes to corresponding **get** nodes but includes the *gets* list in every internal node. Note that the *gets* list is empty in most internal nodes. Every **finish** node, whose subtree contains some **get** instance, induces a non-empty *gets* list in its parent. This is because the parent of the **finish** node will be in the set of *parallelism defining nodes* of those **get** instances. Hence, the space overhead of the enhanced DPST is  $O(a + f + g + f * g)$ , where  $a$  refers to the number of **async** instances (including **future** instances),  $f$  refers to the number of **finish** instances, and  $g$  refers to the number of **get** instances in the program. Note that there is no need for a term to represent the step instances because the number of step instances is proportional to the number of **async**, **finish**, and **get** instances in the program. The enhancement increases the asymptotic space overhead of the DPST by a factor of  $f$ .

The advantage of the enhancement is that the algorithm to compute DMHP does not make any recursive calls to itself. The only extra work done during DMHP computation now is to look up the *gets* list in the path from the right node to the LCA of the two nodes. We assume that the look up on the *gets* list in every node takes  $O(1)$  time. Let  $S_1, S_2$  be the two nodes involved in the DMHP computation. Let  $S_1$  be to the left of  $S_2$  in the DPST. There can be at most  $h/2$  elements in the set of *parallelism defining futures* for  $S_1$ , where  $h$  is the height of the DPST. This is because, for a **future** to be in  $\text{PDF}(S_1)$ , its child along the path from  $S_1$  to the LCA has to be a **finish**. So, in the worst case, every alternate node along the path from  $S_1$  to the LCA could be a **future** in  $\text{PDF}(S_1)$ . This is also bounded by the number of

future instances in the program,  $ft$ . So, the size of the set  $\text{PDF}(S_1)$  is  $\min(h/2, ft)$ .

Every node along the path from  $S_2$  to LCA may contain its own *gets* list. This is also bounded by the number of *finish* instances in the program because every *finish* node may induce a *gets* list in its parent. So, the total number of nodes with *gets* list along the path from  $S_2$  to LCA is bounded by  $\min(h, f)$ , where  $f$  denotes the number of *finish* instances in the program. Since every *future* in the set  $\text{PDF}(S_1)$  has to be looked up in every *gets* list along the path from  $S_2$  to LCA, the total time required for this look up is  $O(\min(h/2, ft) * \min(h, f))$ .

When the height of the DPST is small, i.e., when the nesting depth of the *finish*, *async*, and *future* instances is low in the program, it would be beneficial to use this enhanced DPST. In that case, the time taken for DMHP computation would reduce considerably at the cost of increase in some space overhead.

### 4.3.3 Shadow Memory with Futures

Now, we discuss the changes that are needed in the shadow memory in SPD3 to detect data races in programs with *futures*. Recall that the shadow memory of the SPD3 algorithm for *async* and *finish* contains three fields;  $M_s.w$ ,  $M_s.r_1$ , and  $M_s.r_2$ . While  $M_s.w$  stores the most recent step that wrote  $M$ ,  $M_s.r_1$  and  $M_s.r_2$  store two steps that read  $M$  such that the subtree under their LCA includes all the steps that read  $M$  in parallel.

Since the soundness and precision guarantees of our algorithm hold only until the first data race, we assume that no data races have been observed so far. Hence, all the writes to every memory location must be ordered. So, even with *futures*, we need only one field,  $M_s.w$ , to store the most recent step that wrote  $M$ . But it is no longer sufficient to store just two readers to represent an entire subtree of readers. This is

because if a new access is not in parallel with both the stored readers,  $M_s.r_1$  and  $M_s.r_2$ , due to explicit `get` operations on some `futures`, then there is no guarantee that the new access is not in parallel with all the discarded readers.

We address this problem with the readers as follows. We store two steps that read  $M$ , in  $M_s.r_1$  and  $M_s.r_2$ , such that the  $\text{PDF}(M_s.r_1)$  and  $\text{PDF}(M_s.r_2)$  are empty, and the  $\text{LCA}(M_s.r_1, M_s.r_2)$ , say  $L$ , is the highest among every such pair of readers. In addition to these two readers, we also store a list of steps that read  $M$ ,  $M_s.fts$ , and are outside the subtree under  $L$  such that their PDF set is non-empty. Note that when two steps that read  $M$  have the same PDF set, we only need to store one of them, because any step that is parallel with one of them will be in parallel with the other as well. There is no need to store those steps that are within the subtree under  $L$  even if their PDF set is non-empty, because those steps will be covered by the two readers,  $M_s.r_1$  and  $M_s.r_2$ .

Also, we maintain a map from the LCA of these steps with  $M_s.r_1$ , to the steps that have this node as their LCA. This map from the LCA nodes to the steps is needed to prune the list  $M_s.fts$ , when steps in that list come within the subtree under  $\text{LCA}(M_s.r_1, M_s.r_2)$  as  $M_s.r_1$  and  $M_s.r_2$  are updated.

In summary, the shadow memory for a memory location  $M$  will contain the following fields for race detection using the SPD3 algorithm in the presence of `futures`.

- $M_s.w$  : a field that stores the most recent step that wrote  $M$ .
- $M_s.r_1$  : a field that stores a step  $S_1$  that read  $M$ , such that  $\text{PDF}(S_1)$  is empty.
- $M_s.r_2$  : a field that stores another step  $S_2$  that read  $M$ , such that  $\text{PDF}(S_2)$  is empty.  $\text{LCA}(S_1, S_2)$  is the highest among every such pair of steps that read  $M$  with their PDF set empty.

- $M_s.fts$  : a list that stores every step  $S$  that read  $M$ , such that:
  - $PDF(S)$  is *unique* and non-empty, and
  - $S$  is outside the subtree under  $LCA(M_s.r_1, M_s.r_2)$ .
- $M_s.lca\_map$  : Every step  $S$  in  $M_s.fts$  is outside the subtree under  $LCA(M_s.r_1, M_s.r_2)$ . Hence,  $LCA(S, M_s.r_1)$  is higher in the DPST than  $LCA(M_s.r_1, M_s.r_2)$ . This field maintains a *map* from every such higher LCA node to the set of steps that have this node as their LCA with  $M_s.r_1$ .

#### 4.3.4 Extended SPD3 Algorithm

We have already shown the changes needed in the DPST and the shadow memory of every memory location to support futures. Now, we describe the extension to the SPD3 algorithm to support futures.

Algorithm 11 show the modified Write-Check algorithm with support for futures. The only modification from the original Write-Check algorithm is the additional check done in the loop in line 7. This loop checks if the new step  $S$  that writes to  $M$  conflicts with any of the steps in the list  $M_s.fts$  and reports a data race if it finds any such conflicting steps.

Algorithm 12 shows the steps that need to be performed on read operations when the program contains futures. In this algorithm, if  $M_s.r_1$  is null, then it is set to  $S$  if the PDF set of  $S$  is empty or added to the list of steps,  $M_s.fts$ , using the function *AddStepToList*, if the PDF set of  $S$  is non-empty. The function *AddStepToList* adds the given step  $S$  to the list  $M_s.fts$  only if there are no other steps in  $M_s.fts$  whose PDF set is same as  $S$ .

If  $M_s.r_2$  is null, then the same steps as in Algorithm 4 are performed if the PDF

---

**Algorithm 11:** Write Check w/ support for Futures
 

---

**Input:** Memory location  $M$ , Step  $S$  that writes to  $M$

```

1 if  $DMHP(M_s.r_1, S)$  then
2   | Report a read-write race between  $M_s.r_1$  and  $S$ 
3 end
4 if  $DMHP(M_s.r_2, S)$  then
5   | Report a read-write race between  $M_s.r_2$  and  $S$ 
6 end
7 for  $S_f \in M_s.fts$  do
8   | if  $DMHP(S_f, S)$  then
9     | Report a read-write race between  $S_f$  and  $S$ 
10  | end
11 end
12 if  $DMHP(M_s.w, S)$  then
13  | Report a write-write race between  $M_s.w$  and  $S$ 
14 else
15  |  $M_s.w \leftarrow S$ 
16 end

```

---



---

**Algorithm 12:** Read Check w/ support for Futures
 

---

**Input:** Memory location  $M$ , Step  $S$  that reads  $M$

```

1 if  $DMHP(M_s.w, S)$  then Report a write-read data race between  $M_s.w$  and  $S$ 
2 if  $M_s.r_1 = null$  then
3   if  $PDF(S) = \emptyset$  then  $M_s.r_1 \leftarrow S$ 
4   else AddStepToList ( $S, M_s.fts$ )
5 else if  $M_s.r_2 = null$  then
6   if  $PDF(S) = \emptyset$  then
7     if  $DMHP(M_s.r_1, S)$  then  $M_s.r_2 \leftarrow S$ 
8     else  $M_s.r_1 \leftarrow S$ 
9   else AddStepToList ( $S, M_s.fts$ )
10 else if  $\neg DMHP(M_s.r_1, S) \wedge \neg DMHP(M_s.r_2, S)$  then
11    $M_s.r_1 \leftarrow M_s.r_2 \leftarrow null$ 
12   if  $PDF(S) = \emptyset$  then  $M_s.r_1 \leftarrow S$ 
13   else AddStepToList ( $S, M_s.fts$ )
14 else if  $DMHP(M_s.r_1, S) \wedge DMHP(M_s.r_2, S)$  then
15    $lca_{12} \leftarrow LCA(M_s.r_1, M_s.r_2)$ 
16    $lca_{1s} \leftarrow LCA(M_s.r_1, S)$ 
17   if  $lca_{1s} >_{dpst} lca_{12}$  then
18     if  $PDF(S) = \emptyset$  then
19        $M_s.r_1 \leftarrow S$ 
20       PruneStepList ( $S, lca_{12}, lca_{1s}, M_s.fts, M_s.lca\_map$ )
21     else AddStepToList ( $S, M_s.fts$ )
22   end
23 end

```

---

set of  $S$  is empty. If the PDF set of  $S$  is non-empty, then  $S$  is added to the list of steps,  $M_s.fts$ , using the function *AddStepToList*.

If the current step  $S$  that reads  $M$  is not in parallel with both  $M_s.r_1$  and  $M_s.r_2$ , then these two readers are set to null. If the PDF set of  $S$  is empty, then  $M_s.r_1$  is updated to  $S$ . If not,  $S$  is added to the list of steps,  $M_s.fts$ , using the function *AddStepToList*.

If the current step  $S$  is in parallel with both  $M_s.r_1$  and  $M_s.r_2$ , then the algorithm checks if  $S$  is outside the subtree under  $LCA(M_s.r_1, M_s.r_2)$ . If  $S$  is inside the subtree under  $LCA(M_s.r_1, M_s.r_2)$ , then  $S$  can be discarded because it is covered by the two stored readers. If  $S$  is outside the subtree under  $LCA(M_s.r_1, M_s.r_2)$ , then the algorithm checks if PDF set of  $S$  is empty. If  $PDF(S)$  is empty,  $M_s.r_1$  is set to  $S$ , thereby pushing the LCA higher up the DPST and covering a bigger subtree using the pair of readers. Since the pair of readers cover a bigger subtree now, the algorithm prunes the list of steps  $M_s.fts$  to remove those steps that now exist within the covered subtree. This is done by the function *PruneStepList*. Specifically, the function looks at the nodes along the path from the old LCA to the new LCA. For each of these nodes, it looks for an entry in the map  $M_s.lca\_map$ , that maps these nodes to the set of steps which have them as their LCA with the existing readers. All these steps can be removed from the list  $M_s.fts$ , since they are now covered by the subtree under the new LCA. If  $PDF(S)$  is non-empty, then  $S$  is added to the list of steps,  $M_s.fts$ , using the function *AddStepToList*. The details of the functions *AddStepToList* and *PruneStepList* are not shown here.

### 4.3.5 Soundness and Precision

The extended SPD3 algorithm to support `futures` is sound and precise for a given input. The extra work done to extend SPD3 to support `futures` is to guarantee soundness and precision. Suppose, the shadow memory does not store the list of steps  $M_s.fts$  and the map  $M_s.lca\_map$  and the algorithm does not do the extra work associated with these fields. The algorithm would still be precise for programs with `futures` but it will not be sound. In other words, the algorithm may miss data races that occur during some execution of the program for the given input.

### 4.3.6 Space Overhead

The space overhead of the modified DPST has already been discussed earlier. The space overhead for every memory location was  $O(1)$  with just `async` and `finish`. In the presence of `futures`, it is no longer constant, because of the presence of the list  $fts$  in every shadow memory location. In the worst case, the size of this list could be equal to the number of `future` instances in the program. This is because every `future` instance could have a unique step reporting to it and hence, every such step need to be stored in  $fts$ . Also, the size of the map  $lca\_map$  could be as large as the height of the DPST, since we need to store the mapping from every node higher than the current LCA, in the worst case. The total size of the sets that these nodes are mapped on to is the same as the size of  $fts$ .

The worst case space overhead for every memory location is  $O(ft + h)$ , where  $ft$  is the number of `future` instances in the program and  $h$  is the height of the DPST. Though the worst case space overhead is high, in practice, if the program has a good mix of `asyncs`, `finishes`, and `futures`, then the size of the  $fts$  list is not expected to grow large.

### 4.3.7 Time Overhead

The time overhead of the SPD3 algorithm increases in the presence of `future`s by a factor of  $ft$ , where  $ft$  refers to the number of `future` instances in the program. This is because every write to a memory location could potentially result in checking if the new step conflicts with any of the steps in the list  $fts$ , whose worst case size is  $ft$ . Also, every read of a memory location could potentially result in updating the list  $fts$ , which in turn requires checking if any step in  $fts$  has the same PDF set as the new step. The pruning of the list  $fts$  during read operations would involve a constant time look up in the map for every node in the path from the old LCA to the new LCA, and deleting the corresponding steps from the list  $fts$ .

In the worst case, the time taken for every memory operation is  $O(ft * L * g)$ , where  $ft$  is the number of `future` instances in the program,  $L$  is the length of the longer path in the LCA computation, and  $g$  is the number of `get` instances in the program. This is because every DMHP computation will take  $O(L * g)$ , assuming we use the DPST with pointers from `future` nodes to their corresponding `get` nodes.

## Chapter 5

### Correctness Proofs

We presented the ESP-bags algorithm and the SPD3 algorithm for data race detection in HJ programs. While the ESP-bags algorithm is a sequential algorithm, the SPD3 algorithm is parallel. In this chapter, we prove the correctness of both the algorithms for `async` and `finish` constructs in HJ.

#### 5.1 ESP-bags for Async-Finish

In this section, we prove the correctness of the basic ESP-bags algorithm that detects data races in programs with `async` and `finish` constructs. First, we define the may-happen-in-parallel relation in terms of the Computation Graph of an HJ program execution. We then discuss about the use of Dynamic Program Structure Tree (DPST) in proving the correctness of ESP-bags. Note that the DPST abstraction is used to establish the proof of correctness and is not actually constructed by the ESP-bags algorithm. Then, we prove a couple of important properties of the DPST that are necessary to establish the correctness of ESP-bags. Using the invariants on the DPST, we establish a relation between may-happen-in-parallel and the contents of the P-bag. Finally, we prove that the ESP-bags algorithm detects a data race in a program for a given input if and only if a data race exists; i.e., the algorithm is precise and sound for the *given input*.

Recall the definition of the Computation Graph of an HJ program execution from

Section 2.2. Figure 2.1 shows an HJ program with finish and async constructs and the computation graph corresponding to its execution in which the for loop in line 5 is executed only once.

**Definition 5.1.1.** Two statement instances  $s_1$  and  $s_2$  in a schedule  $\delta$  of a program may-happen-in-parallel, written as  $\text{DMHP}(s_1, s_2) = \text{true}$ , if and only if there is no path from  $s_1$  to  $s_2$  and from  $s_2$  to  $s_1$  in the computation graph of  $\delta$ .  $\square$

When two statement instances  $s_1$  and  $s_2$  in a schedule  $\delta$  of a program  $P$  for an input  $\psi$  may-happen-in-parallel,  $\text{DMHP}(s_1, s_2) = \text{true}$ , it means that there is a possible schedule of  $P$  with input  $\psi$  in which  $s_1$  and  $s_2$  execute in parallel.

We use the Dynamic Program Structure Tree (DPST) defined in Section 4.1.1 as part of the SPD3 algorithm, to prove the correctness of ESP-bags. Note that the DPST used in SPD3 groups statements into steps and uses step as the basic unit of computation. To make the proofs simpler here, we assume that the statements are not grouped into steps and there is a leaf node in the DPST for every statement instance. Note that all the properties of the DPST hold with this change.

**Theorem 5.1.1.** *Every data-race-free HJ program with finish and async constructs has a unique DPST that corresponds to all possible executions for a given input.*

*Proof.* Let us consider an HJ program  $P$  with finish and async constructs that contains no data races. The immediately enclosing finish for every statement in  $P$  is the same across all possible executions of  $P$  for a given input  $\psi$ . Also every statement in  $P$  belongs to the same task across all possible executions of  $P$  for input  $\psi$ . Hence, in every DPST of  $P$  that corresponds to different executions of  $P$  for an input  $\psi$ , the parent-child relationship is unique between nodes corresponding to all the instances

of finish, async, and statements in  $P$ . In other words, if node  $\alpha$  is the parent of node  $\beta$  in a DPST of  $P$ , then  $\alpha$  is the parent of  $\beta$  in every DPST of  $P$  for an input  $\psi$ .

The only other source of non-determinism could be in the order of edges from an internal node to its children. By definition of the DPST, all the edges from every internal node to its children are arranged according to the program order. Hence, there is a unique DPST for every HJ program with finish and async constructs for a given input.  $\square$

**Corollary 5.1.2.** *Every data-race-free HJ program with finish and async constructs has a unique Computation Graph that corresponds to all possible executions for a given input.*

**Theorem 5.1.3.** *The sequential depth-first execution of an HJ program explores the DPST of the program corresponding to this execution in depth-first order from left to right.*

*Proof.* By definition of DPST, the edges from every internal node to its children are ordered according to the program order of the corresponding statements. The sequential depth-first execution of an HJ program will execute the statements in the program order, which corresponds to the left to right depth-first order of the nodes in its DPST.  $\square$

**Theorem 5.1.4.** *Let  $\Gamma$  be the DPST corresponding to the sequential depth-first execution of an HJ program  $P$  with an input  $\psi$ . Let statement instance  $s_1$  be to the left of statement instance  $s_2$  ( $s_1 \neq s_2$ ) in  $\Gamma$ . During the sequential depth-first execution of  $P$  with input  $\psi$  in the ESP-bags algorithm, when  $s_2$  is being executed, the ID of the task  $\tau$  that executes  $s_1$  will be in a  $P$ -bag if and only if  $s_1$  may-happen-in-parallel with  $s_2$ .*

*Proof.* Let  $\Gamma$  be the DPST of  $P$  for input  $\psi$ . Let  $\text{LCA}(s_1, s_2) = \lambda$ . Since  $s_1$  and  $s_2$  are leaf nodes in the DPST,  $\lambda \neq s_1$  and  $\lambda \neq s_2$ .

*if:* Let us assume  $s_1$  may-happen-in-parallel with  $s_2$ . During the sequential depth-first execution of  $P$ ,  $s_1$  will be executed before  $s_2$  because of the assumption that  $s_1$  is to the left of  $s_2$ . Let  $A_1$  denote the DPST ancestor of  $s_1$  that is the child of  $\lambda$ . We know from Theorem 4.1.1 that  $A_1$  must be an async node. According to the rules of the ESP-bags algorithm, when the sequential depth-first execution returns from an async to its parent, the contents of the S and P bags of the async are emptied into the P bag of the parent. These contents stay in the P bag of the parent until the execution reaches the end of the parent. In our case, when the sequential depth-first execution of ESP-bags returns from  $A_1$ , the ID of the task  $\tau$  that owns  $s_1$  will be put in the P-bag of  $\lambda$ , which is the parent of  $A_1$  in  $\Gamma$ . The ID of  $\tau$  will stay in the P-bag of  $\lambda$  until the execution completes the execution of the subtree under  $\lambda$ . By definition of  $\lambda$  and  $A_1$  we know that  $s_2$  is in a subtree whose root is a peer of  $A_1$  and is to the right of  $A_1$ . Hence when  $s_2$  is executed, the ID of  $\tau$  will be in a P-bag.

*only if:* Let us assume that the ID of the task  $\tau$  that owns  $s_1$  is in a P-bag when  $s_2$  is executed under the sequential depth-first execution of the ESP-bags algorithm. Let  $A_1$  denote the DPST ancestor of  $s_1$  that is the child of  $\lambda$ .

Case 1:  $A_1$  is a finish node. In this case  $\tau$  will be in a S bag when  $s_2$  is executed, according to the rules of the ESP-bags algorithm.

Case 2:  $A_1$  is the node  $s_1$ . Again in this case  $\tau$  will be in a S bag when  $s_2$  is executed, according to the rules of the ESP-bags algorithm.

Hence  $A_1$  can neither be a finish node nor the node  $s_1$ .  $A_1$  must be an async node. Following from Theorem 4.1.1,  $s_1$  may-happen-in-parallel with  $s_2$ .  $\square$



**Theorem 5.1.5** (Precision and Soundness). *The ESP-bags algorithm detects a data race in an HJ program for a given input if and only if a data race exists.*

*Proof.* Let us consider an HJ program  $P$  that is executed with an input  $\psi$ . Let  $\Gamma$  denote the DPST corresponding to the sequential depth-first execution of  $P$  with input  $\psi$ .

*if:* Let us assume that there is a data race in some schedule of  $P$  with input  $\psi$ . There are two statements,  $s_1$  and  $s_2$ , that may-happen-in-parallel, both accessing the same memory location  $L$ , and one of those is a write. Without loss of generality, let us assume that  $s_1$  executes before  $s_2$  during ESP-bags's sequential depth-first execution of  $P$ . Thus  $s_1$  will be to the left of  $s_2$  in  $\Gamma$ . From Theorem 5.1.4 it follows that when  $s_2$  is executed, the task  $\tau$  that owns  $s_1$  will be in a P-bag.

Case 1:  $s_2$  contains a read of  $L$ . In this case,  $s_1$  will contain a write to  $L$ . When  $s_2$  is executed during the sequential depth-first execution, the ESP-bags algorithm checks if the previous writer of  $L$  is in a P-bag. In this case, since  $\tau$  is in a P-bag, the algorithm signals a data race.

Case 2:  $s_2$  contains a write of  $L$ . Now  $s_1$  may contain either a read or a write to  $L$ . When  $s_2$  is executed during the sequential depth-first execution, the ESP-bags algorithm checks if the previous reader or writer of  $L$  is in a P-bag. In this case, since  $\tau$  is in a P-bag, the algorithm signals a data race.

*only if:* Let us assume that the ESP-bags algorithm detects a data race in  $P$  with input  $\psi$ . According to the rules of the ESP-bags algorithm, a data race will be signaled only in two cases:

Case 1: On the read of a memory location  $L$  in a statement  $s_2$ , the previous writer of  $L$  (corresponding to a write in a statement  $s_1$ ), say  $\tau$ , is in a P-bag. It follows

from Theorem 5.1.4 that  $s_1$  may-happen-in-parallel with  $s_2$ . Hence, there is a data race in some execution of  $P$  with input  $\psi$ .

Case 2: On the write of a memory location  $L$ , the previous reader or writer of  $L$  (corresponding to a read or a write in a statement  $s_1$ ), say  $\tau$ , is in a P-bag. It follows from Theorem 5.1.4 that  $s_1$  may-happen-in-parallel with  $s_2$ . Hence, there is a data race in some execution of  $P$  with input  $\psi$ .  $\square$

In summary, if the ESP-bags algorithm reports a data race, then it is a true data race and if the ESP-bags algorithm does not report a data race, then no execution of that program for that input will result in a data race.

## 5.2 SPD3 for Async-Finish

In this section, we prove the correctness of the basic SPD3 algorithm that detects data races in HJ programs with `async` and `finish` constructs. First, we define a few terms and relations that will be used extensively in our proof. Then, we present a few lemmas along with their proofs. These lemmas form the basis of our correctness proof for SPD3. We then present three theorems that prove that the SPD3 algorithm is sound, i.e., if the algorithm does not report a data race, then no execution of that program for that input will have a data race. Then, we present three theorems that prove the precision of SPD3, i.e., if the algorithm reports a data race, then it is a true data race. Finally, we present the theorem that uses these earlier results to show that SPD3 is sound and precise for a given input.

**Definition 5.2.1.** A parallel schedule of a program  $P$  with a given input is a mapping from every dynamic instruction instance in  $P$  to a unique pair  $(w, t)$ , where  $w$  is the worker that executes the instruction at time  $t$ .  $\square$

Since, the SPD3 algorithm works with steps as the basic unit of computation, the parallel schedule can also be considered to be a mapping from every step to a unique triple  $(w, t_1, t_2)$ , where the worker  $w$  executes the step from time  $t_1$  to  $t_2$ .

**Definition 5.2.2.** For two memory operations,  $O_1$  and  $O_2$ , on the same location, operation  $O_1$  scheduled at  $(w, t_1)$  is said to execute before operation  $O_2$  scheduled at  $(w', t_2)$  if  $t_1 < t_2$ . We assume that no two memory operations on the same location can execute at the same time.  $\square$

Note that when memory operation  $O_1$  executes before memory operation  $O_2$ , we also refer to this as  $O_2$  executing after  $O_1$ .

**Definition 5.2.3.** A step  $S_1$  scheduled at  $(w, t_1, t_2)$  is said to execute before another step  $S_2$  scheduled at  $(w', t_3, t_4)$  if  $t_2 < t_3$ .  $\square$

**Definition 5.2.4.** For three memory operations,  $O_1$ ,  $O_2$ , and  $O_3$ , on the same location, operation  $O_1$  is said to execute between operations  $O_2$  and  $O_3$  if  $O_2$  executes before  $O_1$  and  $O_1$  executes before  $O_3$ .  $\square$

**Definition 5.2.5.** A read-write data race on a memory location  $M$  is a data race caused by a read  $R$  of  $M$  executing before the conflicting write  $W$  to  $M$ , where  $\text{DMHP}(R, W) = \text{true}$ .  $\square$

**Definition 5.2.6.** A write-read data race on a memory location  $M$  is a data race caused by a write  $W$  to  $M$  executing before the conflicting read  $R$  of  $M$ , where  $\text{DMHP}(W, R) = \text{true}$ .  $\square$

**Definition 5.2.7.** A write-write data race on a memory location  $M$  is a data race caused by two conflicting writes,  $W_1$  and  $W_2$ , to  $M$ , where  $\text{DMHP}(W_1, W_2) = \text{true}$ .  $\square$

**Definition 5.2.8.** Consider two nodes,  $n_1$  and  $n_2$ ,  $n_1 \neq n_2$ , in a DPST  $\Gamma$ , such that neither node is an ancestor of the other in  $\Gamma$ . Then, happens-before-defining-node of  $n_1$  and  $n_2$ , denoted by  $\text{HBDN}(n_1, n_2)$ , is defined to be the node  $\eta$ , such that:

- if  $n_1$  is to the left of  $n_2$  in  $\Gamma$ , then  $\eta$  is the ancestor of  $n_1$  which is the child of  $\text{LCA}(n_1, n_2)$ . Note that, if  $n_1$  is a child of  $\text{LCA}(n_1, n_2)$ , then  $\eta$  is the node  $n_1$ .
- if  $n_2$  is to the left of  $n_1$  in  $\Gamma$ , then  $\eta$  is the ancestor of  $n_2$  which is the child of  $\text{LCA}(n_1, n_2)$ . Note that, if  $n_2$  is a child of  $\text{LCA}(n_1, n_2)$ , then  $\eta$  is the node  $n_2$ . □

For any two steps,  $S_1$  and  $S_2$ , in a DPST, it follows from Theorem 4.1.1 that:

$$\text{DMHP}(S_1, S_2) = \text{true} \Leftrightarrow \text{HBDN}(S_1, S_2) \text{ is an async.}$$

The motivation behind the name *happens-before-defining-node* is that the node  $\text{HBDN}(S_1, S_2)$  defines the *happens-before* relation between  $S_1$  and  $S_2$ , i.e.  $\text{HBDN}(S_1, S_2)$  defines whether  $S_1$  and  $S_2$  may execute in parallel.

**Lemma 5.2.1.** Consider two steps,  $S_1$  and  $S_2$ ,  $S_1 \neq S_2$ , in a DPST  $\Gamma$ . Let  $\text{LCA}(S_1, S_2) = \lambda$ . Every node  $S_k$  which may execute in parallel with one of  $S_1, S_2$  and not the other, will be in the subtree under  $\lambda$ .

*Proof.* For any node  $S_k$  that is outside the subtree under  $\lambda$ ,  $\text{HBDN}(S_1, S_k) = \text{HBDN}(S_2, S_k)$ . Hence,  $\text{DMHP}(S_1, S_k) = \text{DMHP}(S_2, S_k)$ . So, it cannot be the case that  $S_k$  executes in parallel with one of  $S_1, S_2$  and not the other. Thus,  $S_k$  has to be in the subtree under  $\lambda$ . □

**Lemma 5.2.2.** If a step  $S_1$  executes before a step  $S_2$  in an execution  $\delta$  of a program  $P$  with input  $\psi$  and  $\text{DMHP}(S_1, S_2) = \text{false}$ , then  $S_1$  will execute before  $S_2$  in all executions of  $P$  with  $\psi$ .

*Proof.* Consider the DPST  $\Gamma$  of the program  $P$  with input  $\psi$ . Let  $\text{HBDN}(S_1, S_2) = \eta$ . Since  $\text{DMHP}(S_1, S_2) = \text{false}$ ,  $\eta$  is not an async. Since  $S_1$  executes before  $S_2$  in the execution  $\delta$ ,  $S_1$  has to be to the left of  $S_2$  in  $\Gamma$ . We know that every execution of  $P$  with  $\psi$  will result in the same DPST  $\Gamma$ . Since  $\text{HBDN}(S_1, S_2), \eta$ , is not an async,  $S_1$  has to execute before  $S_2$  in all executions of  $P$  with input  $\psi$ .  $\square$

**Lemma 5.2.3.** *Consider two steps,  $S_1$  and  $S_2$ ,  $S_1 \neq S_2$ , in a DPST  $\Gamma$  such that  $\text{DMHP}(S_1, S_2) = \text{false}$ . Then, for any step,  $S_k$ , in  $\Gamma$  that may execute after  $S_1$  and  $S_2$ :*

$$\text{DMHP}(S_1, S_k) = \text{true} \Rightarrow \text{DMHP}(S_2, S_k) = \text{true}$$

*Proof.* Let  $\text{LCA}(S_1, S_2) = \lambda$ . Let  $\text{HBDN}(S_1, S_2) = \eta$ . Since  $\text{DMHP}(S_1, S_2) = \text{false}$ ,  $\eta$  is not an async. Without loss of generality, let us assume that  $S_1$  executes first followed by  $S_2$ . Since  $\text{DMHP}(S_1, S_2) = \text{false}$ , it follows from Lemma 5.2.2 that  $S_1$  will execute before  $S_2$  in all executions of the given program with the given input. Hence,  $\eta$  is an ancestor of  $S_1$  that is a child of  $\lambda$ .

Case 1:  $S_k$  is outside the subtree under  $\lambda$ .

$$\begin{aligned} \text{HBDN}(S_1, S_k) &= \text{HBDN}(S_2, S_k). \text{ Hence } \text{DMHP}(S_1, S_k) = \text{DMHP}(S_2, S_k), \text{ i.e.,} \\ \text{DMHP}(S_1, S_k) &= \text{true} \Rightarrow \text{DMHP}(S_2, S_k) = \text{true} \end{aligned}$$

Case 2:  $S_k$  is inside the subtree under  $\lambda$  and to the left of the subtree under  $\eta$ .

$$\begin{aligned} \text{Let } \text{HBDN}(S_1, S_k) &= \eta'. \text{ If } \text{DMHP}(S_1, S_k) = \text{true}, \eta' \text{ is an async. Since } S_k \text{ is} \\ &\text{to the left of the subtree under } \eta \text{ and } S_2 \text{ is to the right of the subtree under } \eta, \\ \text{HBDN}(S_2, S_k) &\text{ is } \eta'. \text{ Since } \eta' \text{ is an async, } \text{DMHP}(S_2, S_k) = \text{true}. \end{aligned}$$

Case 3:  $S_k$  is inside the subtree under  $\lambda$  and to the right of the subtree under  $\eta$ .

$$\text{HBDN}(S_1, S_k) = \eta. \text{ Since } \eta \text{ is not an async, } \text{DMHP}(S_1, S_k) = \text{false}. \text{ In this}$$

case,  $\text{DMHP}(S_1, S_k)$  can never be true.

Case 4:  $S_k$  is inside the subtree under  $\eta$ .

Since  $\eta$  is not an async and  $S_2$  is to the right of the subtree under  $\eta$ ,  $S_k$  can never execute after  $S_2$ . Hence, this case is not possible.  $\square$

**Lemma 5.2.4.** *Consider two steps,  $S_1$  and  $S_2$ ,  $S_1 \neq S_2$ , in a DPST  $\Gamma$ , such that  $\text{DMHP}(S_1, S_2) = \text{true}$ . Let  $\text{LCA}(S_1, S_2) = \lambda$ . Let  $S_3$  denote a step in the subtree under  $\lambda$ . Then, for any step,  $S_k$ , in  $\Gamma$  that may execute after  $S_1$  and  $S_2$ :*

$$\begin{aligned} \text{DMHP}(S_k, S_3) = \text{true} &\Rightarrow \text{DMHP}(S_k, S_1) = \text{true} \parallel \\ &\text{DMHP}(S_k, S_2) = \text{true} \end{aligned}$$

*Proof.* Without loss of generality, let us assume that  $S_1$  is to the left of  $S_2$  in  $\Gamma$ . Let  $\text{HBDN}(S_1, S_2) = \eta$ . Since  $\text{DMHP}(S_1, S_2) = \text{true}$ ,  $\eta$  is an async.

Case 1:  $S_k$  is outside the subtree under  $\lambda$ .

Since  $S_k$  is outside the subtree under  $\lambda$ ,  $\text{HBDN}(S_1, S_k) = \text{HBDN}(S_2, S_k) = \text{HBDN}(S_3, S_k)$ . Hence,  $\text{DMHP}(S_3, S_k) = \text{DMHP}(S_1, S_k) = \text{DMHP}(S_2, S_k)$ .

Case 2:  $S_k$  is within the subtree under  $\lambda$ .

Case 2a:  $S_k$  is within the subtree under  $\eta$ .

$\text{HBDN}(S_2, S_k) = \eta$  and  $\eta$  is an async. So,  $\text{DMHP}(S_2, S_k) = \text{true}$ .

Case 2b:  $S_k$  is outside the subtree under  $\eta$  and to the left of  $S_1$ .

Let  $\text{HBDN}(S_1, S_k) = \zeta$ . If  $\zeta$  is a finish or  $\zeta = S_k$ ,  $S_k$  can never execute after  $S_1$  and  $S_2$ , i.e., no execution of the given program will have  $S_k$  executing after  $S_1$  and  $S_2$ . Hence,  $\zeta$  must be an async and  $\text{DMHP}(S_k, S_1) = \text{true}$ .

Case 2c:  $S_k$  is outside the subtree under  $\eta$  and to the right of  $S_1$ .

$\text{HBDN}(S_1, S_k) = \eta$  and  $\eta$  is an async. So,  $\text{DMHP}(S_1, S_k) = \text{true}$ .  $\square$

**Theorem 5.2.1.** *If Algorithms 3 and 4 do not report any data race in some execution of a program  $P$  with input  $\psi$ , then no execution of  $P$  with  $\psi$  will have a write-write data race on any memory location  $M$ .*

*Proof.* Consider an execution  $\delta$  of a program  $P$  with input  $\psi$  in which Algorithms 3 and 4 do not report any data race.

Suppose that a write-write data race,  $\chi$ , occurs on a memory location  $M$  in some execution  $\delta'$  of  $P$  with  $\psi$ . Let  $\mathcal{W}_1$  and  $\mathcal{W}_2$  denote the two steps that write to  $M$  resulting in the data race in  $\delta'$ , i.e,  $\text{DMHP}(\mathcal{W}_1, \mathcal{W}_2) = \text{true}$ . Note that the execution  $\delta'$  does not have any data race until  $\chi$  occurs. Without loss of generality, let us assume  $\mathcal{W}_1$  writes to  $M$  first and  $\mathcal{W}_2$  writes later in  $\delta$ .

Case 1: There are no writes to  $M$  between  $\mathcal{W}_1$  and  $\mathcal{W}_2$  in  $\delta$ .

When  $\mathcal{W}_1$  occurs in  $\delta'$ , Algorithm 3 checks if any of the previous readers and writers of  $M$  (in the three fields of  $M_s$ ) can execute in parallel with  $\mathcal{W}_1$ . Since  $\chi$  is the first data race to occur in  $\delta'$ , they can never execute in parallel with  $\mathcal{W}_1$ . Since the DPST is same across all executions of  $P$  with  $\psi$ , this applies to  $\delta$  as well. Also, it follows from Lemma 5.2.2 that the previous readers and writers of  $M$  will execute before  $\mathcal{W}_1$  in  $\delta$ . Hence, when  $\mathcal{W}_1$  occurs in  $\delta$ , Algorithm 3 sets  $M_s.w$  to  $\mathcal{W}_1$ . Then, when  $\mathcal{W}_2$  occurs in  $\delta$ , Algorithm 3 will see that  $\mathcal{W}_2$  can execute in parallel with  $\mathcal{W}_1$  and signal a write-write race between them. This is contradicting our assumption that both our algorithms do not report any data race in  $\delta$ .

Case 2: There are writes to  $M$  by steps  $\mathcal{W}_i \cdots \mathcal{W}_j$  between  $\mathcal{W}_1$  and  $\mathcal{W}_2$  in  $\delta$ .

The writes to  $M$  happen in this order in  $\delta'$ :  $\mathcal{W}_1, \mathcal{W}_i \cdots \mathcal{W}_j, \mathcal{W}_2$ . Since the data race  $\chi$  between  $\mathcal{W}_1$  and  $\mathcal{W}_2$  is the first data race in  $\delta'$ , there must have been no races among the writes  $\mathcal{W}_1, \mathcal{W}_i \cdots \mathcal{W}_j$ . In other words,  $\text{DMHP}(\mathcal{W}_1, \mathcal{W}_i) = \text{false}, \dots, \text{DMHP}(\mathcal{W}_{j-1}, \mathcal{W}_j) = \text{false}$ . Since the DPST is same across all executions of  $P$  with  $\psi$ , this applies to  $\delta$  as well. Also, it follows from Lemma 5.2.2 that these writes to  $M$  occur in the same order in  $\delta$  as well. Hence in  $\delta$ , Algorithm 3 would set  $M_s.w$  field to  $\mathcal{W}_1, \mathcal{W}_i \cdots \mathcal{W}_j$  in order. When  $\mathcal{W}_2$  occurs in  $\delta$ ,  $M_s.w$  will contain  $\mathcal{W}_j$ . It follows from Lemma 5.2.3 that if  $\text{DMHP}(\mathcal{W}_1, \mathcal{W}_2) = \text{true}$ , then  $\text{DMHP}(\mathcal{W}_j, \mathcal{W}_2) = \text{true}$ . Hence, when  $\mathcal{W}_2$  occurs in  $\delta$ , Algorithm 3 will see that  $\mathcal{W}_2$  can execute in parallel with  $\mathcal{W}_j$  and signal a write-write race between them. This is contradicting our assumption that both our algorithms do not report any data race in  $\delta$ .  $\square$

**Theorem 5.2.2.** *If Algorithms 3 and 4 do not report any data race in some execution of a program  $P$  with input  $\psi$ , then no execution of  $P$  with  $\psi$  will have a read-write data race on any memory location  $M$ .*

*Proof.* Consider an execution  $\delta$  of a program  $P$  with input  $\psi$  in which Algorithms 3 and 4 do not report any data race.

Suppose that a read-write data race,  $\chi$ , occurs on a memory location  $M$  in some execution  $\delta'$  of  $P$  with  $\psi$ . Let  $\mathcal{R}_1$  and  $\mathcal{W}_1$  denote the steps that read and write  $M$  resulting in the data race in  $\delta'$ , i.e,  $\text{DMHP}(\mathcal{R}_1, \mathcal{W}_1) = \text{true}$ . Note that the execution  $\delta'$  does not have any data race until  $\chi$  occurs.

Case 1:  $\mathcal{R}_1$  executes before  $\mathcal{W}_1$  in  $\delta$ .

When  $\mathcal{R}_1$  occurs in  $\delta$ , Algorithm 4 either updates one of the readers of  $M_s$  with  $\mathcal{R}_1$  or chooses not to update the readers because  $\mathcal{R}_1$  can execute in parallel with



$M_s.r_1$  and  $M_s.r_2$  and is also within the subtree of  $\text{LCA}(M_s.r_1, M_s.r_2)$  ( $= \lambda$ ).

Case 1.a: There are no reads of  $M$  between  $\mathcal{R}_1$  and  $\mathcal{W}_1$  in  $\delta$ .

If  $M_s.r_1$  or  $M_s.r_2$  contains  $\mathcal{R}_1$ , when  $\mathcal{W}_1$  occurs in  $\delta$ , Algorithm 3 will find that  $\mathcal{W}_1$  can execute in parallel with  $\mathcal{R}_1$  and report a read-write data race. This contradicts our assumption that our algorithm does not report a data race in  $\delta$ .

If  $M_s.r_1$  and  $M_s.r_2$  does not contain  $\mathcal{R}_1$ , when  $\mathcal{W}_1$  occurs in  $\delta$ , Algorithm 3 will find that  $\mathcal{W}_1$  can execute in parallel with at least one of  $M_s.r_1, M_s.r_2$ . (This follows from Lemma 5.2.4 and the fact that  $\text{DMHP}(\mathcal{R}_1, \mathcal{W}_1) = \text{true}$ .) Again, Algorithm 3 will report a read-write data race which contradicts our assumption.

Case 1.b: There are reads of  $M$  by steps  $\mathcal{R}_i \cdots \mathcal{R}_j$  between  $\mathcal{R}_1$  and  $\mathcal{W}_1$  in  $\delta$ .

Case 1.b.1:  $\forall \mathcal{R}_k$  in  $[\mathcal{R}_i \cdots \mathcal{R}_j]$   $\text{DMHP}(\mathcal{R}_1, \mathcal{R}_k) = \text{true}$ .

After  $\mathcal{R}_j$  completes in  $\delta$ ,  $M_s.r_1$  and  $M_s.r_2$  will be updated such that  $\mathcal{R}_j$  is in the subtree under  $\text{LCA}(M_s.r_1, M_s.r_2)$  ( $= \lambda$ ). Since  $\text{DMHP}(\mathcal{R}_1, \mathcal{R}_j) = \text{true}$ ,  $\mathcal{R}_1$  must also be in the subtree under  $\lambda$ . From Lemma 5.2.4 it follows that either  $\text{DMHP}(M_s.r_1, \mathcal{W}_1) = \text{true}$  or  $\text{DMHP}(M_s.r_2, \mathcal{W}_1) = \text{true}$ . Hence, Algorithm 3 should have reported a read-write data race in  $\delta$ . A contradiction.

Case 1.b.2:  $\exists \mathcal{R}_k$  in  $[\mathcal{R}_i \cdots \mathcal{R}_j]$  such that  $\text{DMHP}(\mathcal{R}_1, \mathcal{R}_k) = \text{false}$ .

Since  $\text{DMHP}(\mathcal{R}_1, \mathcal{W}_1) = \text{true}$ ,  $\text{DMHP}(\mathcal{R}_k, \mathcal{W}_1) = \text{true}$  from Lemma 5.2.3.

Hence, Algorithm 3 should have reported a read-write data race in  $\delta$ .

A contradiction.

Case 2:  $\mathcal{W}_1$  executes before  $\mathcal{R}_1$  in  $\delta$ .

Case 2.a: There are no writes to  $M$  between  $\mathcal{W}_1$  and  $\mathcal{R}_1$  in  $\delta$ .

When  $\mathcal{W}_1$  occurs in  $\delta$ , Algorithm 3 will update  $M_{s.w}$  to  $\mathcal{W}_1$ . When  $\mathcal{R}_1$  occurs in  $\delta$ , Algorithm 4 will see that  $\text{DMHP}(\mathcal{W}_1, \mathcal{R}_1) = \text{true}$  and should have reported a write-read data race in  $\delta$ . A contradiction.

Case 2.b: There are writes to  $M$  by steps  $\mathcal{W}_i \cdots \mathcal{W}_j$  between  $\mathcal{W}_1$  and  $\mathcal{R}_1$  in  $\delta$ .

If any  $\mathcal{W}_k$  in  $[\mathcal{W}_i \cdots \mathcal{W}_j]$  can execute in parallel with  $\mathcal{W}_1$ , i.e.  $\text{DMHP}(\mathcal{W}_1, \mathcal{W}_k) = \text{true}$ , Algorithm 3 should have reported a write-write data race in  $\delta$ . So,  $\forall \mathcal{W}_k$  in  $[\mathcal{W}_i \cdots \mathcal{W}_j]$   $\text{DMHP}(\mathcal{W}_1, \mathcal{W}_k) = \text{false}$ . Hence, when  $\mathcal{W}_j$  executes in  $\delta$ , Algorithm 3 will update  $M_{s.w}$  to  $\mathcal{W}_j$ . From Lemma 5.2.3,  $\text{DMHP}(\mathcal{W}_1, \mathcal{R}_1) = \text{true} \implies \text{DMHP}(\mathcal{W}_j, \mathcal{R}_1) = \text{true}$ . Hence, Algorithm 4 should have report a write-read data race in  $\delta$ . A contradiction.  $\square$

**Theorem 5.2.3.** *If Algorithms 3 and 4 do not report any data race in some execution of a program  $P$  with input  $\psi$ , then no execution of  $P$  with  $\psi$  will have a write-read data race on any memory location  $M$ .*

*Proof.* Consider an execution  $\delta$  of a program  $P$  with input  $\psi$  in which Algorithms 3 and 4 do not report any data race.

Suppose that a write-read data race,  $\chi$ , occurs on a memory location  $M$  in some execution  $\delta'$  of  $P$  with  $\psi$ . Let  $\mathcal{W}_1$  and  $\mathcal{R}_1$  denote the steps that write and read  $M$  resulting in the data race in  $\delta'$ , i.e,  $\text{DMHP}(\mathcal{W}_1, \mathcal{R}_1) = \text{true}$ . Note that the execution  $\delta'$  does not have any data race until  $\chi$  occurs.

Case 1:  $\mathcal{W}_1$  executes before  $\mathcal{R}_1$  in  $\delta$ .

Same as Case 2 in Theorem 5.2.2.

Case 2:  $\mathcal{R}_1$  executes before  $\mathcal{W}_1$  in  $\delta$ .

Same as Case 1 in Theorem 5.2.2.  $\square$

**Theorem 5.2.4.** *If Algorithm 3 reports a write-write race on a memory location  $M$  during an execution of a program  $P$  with input  $\psi$ , then there exists at least one execution of  $P$  with  $\psi$  in which this race exists.*

*Proof.* Since Algorithm 3 reports a write-write race on  $M$ , there must be two steps,  $\mathcal{W}_1$  and  $\mathcal{W}_2$ , that write  $M$  such that  $\text{DMHP}(\mathcal{W}_1, \mathcal{W}_2) = \text{true}$ . From the definition of DMHP, it follows that there is a schedule  $\delta$  of  $P$  with  $\psi$  in which  $\mathcal{W}_1$  and  $\mathcal{W}_2$  execute in parallel. Hence, the write-write data race exists in  $\delta$ .  $\square$

**Theorem 5.2.5.** *If Algorithm 3 reports a read-write race on a memory location  $M$  during an execution of a program  $P$  with input  $\psi$ , then there exists at least one execution of  $P$  with  $\psi$  in which this race exists.*

*Proof.* Since Algorithm 3 reports a read-write race on  $M$ , there must be two steps,  $\mathcal{R}_1$  and  $\mathcal{W}_1$ , that read and write  $M$  respectively such that  $\text{DMHP}(\mathcal{R}_1, \mathcal{W}_1) = \text{true}$ . From the definition of DMHP, it follows that there is a schedule  $\delta$  of  $P$  with  $\psi$  in which  $\mathcal{R}_1$  and  $\mathcal{W}_1$  execute in parallel. Hence, the read-write data race exists in  $\delta$ .  $\square$

**Theorem 5.2.6.** *If Algorithm 4 reports a write-read race on a memory location  $M$  during an execution of a program  $P$  with input  $\psi$ , then there exists at least one execution of  $P$  with  $\psi$  in which this race exists.*

*Proof.* Since Algorithm 4 reports a write-read race on  $M$ , there must be two steps,  $\mathcal{W}_1$  and  $\mathcal{R}_1$ , that write and read  $M$  respectively such that  $\text{DMHP}(\mathcal{W}_1, \mathcal{R}_1) = \text{true}$ . From the definition of DMHP, it follows that there is a schedule  $\delta$  of  $P$  with  $\psi$  in which  $\mathcal{W}_1$  and  $\mathcal{R}_1$  execute in parallel. Hence, the write-read data race exists in  $\delta$ .  $\square$

**Theorem 5.2.7.** *The SPD3 race detection algorithm, described by Algorithms 3 and 4, is sound and precise for a given input.*

*Proof.* From Theorems 5.2.1, 5.2.2, and 5.2.3 it follows that if our race detection algorithm does not report any data race in some execution of a program  $P$  with input  $\psi$ , then no execution of  $P$  with  $\psi$  will have a data race on any memory location  $M$ . Hence the algorithm is sound for a given input.

From Theorems 5.2.4, 5.2.5, and 5.2.6 it follows that if our race detection algorithm reports a data race for a program  $P$  with input  $\psi$ , then there exists at least one execution of  $P$  with  $\psi$  in which the race will occur. Hence the algorithm is precise for a given input. □

# Chapter 6

## Implementation

This chapter describes the implementation of the ESP-bags and the SPD3 algorithm for HJ programs with `async`, `finish`, and `isolated` constructs. First, we discuss the common details about the design and implementation of both data race detectors. We then discuss the implementation of disjoint-sets used in ESP-bags. Then, we describe the implementation of the Dynamic Program Structure Tree (DPST) that is used in the SPD3 algorithm. Finally, we present a technique to relax the atomicity requirement while updating the shadow memory in the SPD3 algorithm.

### 6.1 Design of Data Race Detectors

In this section, we describe the design and implementation details that are common to both ESP-bags and SPD3. We also explain the HJ system architecture with the focus on where and how the race detectors fit in.

The implementation of both ESP-bags and SPD3 involves two separate components:

- the race detection runtime
- the race detection instrumentor

The race detection runtime implements the actual race detection algorithm and defines interfaces for the HJ program to interact with the algorithm. The race detection

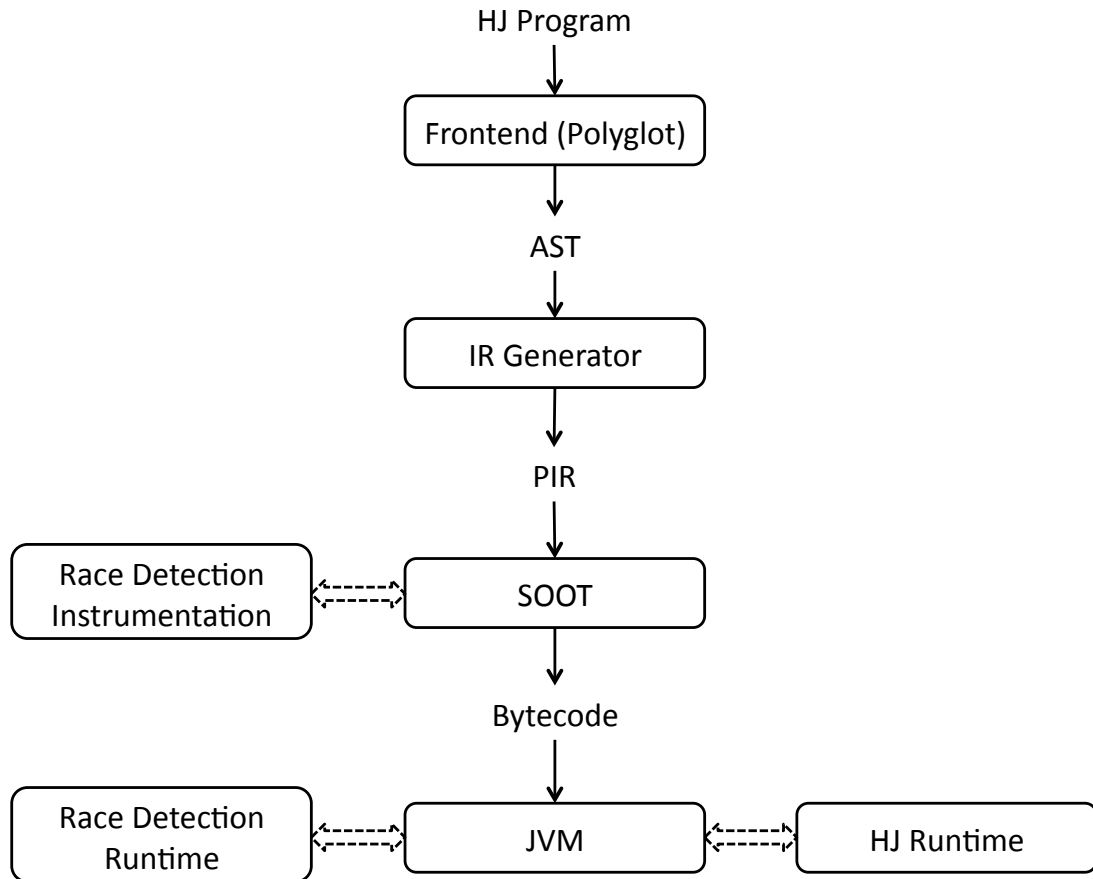


Figure 6.1 : HJ System Architecture with Race Detection

instrumentor adds the necessary instrumentations to the HJ program so that the algorithm can detect data races during the execution of the program.

The race detection runtime for both the algorithms were implemented as Java libraries. For every memory location, the shadow location was implemented by extending the `hj.lang.Object` class with shadow fields. In the case of arrays, *array views* [9, 10] were used as anchors for shadow arrays.

Now, we briefly describe the HJ system architecture and also the interaction of the two components of the data race detectors with the HJ system. The full description of the HJ system architecture can be found in [45]. Figure 6.1 shows the HJ

system architecture along with the race detector components. First, the HJ program is parsed and an Abstract Syntax Tree (AST) is generated by the frontend which is based on Polyglot [46]. The IR generator reads in the AST and generates a Parallel Intermediate Representation (PIR) for the given HJ program. The PIR is an intermediate representation that extends Soot's Jimple IR [47] with parallel constructs such as `async`, `finish`, `isolated`, and `future`. The HJ compiler includes an extended set of Soot passes to handle the PIR with the parallel constructs in HJ and to optimize the PIR of the given HJ program. It also includes a Soot pass that transforms the PIR to target a HJ runtime, say the work-stealing runtime. This pass would eliminate all the parallel constructs from the PIR and transform the code to use the HJ runtime for execution. The final step in the set of Soot passes is to generate bytecode from the transformed PIR. The bytecode then runs on a JVM like any other Java program. The execution of the bytecode on the JVM will use the HJ runtime that it was targeted to execute with.

The race detection runtime library, written in Java, is compiled to bytecode by the regular Java compiler, `javac`. The race detection instrumentor for both the algorithms are implemented as Soot transformation passes on the PIR. To enable data race detection, the race detection instrumentor is added to the list of passes that execute in Soot. This instrumentation pass adds the necessary calls to the race detection runtime library at `async`, `finish`, and `isolated` boundaries and also on reads and writes to shared memory locations. Now, the generated bytecode will contain these calls to the race detection runtime library. When this bytecode runs on the JVM, the race detection runtime monitors the accesses to shared memory locations in the program and detects data races when they occur.

## 6.2 Disjoint-set in ESP-bags

The implementation of the ESP-bags algorithm is mostly covered by the details in Section 6.1. The only part that remains is the maintaining of S and P bags of all the tasks during the execution of the program. The S and P bags are implemented as sets. Every task has two sets associated with it, one for the S bag and the other for the P bag of the task. As we discussed earlier in Section 3.2.1, since all the S and P bags of all the tasks are disjoint, the sets to store them are maintained as a disjoint-set forest [48]. The disjoint-set forest supports all the operations that are used in the ESP-bags algorithm, like MakeSet, Union, and Lookup, in an efficient manner.

The disjoint-set forest is implemented as follows. Every set is represented as a tree, where the nodes point to their parents. The root of the tree is the representative element of the set. A flag is attached to the root of every tree to indicate whether the set represents an S bag or a P bag. The MakeSet operation creates a tree with a single node because the set initially contains only one element. The Union operation on two sets combines the trees corresponding to the two sets into one by attaching the root of one tree to the other. The Lookup operation follows the parent pointer from the node to its root and returns the root because it is the representative element for that set. Note that the implementation also includes two optimizations to improve the asymptotic performance of the disjoint-set forest, namely *union by rank* and *path compression* [48].



### 6.3 DPST in SPD3

The SPD3 algorithm uses the Dynamic Program Structure Tree (DPST) to detect data races in HJ programs. In this section, we describe an implementation of the DPST and discuss how this implementation of DPST is amenable to garbage collection.

The DPST of the program being executed is built to maintain the parent-child relationship of asyncs, finishes and steps in the program. Every node in the DPST consists of the following 4 fields:

- *parent*: the DPST node which is the parent of this node.
- *depth*: an integer that stores the depth of this node. The root node of the DPST has depth 0. Every other node in the DPST has depth one greater than its parent. This field is immutable.
- *num\_children*: number of children of this node currently in the DPST. This field is initialized to 0 and incremented when child nodes are added.
- *seq\_no*: an integer that stores the ordering of this node among the children of its parent, i.e., among its siblings. Every node's children are ordered from left to right. They are assigned sequence numbers starting from 1 to indicate this order. This field is also immutable.

The use of depth for nodes in the DPST leads to a lowest common ancestor (LCA) algorithm with better complexity (than if we had not used this field). The use of sequence numbers to maintain the ordering of a node's children makes it easier to check for may happen in parallel given two steps in the program.

Note that all the fields of a node in the DPST can be initialized/updated without any synchronization: the *parent* field initialization is trivial because there are no competing writes to that field; the *depth* field of a node is written only on initialization, is never updated, and is read only after the node is created; the *num\_children* field is incremented whenever a child node is added, but for a given node, its children are always added sequentially in order from left to right; the *seq\_no* field is written only on initialization, is never updated, and is read only after the node is created.

### **Amenability to Garbage Collection**

Every node in a DPST points to its parent. There are no pointers from a node to its children or to its siblings in a DPST. The only reference to step nodes (leaf nodes) are from the shadow memory of the shared memory locations during program execution. This design ensures that the nodes in a DPST can be garbage collected as and when they become obsolete.

We now explain the process of garbage collection on a DPST. Suppose there are  $v$  shared memory locations during a program execution. There are  $v$  shadow memory locations, one for each memory location. Also, there is a pointer from each memory location to its shadow memory. The shadow memory stores the fields as described in the SPD3 algorithm. Each of these fields in the shadow memory points to a step in the DPST. Suppose there are  $u$  memory locations whose shadow memory point to a particular step  $S$ . During program execution, when all these  $u$  memory locations get garbage collected, the shadow memories corresponding to all these  $u$  memory locations can also be garbage collected. Now, since there are no references to the step  $S$ , the node corresponding to the step  $S$  can be garbage collected. When all the nodes within a subtree in the DPST get garbage collected, the root of the subtree

can be garbage collected.

Since DPST is a dynamic data structure, its size can grow arbitrarily large during program execution as there can be multiple instances for every statement in the program and each such instance gets a node in the DPST. Hence, the property of the DPST to allow for an implementation that is amenable to garbage collection is very important for using it in practical applications.

## 6.4 Relaxing the Atomicity Requirement in SPD3

A memory action for an access to a memory location  $M$  involves reading the fields of its shadow memory location  $M_s$ , computing the necessary *DMHP* information and checking appropriate predicates, and possibly updating the fields of  $M_s$ . Let us refer to these three stages as *read*, *compute*, and *update* of a memory action.

In our algorithm, every memory action on a shadow memory  $M_s$  has to execute atomically relative to other memory actions on  $M_s$ . When there are parallel reads to a memory location, this atomicity requirement effectively serializes the memory actions due to these reads. Hence this atomicity requirement induces a bottleneck in our algorithm when the program is executed on a large number of threads. Note that the atomicity requirement does not result in a bottleneck in the case of writes to a memory location because the memory actions due to writes have no contention in data race free programs. (In a data race free program, there is a happens-before ordering between a write and every other access to a memory location.)

We now present our implementation strategy to overcome this atomicity requirement without sacrificing the correctness of our algorithm. This implementation strategy is based on the solution to the reader-writer problem proposed by Leslie Lamport in [49]. Our implementation allows multiple memory actions on the same shadow

memory to proceed in parallel. This is done by adding two atomic integers to every shadow memory, i.e.,  $M_s$  contains the following two additional fields:

- *startVersion*: an atomic integer that denotes the version number of  $M_s$
- *endVersion*: an atomic integer that denotes the version number of  $M_s$ .

Both *startVersion* and *endVersion* are initialized to zero. Every time any of the fields  $M_s.w$ ,  $M_s.r_1$ , or  $M_s.r_2$  are updated,  $M_s.startVersion$  and  $M_s.endVersion$  are incremented by one. The following invariant is maintained on every shadow memory  $M_s$  during the execution of our algorithm: *any consistent snapshot of  $M_s$  will have the same version number in both *startVersion* and *endVersion**. Now, we show how the read, compute, and update stages of a memory action on  $M_s$  are performed. Note that these rules use a *CompareAndSet (CAS)*<sup>1</sup> primitive which is *atomic* relative to every operation on the same memory location.

#### Read:

1. Read the version number in  $M_s.startVersion$  into a local variable,  $X$ .
2. Read the fields  $M_s.w$ ,  $M_s.r_1$ , and  $M_s.r_2$  into local variables,  $W$ ,  $R_1$ , and  $R_2$ .
3. Perform a *fence* to ensure that all operations above are complete.
4. Read the version number in  $M_s.endVersion$  into a local variable,  $Y$ .
5. If  $X$  is not the same as  $Y$ , restart the *read* stage.

#### Compute:

---

<sup>1</sup>It is also referred to as *CompareAndSwap* in the literature.

1. Perform the computation on the local variables,  $W$ ,  $R_1$ , and  $R_2$ .

**Update:**

1. Do the following steps if an update to any of the fields  $M_s.w$ ,  $M_s.r_1$ , or  $M_s.r_2$  is necessary.
2. Perform a *CAS* on the version number in  $M_s.endVersion$  looking for the value  $X$  and updating it with an increment of one.
3. If the above *CAS* fails, restart the memory action from the beginning of the *read* stage.
4. Write to the required fields of  $M_s$ .
5. Write the incremented version number to  $M_s.startVersion$ .

When a memory action on  $M_s$  completes the *read* stage, the above rules ensure that a consistent snapshot of  $M_s$  was captured. This is because the *read* stage completes only when the same version number is seen in both  $M_s.startVersion$  and  $M_s.endVersion$ .

The *CAS* in the *update* stage of the memory action on  $M_s$  succeeds only when  $M_s.endVersion$  has the version number that was found in the *read* stage earlier. The *update* stage completes by writing to the reader and writer fields of  $M_s$  as necessary, followed by incrementing the version number in  $M_s.startVersion$ . When the *update* stage completes, both  $M_s.startVersion$  and  $M_s.endVersion$  will have the same version number and thus, the fields of  $M_s$  are retained in a consistent state.

The *CAS* in the *update* stage of a memory action  $\alpha$  on  $M_s$  also ensures that the fields of  $M_s$  are updated only if it has not already been updated by any memory action on  $M_s$ , since the *read* stage of  $\alpha$ . If this *CAS* fails, then there has been some

update to  $M_s$  since the *read* stage and hence, the computations are discarded and the memory action is restarted from the beginning of the *read* stage. Thus, the memory actions are guaranteed to be atomic relative to other memory actions on the same memory location.

The main advantage of this implementation is that it allows multiple memory actions on the same shadow memory  $M_s$  to proceed in parallel. But if more than one of them needs to update the fields of  $M_s$ , then only one of them is guaranteed to succeed while the others repeat the action. This is especially beneficial when there are multiple parallel accesses to  $M$  whose memory actions do not update the fields of  $M_s$ . In our algorithm, this occurs when there are reads by step  $S$  such that  $S$  is in the subtree rooted at  $LCA(M_s.r_1, M_s.r_2)$ . These cases occur frequently in practice thereby emphasizing the importance of relaxing the atomicity requirement.

Our algorithm is implemented in Java and we use the *AtomicInteger* from Java Concurrency Utilities for the version numbers. The *CAS* on *Atomic Integer* is guaranteed to execute atomically with respect to other operations on the same location. Also, the *CAS* acts as a barrier for the memory effects of the instructions on its either side, i.e., all the instructions above it are guaranteed to complete before it executes and no instructions below it will execute before it completes. This is the same as the memory effects of the *fence* that is used in the read stage. The read of an *AtomicInteger* has the memory effects of the read of a volatile in Java. Hence, it does not allow any instruction after it to execute until it completes. Similarly, the write to an *AtomicInteger* has the memory effects of the write to a volatile in Java. Hence, it does not execute until all the instructions before it complete.

## Chapter 7

### Static Optimizations for Data Race Detection

The race detection algorithms are implemented as a *Java* library. Recall that the algorithms require that actions be taken on every read and write to a shared memory location. It is during these actions that the algorithms check if the current task can race with the task recorded in the access history of the memory location. To test a given program for data races using our algorithms, we use a compiler transformation pass that instruments read and write operations on a heap location or an array in the program with appropriate calls to the library. A naïve way to perform this is to instrument every access to every shared memory location. But some of these instrumentations may be redundant, i.e., removing them will not affect the process of checking for data races in the program. This is because some read and write operations are guaranteed to never introduce any additional data races in the program and hence such operations need not be instrumented.

As described earlier, our race detection algorithms also keep track of the `finish`, `async`, `isolated` blocks in the program. Hence, they require instrumentations for the start and end of every such block in the program. But these instrumentations are all necessary to maintain the parallelism structure at runtime in our algorithms<sup>1</sup>.

In this chapter, we describe static analysis that can be used to reduce the instru-

---

<sup>1</sup>Though some redundant `finish` blocks can be removed as well, we ignore it because, removing these redundant `finish` blocks will not improve the runtime of our algorithms (since the number of memory accesses in a typical parallel program is much larger than the number of `finish` instances).

mentation and hence improve the runtime performance of the instrumented program. These analysis apply to all the race detection algorithms that we have described so far. We also include an example that depicts how each of these static analysis are used to eliminate instrumentation points. Figure 7.1 shows a program in HJ with all its read and write operations instrumented (*markRead* and *markWrite* refers to the call to the library). Suppose that the main task is always guaranteed to start executing this portion of the program. This will be used as the baseline to depict these optimizations. Note that the instrumentations that are needed for the *finish* and *async* blocks are not shown in this example.

## 7.1 Main Task Check Elimination in Sequential Code Regions

The first static optimization aims at eliminating redundant instrumentation points that are added in the sequential code regions in the main task. A parallel program will always start and end with sequential code regions and will contain alternating parallel and sequential code regions in the middle. It is clear that no read or write operation in the sequential code regions of the program can result in a data race. Hence, there is no need to instrument the read and write operations in such sequential code regions of the program. In an HJ program, the sequential code regions are the regions of the program that are outside the outermost *finish* blocks<sup>2</sup> and are executed by the *main task*. Thus, in an HJ program, there is no need to instrument the read and write operations in such sequential code regions of the main task.

---

<sup>2</sup>This is assuming there are no *asyncs* outside any *finish* in the program. If there are any such *asyncs*, then the only sequential code regions in the program are the regions outside the outermost *finish* and before the first such *async*.



```

1  int [] A, B; Foo p;
2  ... ...
3  markWrite(p, x);
4  p.x = 0;
5  finish {
6    for (int i=0; i<size; i++ ) {
7      final int ind = i;
8      async {
9        markRead(A, ind);
10       markRead(B, ind);
11       markWrite(p, x);
12       p.x = A[ind] + B[ind];
13       Foo q = new Foo();
14       for (int j=0; j<ind; j++) {
15         markRead(p, x);
16         markWrite(q, x);
17         q.x = p.x + 1;
18         markRead(q, y);
19         markWrite(B, j);
20         B[i] = q.y + ind;
21       }
22     }
23   }
24 }

```

Figure 7.1 : An example HJ program with all read and write operations instrumented

```

1  int [] A, B; Foo p;
2  ... ...
3  p.x = 0;
4  finish {
5    for (int i=0; i<size; i++ ) {
6      final int ind = i;
7      async {
8        markRead(A, ind);
9        markRead(B, ind);
10       markWrite(p, x);
11       p.x = A[ind] + B[ind];
12       Foo q = new Foo();
13       for (int j=0; j<ind; j++) {
14         markRead(p, x);
15         markWrite(q, x);
16         q.x = p.x + 1;
17         markRead(q, y);
18         markWrite(B, j);
19         B[i] = q.y + ind;
20       }
21     }
22   }
23 }

```

Figure 7.2 : After applying the main task check elimination optimization on the program in Figure 7.1

Figure 7.2 shows the result of eliminating the instrumentation points in the sequential code regions of the program in Figure 7.1. The program in Figure 7.1 contains a write to a heap location  $p.x$  in line 4 which is part of the sequential code region executed by the main task. Hence the corresponding call to the library in line 3 can be eliminated.

## 7.2 Read-only Check Elimination in Parallel Code Regions

The input program may have shared memory locations that are written by the sequential regions of the program and only read within parallel regions of the program. Such read operations within parallel regions of the program need not be instrumented because parallel tasks reading from the same memory location will never lead to a conflict. In order to perform this optimization, the compiler implements an interprocedural side-effect analysis to detect potential write operations to shared memory locations within the parallel regions of the given program. If there is no possible write to a shared memory location  $M$  in the parallel regions of the program, that clearly shows that all accesses to  $M$  in the parallel regions must be read-only and hence the instrumentation points corresponding to these reads can be eliminated. (The checks for the writes in the sequential regions, if any, will be eliminated by the rule in Section 7.1).

The result of applying this optimization on the program in Figure 7.2 is shown in figure 7.3. There is no write to array  $A$  within the parallel regions of the program in Figure 7.2. Hence the instrumentation in line 8 corresponding to the read of  $A$  in line 11 can be removed.

```

1  int [] A, B; Foo p;
2  ... ...
3  p.x = 0;
4  finish {
5    for (int i=0; i<size; i++ ) {
6      final int ind = i;
7      async {
8        markRead(B, ind);
9        markWrite(p, x);
10       p.x = A[ind] + B[ind];
11       Foo q = new Foo();
12       for (int j=0; j<ind; j++) {
13         markRead(p, x);
14         markWrite(q, x);
15         q.x = p.x + 1;
16         markRead(q, y);
17         markWrite(B, j);
18         B[j] = q.y + ind;
19       }
20     }
21   }
22 }

```

Figure 7.3 : After applying the read-only check optimization on the program in Figure 7.2

```

1  int [] A, B; Foo p;
2  ... ...
3  p.x = 0;
4  finish {
5    for (int i=0; i<size; i++ ) {
6      final int ind = i;
7      async {
8        markRead(B, ind);
9        markWrite(p, x);
10       p.x = A[ind] + B[ind];
11       Foo q = new Foo();
12       for (int j=0; j<ind; j++) {
13         markRead(p, x);
14         q.x = p.x + 1;
15         markWrite(B, j);
16         B[j] = q.y + ind;
17       }
18     }
19   }
20 }

```

Figure 7.4 : After applying the escape analysis and check elimination optimization on the program in Figure 7.3

### 7.3 Escape Analysis

The input program may include many parallel tasks. A data race occurs in the program only when two or more tasks access a shared memory location and at least one of them is a write. Suppose an object is created inside a task and it never escapes that task, then no other task can access this object [50, 51, 52] and hence it cannot lead to a data race. In order to ensure the task-local attribute, the compiler performs an inter-procedural analysis that identifies if an object is shared among tasks. This also requires an alias analysis to ensure that no alias of the object escapes the task. Thus, if an object  $O$  is proven to not escape a task, then the instrumentation points corresponding to all accesses to  $O$  can be eliminated.

The object  $q$  in the program in Figure 7.3 is created in line 11 within a task and it never escapes this task. Thus no access to  $q$  can lead to a determinacy race. Hence, the instrumentation points in line 14 and 16 corresponding to access to  $q$  are eliminated and the resulting program is shown in Figure 7.4.

### 7.4 Loop Invariant Check Motion

Recall that the instrumentation corresponding to a memory access to  $M$  will first check if the task that previously accessed  $M$  conflicts with the current task and also update the information that the current task now accessed  $M$ . If there are multiple accesses of the same type (read or write) to  $M$  by a task, then it is sufficient to instrument one such access because other instrumentations will only add to the overhead by unnecessarily repeating the steps. Suppose the input program accesses a shared memory location  $M$  unconditionally inside a loop, the instrumentation corresponding to this access to  $M$  can be moved outside the loop in order to prevent multiple calls

```

1  int [] A, B; Foo p;
2  ... ...
3  p.x = 0;
4  finish {
5    for (int i=0; i<size; i++) {
6      final int ind = i;
7      async {
8        markRead(B, ind);
9        markWrite(p, x);
10     p.x = A[ind] + B[ind];
11     Foo q = new Foo();
12     if (ind > 0)
13       markRead(p, x);
14     for (int j=0; j<ind; j++) {
15       q.x = p.x + 1;
16       markWrite(B, j);
17       B[j] = q.y + ind;
18     }
19   }
20 }
21 }

```

```

1  int [] A, B; Foo p;
2  ... ...
3  p.x = 0;
4  finish {
5    for (int i=0; i<size; i++) {
6      final int ind = i;
7      async {
8        markRead(B, ind);
9        markWrite(p, x);
10     p.x = A[ind] + B[ind];
11     Foo q = new Foo();
12     for (int j=0; j<ind; j++) {
13       q.x = p.x + 1;
14       markWrite(B, j);
15       B[j] = q.y + ind;
16     }
17   }
18 }
19 }

```

Figure 7.5 : After applying the loop invariant check elimination optimization on the program in Figure 7.4

Figure 7.6 : After applying the read/write check elimination optimization on the program in Figure 7.5

to the instrumented function for  $M$ .

In summary, given a memory access  $M$  that is performed unconditionally on every iteration of a sequential loop, the instrumentation for  $M$  can be hoisted out of the loop by using classical loop-invariant code motion. This transformation includes the insertion of a zero-trip test to ensure that the loop-invariant check is performed only when the loop executes for one or more iterations.

In Figure 7.4, the program contains a read of  $p.x$  in line 13 that is inside a sequential loop. Now, since the same memory location is accessed in every iteration of the loop, the instrumentation for this access is moved out of the loop as shown in Figure 7.5. Note the test for the non-zero trip count in line 12 guarding this instrumentation outside the loop.

## 7.5 Read/Write Check Elimination

In the previous optimization we showed that it is sufficient to instrument one access to a memory location  $M$  if there are multiple accesses of the same type to  $M$  by a task. In this optimization, we claim that if there are two accesses  $M_1$  and  $M_2$  to the same memory location in a task, then we can use the following rules to eliminate one of them. It works on the basic idea that the instrumentation for a write subsumes that for a read in the race detection algorithms. An intuitive argument for this is that, if a read to a memory location  $M$  in a task  $\tau$  causes a data race, then a write to  $M$  in  $\tau$  will definitely cause a data race.

1. If  $M_1$  dominates  $M_2$  and  $M_2$  is a read operation, then the instrumentation for  $M_2$  can be eliminated (since  $M_1$  is either a read or write operation).
2. If  $M_2$  postdominates  $M_1$  and  $M_1$  is a read operation, then the check for  $M_1$  can

be eliminated (since  $M_2$  is either a read or write operation). This rule tends to be applicable in fewer situations than the previous rule in practice, because computation of postdominance includes the possibility of exceptional control flow.

Consider the program in Figure 7.5. There is an instrumentation for the write to  $p.x$  in line 9 and an instrumentation corresponding to the read of the same memory location in line 13. Since the instrumentation in line 9 dominates the one in line 13 and latter is not a write, the latter can be eliminated.

## Chapter 8

### Experimental Results

We have so far presented a sequential algorithm (ESP-bags) and a parallel algorithm (SPD3) for dynamic data race detection in HJ programs. We also described an implementation of ESP-bags and SPD3 for HJ programs with `async`, `finish`, and `isolated` constructs. Also, we discussed some static optimizations to reduce the overhead of dynamic data race detectors.

In this chapter, we evaluate the performance of our implementation of the ESP-bags algorithm and the SPD3 algorithm for `async`, `finish`, and `isolated` constructs, with and without the static optimizations. First, we discuss the experimental setup including the system used for our evaluation, the list of benchmarks that we evaluate our algorithms on, and also the runtime settings used in our evaluation. Then, we discuss the data races that were identified as a result of applying our algorithms on these benchmarks. We then present an evaluation of the ESP-bags algorithm for `async`, `finish`, and `isolated` constructs. Finally, we present an evaluation of the SPD3 algorithm for `async`, `finish`, and `isolated` constructs. We also compare SPD3 with ESP-bags and two other algorithms, Eraser and FastTrack, from past work.

#### 8.1 Experimental Setup

Our experiments were conducted on a 16-core (quad-socket, quad-core per socket) Intel Xeon 2.4GHz system with 30 GB memory, running Red Hat Linux (RHEL



Table 8.1 : List of Benchmarks Evaluated

Source	Benchmark	Input	Description
JGF	Crypt (Section 2)	C	IDEA encryption
	LUFact (Section 2)	C	LU Factorization
	MolDyn (Section 3)	B	Molecular Dynamics simulation
	MonteCarlo (Section 3)	B	Monte Carlo simulation
	RayTracer (Section 3)	B	3D Ray Tracer
	Series (Section 2)	C	Fourier coefficient analysis
	SOR (Section 2)	C	Successive over-relaxation
	SparseMatMult (Section 2)	C	Sparse Matrix multiplication
Bots	FFT	large	Fast Fourier Transformation
	Health	large	Simulates a country health system
	Nqueens	14	N Queens problem
	Strassen	large	Matrix Multiply with Strassen's method
Shootout	Fannkuch	10M	Indexed-access to tiny integer-sequence
	Mandelbrot	8000	Generate Mandelbrot set portable bitmap
EC2	Matmul	1000^2	Matrix Multiplication (Iterative)

5), and Sun Hotspot JDK 1.6. To reduce the impact of JIT compilation, garbage collection and other JVM services, we report the smallest time measured in 3 runs repeated in the same JVM instance for each data point.

We present our evaluations on a suite of 15 benchmarks listed in Table 8.1. It includes eight Java Grande Forum benchmarks (JGF) [53], four Barcelona OpenMP Task Suites benchmarks (BOTS) [54], two Shootout benchmarks [55], and one EC2 challenge benchmark. All benchmarks were written in HJ using only `async`, `finish`, and `isolated` constructs for parallelism, with fine grained one-async-per-iteration parallelism for parallel loops. The original version of the JGF benchmarks in Java con-

tained “chunked” parallel loops with programmer-specified decomposition into coarse grained one-chunk-per-thread parallelism. The fine grained task-parallel versions of the JGF benchmarks used for the evaluation here were obtained by rewriting the chunked loops into “unchunked” parallel loops. In addition, barrier operations in the JGF benchmarks were replaced by appropriate finish constructs.

Note that benchmarks used for evaluation are limited by the availability of programs ported to HJ. HJ-Base refers to the uninstrumented baseline version of each of these benchmarks. All the JGF benchmarks were configured to run with the largest available input size. The input sizes used for all the benchmarks are also shown in Table 8.1.

To simulate the sequential depth-first execution needed for the ESP-bags algorithm, HJ tasks were scheduled on a work-stealing scheduler with work-first policy [19] on 1-thread that performs a sequential depth-first execution. We also evaluated the ESP-bags algorithm on serialized versions of the benchmarks which remove the `async`, `finish`, and `isolated` constructs in the program (by eliding the keywords), thereby producing a sequential depth-first execution without the overhead of parallel constructs (but with the overhead of ESP-bags instrumentation).

To evaluate the SPD3 algorithm, HJ tasks were scheduled on a fixed number of worker threads using a work-stealing scheduler with an adaptive policy [56], which has been shown to perform the best among the schedulers available for HJ across a wide range of benchmarks.

## 8.2 Data Races Observed

While running our algorithms on the coarse grained HJ versions of the eight JGF benchmarks, data races were reported for four of the benchmarks: LUFact, MolDyn,

RayTracer, and SOR. The data race reports pointed to races in shared arrays that were used by the programmer to implement custom barriers. However, all the custom barrier implementations were incorrect because they involved unsynchronized spin loops on shared array elements. Even though the programmer declared the array references as volatile, the volatile declaration does not apply to the elements of the array. (In all fairness to the programmer, the JGF benchmarks were written in the late 1990's when many Java practitioners were unaware of the implications of the Java memory model.) We fixed these data races in the HJ versions of the JGF benchmarks by replacing the incorrect barrier operations with appropriate finish constructs.

Our algorithms also found another data race which turned out to be a benign race. This was due to repeated parallel assignments of the same value to the same location in the HJ version of the MonteCarlo benchmark, which was corrected by removing the redundant assignments. After that, all the benchmarks used for evaluation here were observed to be data-race-free for the inputs used.

### 8.3 Evaluation of ESP-bags for Async-Finish-Isolated

In this section, we evaluate our implementation of the ESP-bags algorithm for `async`, `finish`, and `isolated` constructs along with the static optimizations.

#### 8.3.1 Performance of ESP-bags

Table 8.2 shows the results of applying the ESP-bags algorithm to our benchmarks. The table gives the original execution time for each benchmark, i.e., the execution time of the benchmark without any instrumentation on 1-thread using the work-first work-stealing runtime. It also shows the slowdown of the benchmark when instrumented for the ESP-bags algorithm, with and without the static optimizations

Table 8.2 : Slowdown of the ESP-bags algorithm relative to uninstrumented 1-thread execution time on a work-first work-stealing runtime

Benchmark	Uninstrumented 1-thread Execution Time (s)	Slowdown of ESP-bags	
		w/o optimizations	w/ optimizations
Crypt	10.16	8.52×	12.93×
LUFact	24.91	6.37×	5.17×
MolDyn	7757.11	3.24×	2.67×
MonteCarlo	18.65	2.17×	1.23×
RayTracer	42.70	11.69×	5.15×
Series	1379.66	1.01×	1.01×
SOR	13.33	3.97×	2.66×
SparseMatMult	29.19	7.08×	1.74×
FFT	1.09	4.63×	4.16×
Health	93.04	2.00×	1.84×
Nqueens	33.22	2.77×	2.71×
Strassen	9.78	3.17×	3.07×
Fannkuch	9.11	1.88×	1.86×
Mandelbrot	11.29	1.27×	1.20×
Matmul	34.61	3.83×	1.05×
Geo Mean	-	3.40×	2.47×

described in Chapter 7.

On average, the slowdown for the benchmarks with the ESP-bags algorithm is  $3.40\times$  without optimizations. When all the static optimizations described in Section 7 are applied, the average slowdown drops to  $2.47\times$ . The slowdown factor of  $S$  means that, when data race detection is performed using the ESP-bags algorithm, the benchmark is  $S\times$  slower than the uninstrumented 1-thread execution time on a work-first work-stealing runtime.

With static optimizations, the slowdown for all benchmarks except Crypt is under  $10\times$ . The only other benchmarks which have a slowdown of more than  $5\times$  are LUFact and RayTracer. The slowdowns for seven of the benchmarks are under  $2\times$ . The high overhead in the case of Crypt, LUFact, and RayTracer is due to the fact that these benchmarks perform a large number of memory accesses and each of these accesses require updates to the reader/writer fields of the corresponding memory locations. There is no slowdown in the case of Series because most of the code accesses local variables<sup>1</sup>. In HJ, none of the local variables can be shared across tasks and hence we do not instrument any access to these variables.

### 8.3.2 Performance of Static Optimizations

We now discuss the effects of the static compiler optimizations (described in Section 7) on the benchmarks. The static optimizations that were performed include check elimination in sequential code regions in the main task, read-only check elimination in parallel code regions, escape analysis based optimization, loop invariant check motion, and read/write check elimination. As evident from Table 8.2, some of

---

<sup>1</sup>Note that dynamic data race detection focuses on heap accesses (static and instance data) since there cannot be a race on local variables.

the benchmarks like RayTracer, SparseMatMult, and Matmul benefit a lot from the optimizations, with a maximum reduction in slowdown of about 75% for SparseMatMult. On the other hand, for other benchmarks the reduction is relatively low. The optimizations do not reduce the slowdown much for FFT, Nqueens, and Strassen because these benchmarks do not contain many redundant memory accesses for which the instrumentations could be removed. In the case of LUFact, though many of the instrumentations are eliminated, a significant fraction of them still remain and hence there is not much performance improvement in it due to optimizations. On average, there is a 27% reduction in the slowdown of all the benchmarks due these optimizations.

In Crypt, the slowdown of the ESP-bags algorithm with optimizations is higher than that without optimizations. This anomaly occurs because our results use the minimum execution time in 3 iterations repeated in the same JVM instance. Since the memory footprint of Crypt is very high, it takes a longer time to warm up the hardware resources in this case. With Crypt, when we use the minimum execution time in 30 iterations repeated in the same JVM instance, the optimized version performs better than the unoptimized version as explained in Appendix A.

### **Breakdown of Static Optimizations**

We now describe the effects of each of the static optimizations separately on the performance of the benchmarks. Figure 8.1 shows the breakdown of the effects of each of the static optimizations. The graph also shows the slowdown without any optimization and with the whole set of optimizations enabled. The Main Task Check Elimination optimization described in Section 7 is applied to all the versions included here, including the unoptimized version. This is because we consider that optimiza-

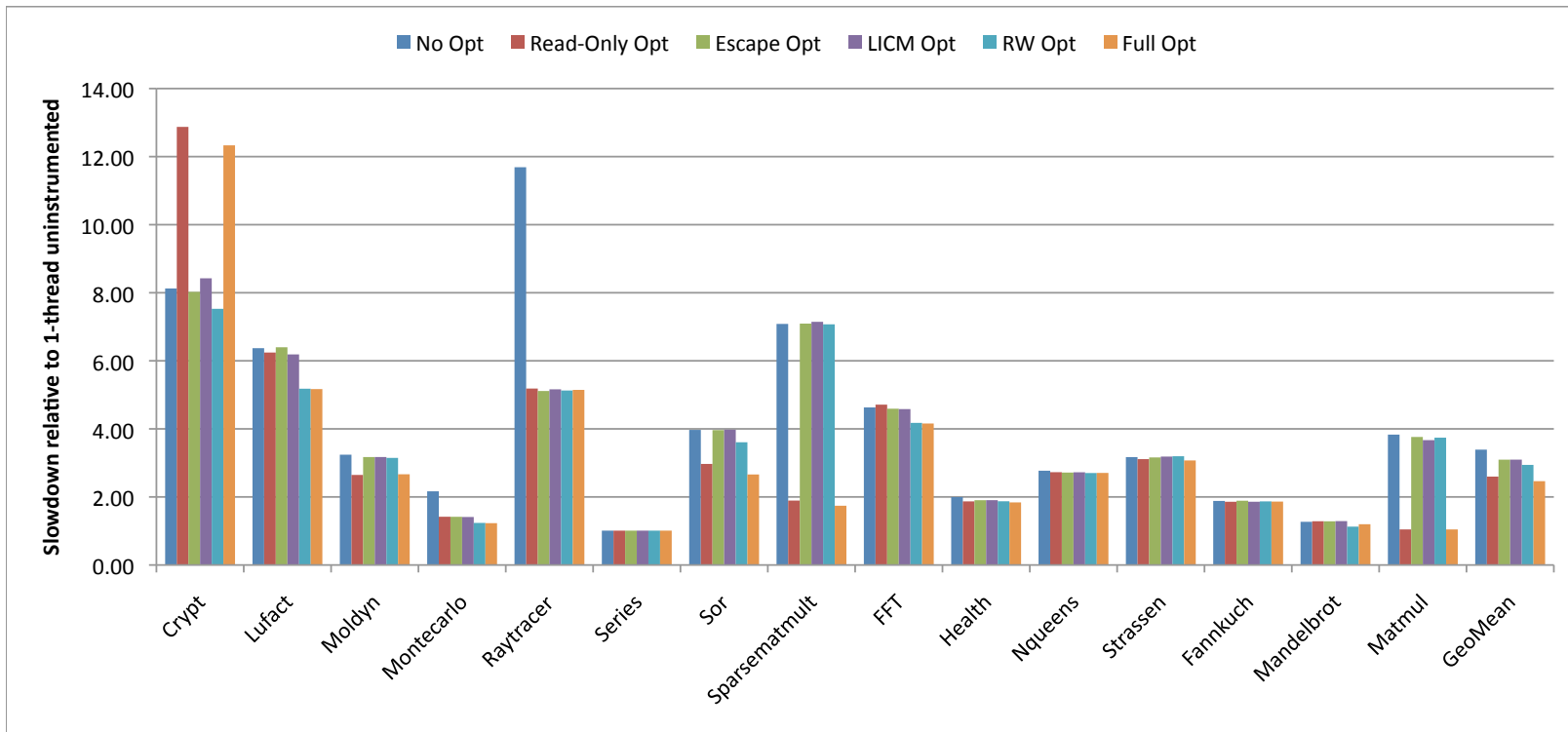


Figure 8.1 : Breakdown of the effect of static optimizations on the performance improvement of the ESP-bags algorithm.

tion as a basic step without which there could be excessive instrumentations. For each benchmark, the first bar shows the slowdown of the benchmark with no optimization, the second bar shows the slowdown when the read-only check elimination alone is enabled, the third one shows the slowdown when the escape analysis based optimization alone is enabled, the fourth bar shows the slowdown when loop invariant code motion alone is enabled, the fifth one shows the slowdown when the read-write optimization alone is enabled, and the last one shows the slowdown when all the optimizations are enabled simultaneously

The read-only check elimination performs much better than the other optimizations for most of the benchmarks, like MolDyn, SOR, SparseMatMult, and Matmul. This is because in these benchmarks the parallel regions include reads to many arrays which are written only in the sequential regions of the code. Hence, this optimization eliminates the instrumentation for all these reads. It contributes the most to the overall performance improvement in the full optimized version. The read-write optimization improves the performance of LUFact, SOR, and FFT a bit, but does not have much effect on other benchmarks. The Loop invariant code motion and the Escape analysis based optimization does not seem to help any of these benchmarks to a great extent.

The slowdown of Crypt in the presence of read-only check elimination is higher than the slowdown in the unoptimized version. This is due to fact that we use the minimum execution time of 3 iterations of the benchmark repeated in the same JVM instance, which is not enough to warm up the hardware resources in the case of Crypt. Appendix A explains how the same anomaly in the fully optimized version is no longer present when we use the minimum execution time of 30 iterations repeated in the same JVM instance. A similar observation was made for Crypt with read-only



check elimination as well. In other words, when we use the minimum execution time of 30 iterations in the same JVM instance, the slowdown of Crypt with read-only check elimination is lower than that with the unoptimized version.

### 8.3.3 Comparison with Serialized Execution

Recall that the ESP-bags algorithm requires that the input program is executed in a sequential depth-first manner. We achieve the sequential depth-first execution using two different techniques: one is by executing the program on 1-thread with the work-first work-stealing runtime and the other is by serializing the given program and executing the serial program. The results we have seen so far were obtained by executing the program on 1-thread with work-first work-stealing. Now, we show the difference in the performance of the ESP-bags algorithm between the 1-thread based execution and the serialized execution. The serialized execution avoids the overhead of parallel constructs (`async`, `finish`), but retains the overhead of tracking S-bags, P-bags, and memory locations.

Figure 8.2 compares the performance of 1-thread uninstrumented, 1-thread with ESP-bags, serialized uninstrumented, and serialized with ESP-bags. The graph shows the execution time for each of these versions normalized to 1-thread uninstrumented time. Hence, the values for 1-thread uninstrumented bar is 1.00 for all the benchmarks. (It is included for clarity.)

The serialized versions are expected to be faster than the 1-thread versions because of the extra overhead involved with 1-thread versions due to the support for parallelism. The serialized uninstrumented version is either same as or less than the 1-thread uninstrumented for all the benchmarks. On an average, the serialized uninstrumented version is 21% faster than the 1-thread uninstrumented version. The

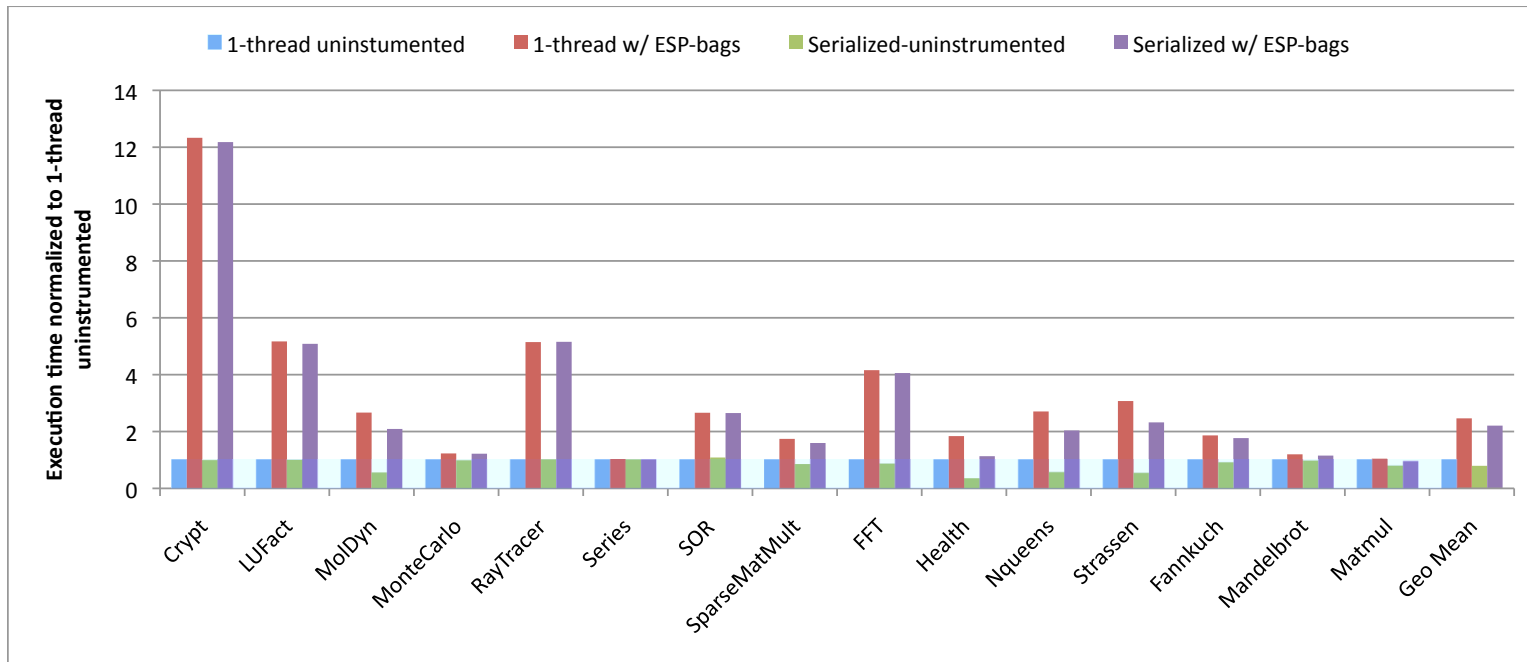


Figure 8.2 : Comparison of the slowdown of ESP-bags using the 1-thread execution with the slowdown of ESP-bags using the serialized execution

serialized version with ESP-bags is almost same as the 1-thread version with ESP-bags in benchmarks like Crypt, LUFact, RayTracer, SOR, and FFT. Whereas, in the case of benchmarks like MolDyn, Health, Nqueens, and Strassen, the serialized version with ESP-bags is faster than the 1-thread version with ESP-bags. On an average, the serialized version with ESP-bags is about 10% faster than the 1-thread version with ESP-bags.

## 8.4 Evaluation of SPD3 for Async-Finish-Isolated

In this section, we evaluate the performance of the SPD3 algorithm for `async`, `finish`, and `isolated`. We present a comparison of the performance of the SPD3 algorithm with that of the ESP-bags algorithm. Also, we compare SPD3 with two dynamic data race detectors from past work, namely Eraser [3] and FastTrack [1].

### 8.4.1 Performance of SPD3

Figure 8.3 shows the relative slowdown of SPD3 for all benchmarks when executed with 1, 2, 4, 8, and 16 worker threads. (Recall that these benchmarks create many more async tasks than the number of worker threads.) The relative slowdown on  $n$  threads refer to the slowdown of the SPD3 instrumented version of the benchmark executing on  $n$  threads compared with the HJ-Base version executing on  $n$  threads. Ideally, a scalable race detector should have a constant relative slowdown as the number of worker threads increases. As evident from Figure 8.3, the slowdown for many of the benchmarks decrease as the number of worker threads increase from 1 to 16. The geometric mean of the slowdowns for all the benchmarks on 16 threads is  $2.78\times$ .

Though the geometric mean is below  $3\times$ , four of the 15 benchmarks (Crypt, LU-

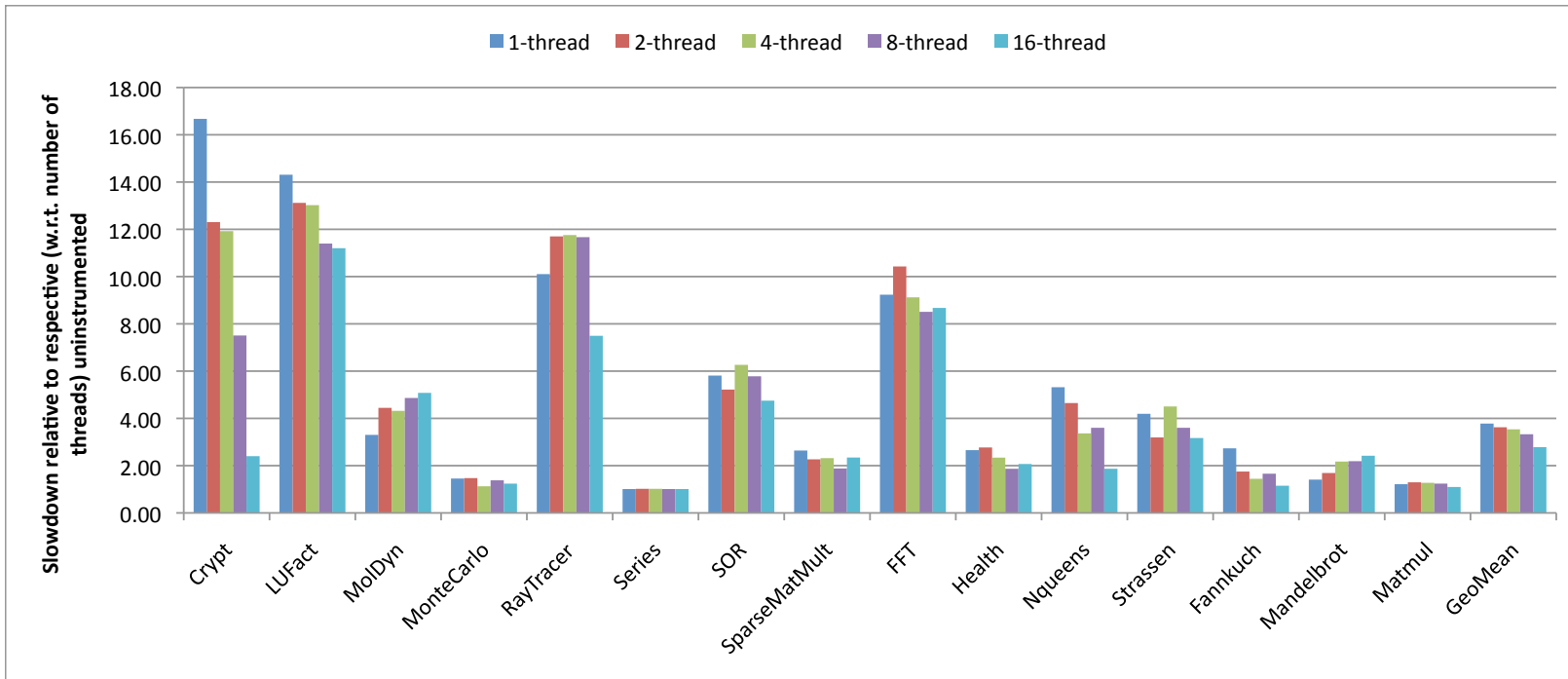


Figure 8.3 : Relative slowdown of SPD3 for all benchmarks on 1, 2, 4, 8, and 16 threads. Relative slowdown on  $n$  threads refers to the slowdown of the SPD3 version on  $n$  threads compared to the HJ-Base version on  $n$  threads.

Fact, RayTracer, and FFT) exhibited a slowdown around  $10\times$  for worker threads from 1 to 16. This is because these benchmarks contain larger numbers of shared locations that need to be monitored for race detection. These were also the benchmarks with the four largest overheads for ESP-bags. As discussed later, other race detection algorithms exhibit much larger slowdowns for these benchmarks than SPD3. Note that even in these cases the slowdowns are similar across 1 to 16 threads. This shows that SPD3 exhibits scalable performance.

The slowdown for 1-thread is higher than that for all other threads in many benchmarks. This is because our implementation uses *compareAndSet* operations on atomic variables. These operations are not optimized for the no contention scenario as with 1-thread. Instead, if we use a lock that is optimized for no contention scenario, the slowdown for 1-thread cases would have been a lot lower. But that implementation does not scale well for larger numbers of threads. For example, the lock based implementation is  $1.8\times$  slower (on average) than the *compareAndSet* implementation when running on 16-threads. While the two implementations are close for many benchmarks (within a factor of 2), there is a difference of up to  $7\times$  for some benchmarks, when running on 16-threads. The *compareAndSet* implementation is always faster than the lock based implementation for larger numbers of threads. Since our aim was to make the algorithm scalable, we chose the *compareAndSet* approach.

Table 8.3 shows the uninstrumented execution times on 1-thread and 16-threads for all the benchmarks and the slowdown of the SPD3 algorithm on 1-thread and 16-threads relative to the uninstrumented 1-thread and 16-thread execution times respectively. For the Matmul benchmark, where the uninstrumented version scales well from 1-thread to 16-threads, the slowdown of SPD3 decreases only by a small fraction. Whereas, in the case of Crypt, where the uninstrumented version with 16-threads is

Table 8.3 : Slowdown of the SPD3 algorithm on 1-thread and 16-threads relative to uninstrumented 1-thread and 16-thread execution times respectively on an adaptive work-stealing runtime

Benchmark	Uninstrumented		Slowdown of SPD3	
	Execution Time (s)		1-thread	16-threads
	1-thread	16-threads		
Crypt	10.21	72.30	16.67×	2.40×
LUFact	25.03	6.45	14.31×	11.20×
MolDyn	8062.37	543.55	3.30×	5.08×
MonteCarlo	20.01	5.41	1.45×	1.24×
RayTracer	43.45	6.99	10.10×	7.49×
Series	1391.90	88.18	1.00×	1.01×
SOR	14.21	2.85	5.81×	4.75×
SparseMatMult	27.12	4.20	2.64×	2.34×
FFT	1.04	0.15	9.23×	8.67×
Health	106.67	58.46	2.66×	2.07×
Nqueens	27.96	11.21	5.32×	1.87×
Strassen	9.64	5.69	4.19×	3.17×
Fannkuch	9.21	33.94	2.73×	1.15×
Mandelbrot	11.35	3.68	1.41×	2.42×
Matmul	33.89	2.05	1.22×	1.10×
GeoMean	-	-	3.78×	2.78×

much slower than the uninstrumented 1-thread execution, the slowdown of SPD3 on 16-threads is much less than the slowdown on 1-thread. It is clear from the table that there is no correlation between the speedup of the uninstrumented version and the scaling of the slowdown factors as we go from 1-thread to 16-threads. Similarly, there is no correlation between the absolute execution times and the slowdown of SPD3.

### Breakdown of Optimizations

Now, we look at the effect of each of the static optimizations on the performance of SPD3. Figure 8.4 shows the breakdown of the effect of the static optimizations on the performance of SPD3 for all the benchmarks we consider while executing on 16-threads. For each benchmark, there are six bars representing the slowdown of the SPD3 algorithm when no optimization, read-only check elimination, escape analysis based optimization, loop invariant code motion, read-write check elimination, and full set of optimizations are enabled, respectively.

As in the case with the ESP-bags algorithm, the read-only check elimination performs the best among the optimizations we presented. This optimization plays a major part in reducing the overhead of SPD3 for benchmarks like LUFact, MolDyn, SOR, SparseMatMult, and Matmul. Again, this is because of the fact that these benchmarks have lot of memory locations that are written in the sequential regions of the program and are only read within parallel regions. The read-write check elimination contributes to the reduction in overhead of SPD3 in benchmarks like LUFact and FFT. The other optimizations, escape analysis based one and loop invariant code motion do not help in reducing the overhead of SPD3 in any of these benchmarks.

The slowdown of Crypt with full optimization is slightly higher than the slowdown in the unoptimized version. This anomaly is due to the reason discussed in the case of

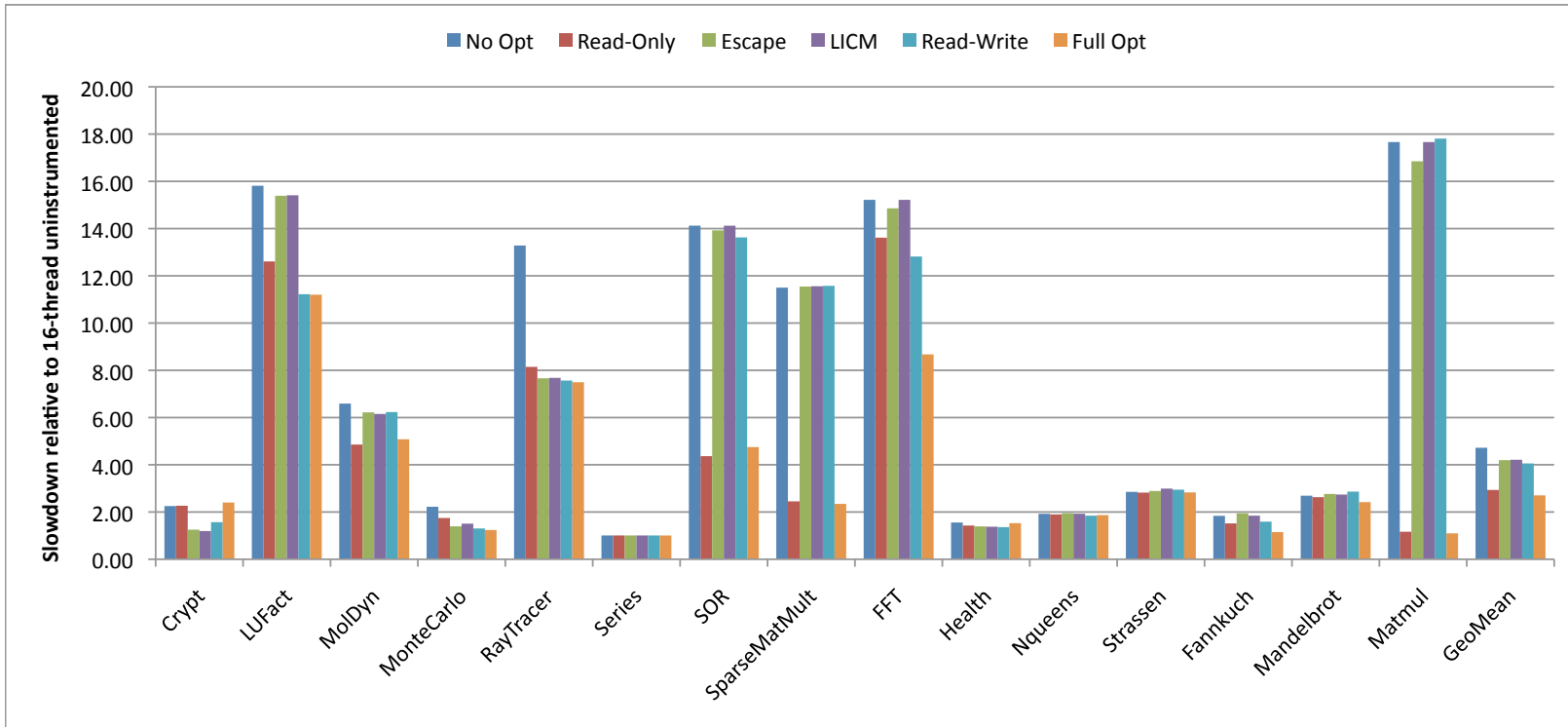


Figure 8.4 : Breakdown of the effect of static optimizations on the performance improvement of the SPD3 algorithm while executing on 16-threads.



Crypt with ESP-bags in Section 8.3.2. For ESP-bags algorithm, Appendix A shows that this anomaly is not present if we use the minimum execution time of 30 iterations, instead of 3 iterations, repeated in the same JVM instance. The same argument holds for SPD3 with full optimization and with read-only check elimination as well.

### Comparison of Synchronized and CompareAndSet Implementations

The SPD3 algorithm executes the input program in parallel but requires that the updates to the shadow memory of every memory location are done *atomically* with respect to other updates to the same memory location. As we discussed earlier, we evaluated two different ways of implementing this atomicity requirement. One is using Java's *synchronized* and the other is using a non-blocking algorithm with *CompareAndSet* (CAS) on atomic integers. Now, we present a comparison of the performance of SPD3 based on these two implementations.

Figure 8.5 shows the comparison of the slowdown of the SPD3 implementation based on synchronized with the slowdown of the implementation based on CAS while executing on 16-threads. Both the slowdown factors in this graph for every benchmark were calculated using the time for the uninstrumented version of the benchmark running on 16-threads as the baseline.

For all the benchmarks except SparseMatMult, the CAS version performs better than the synchronized version on 16-threads. The highest difference is for MolDyn where the CAS version is  $7.2\times$  faster than the synchronized version. For other benchmarks like RayTracer, FFT, Health, Nqueens, and Strassen the CAS version is over  $2\times$  faster than the synchronized version. On an average, the CAS version is  $1.8\times$  faster than the synchronized version. In the case of SparseMatMult, the synchronized version is  $1.2\times$  faster than the CAS version. This is because there are not many paral-

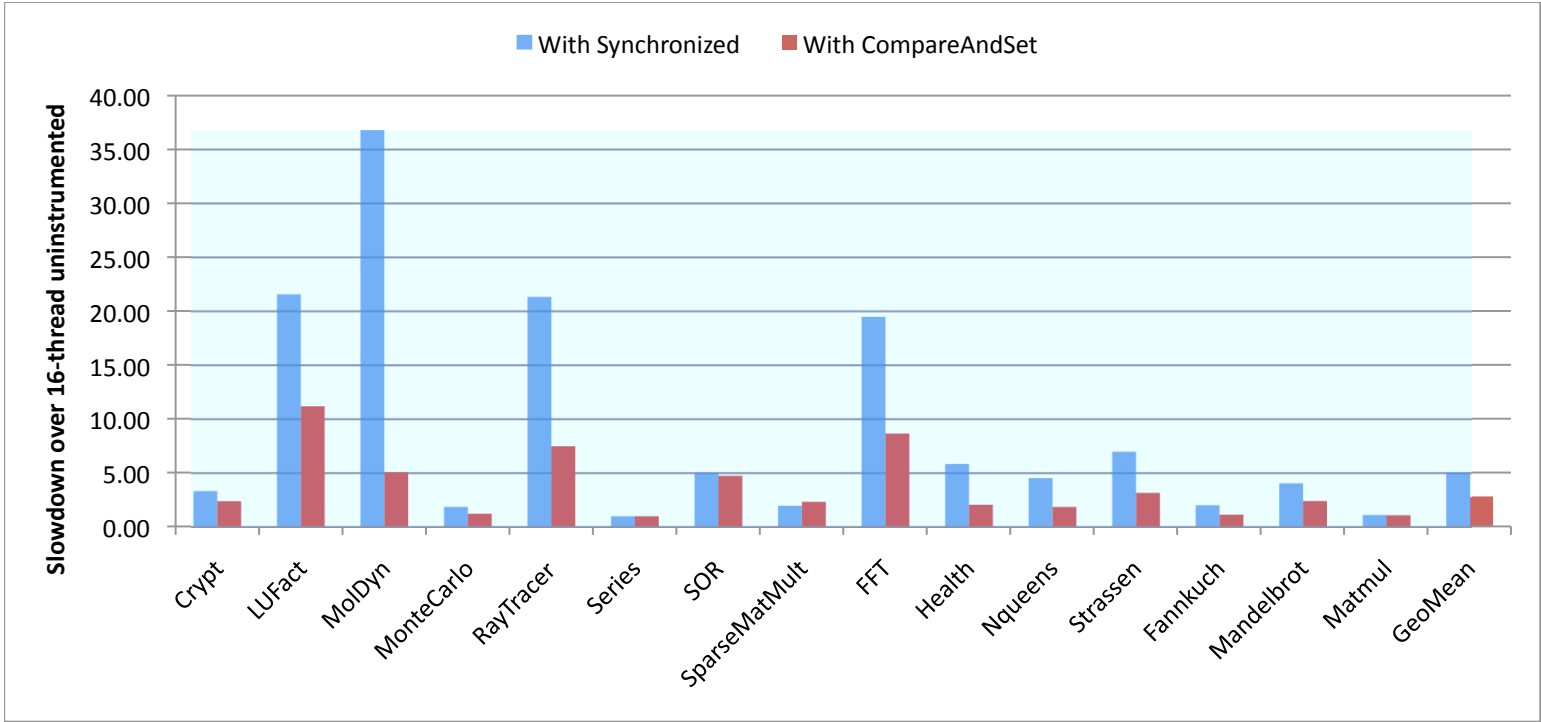


Figure 8.5 : Comparison of the performance of the SPD3 algorithm based on “Synchronized” with that of SPD3 based on “CompareAndSet” while executing on 16-threads.

lel accesses to memory locations in SparseMatMult after the static optimizations. In other words, most of the accesses to shared memory locations in SparseMatMult are ordered with each other. Hence, there is no contention between threads in updating their shadow memory locations due to which the synchronized version performs well. On the other hand, the additional overhead involved with CAS, which is unnecessary for this benchmark, leads to a poor performance.

This clearly proves that the CompareAndSet implementation helps SPD3 perform better than the synchronized version. Hence, all the results presented so far on SPD3 and those that will follow use the implementation based on CompareAndSet.

#### 8.4.2 Comparison of SPD3 with ESP-bags

In this section, we compare the performance of SPD3 with ESP-bags. Figure 8.6 shows the slowdown of ESP-bags and SPD3 for all the benchmarks, relative to the execution time of the 16-thread HJ-Base version. Note that the ESP-bags version runs on 1-thread (because it is a sequential algorithm) while the SPD3 version runs on 16-threads.

This comparison underscores the fact that the slowdown for a sequential approach to data race detection can be significantly larger than that of parallel approaches, when running on a parallel machine. For example, the slowdown is reduced by more than a  $15\times$  factor when moving from ESP-bags to SPD3 for Series and Matmul benchmarks and by more than a  $5\times$  factor for benchmarks like MolDyn and SparseMatMult that scale well. On the other hand, the slowdown for Crypt is similar for ESP-bags and SPD3 because the uninstrumented async-finish version of Crypt does not scale well. On average, SPD3 is  $3.2\times$  faster than ESP-bags on our 16-way system. This gap is expected to further increase on systems with larger number of cores.

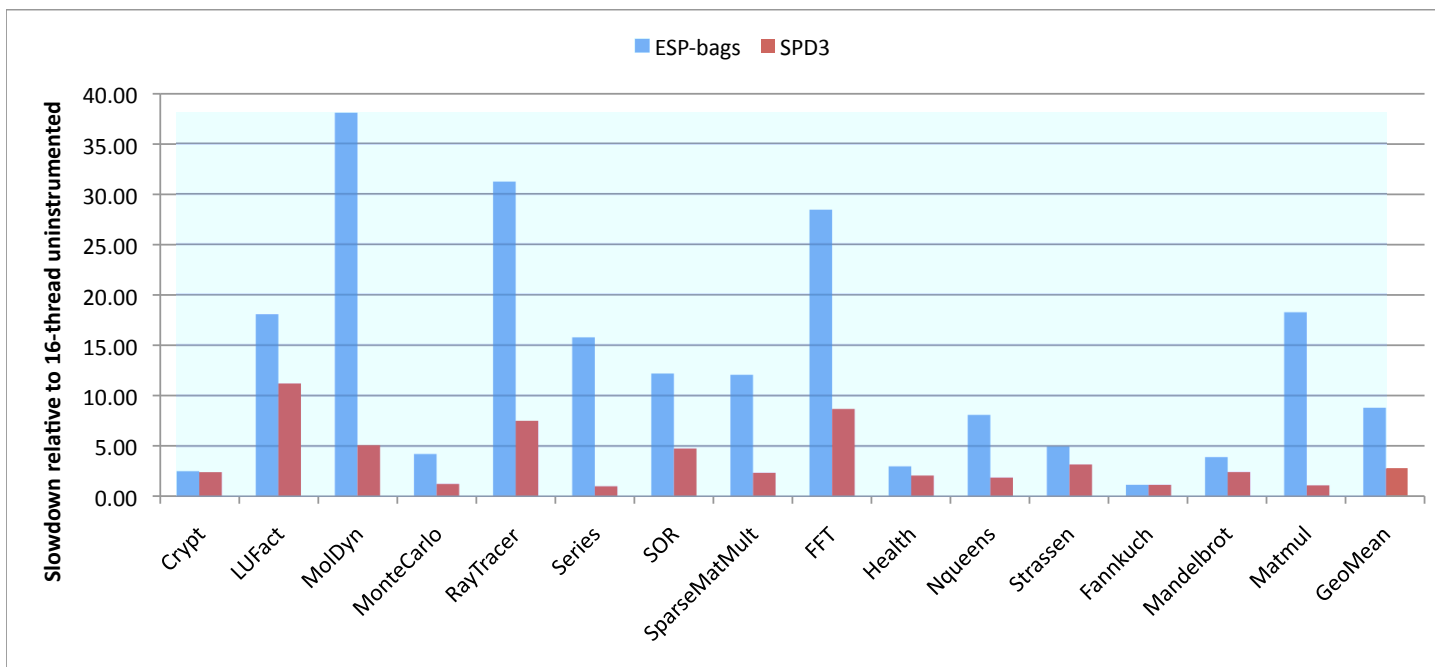


Figure 8.6 : Slowdown of ESP-bags and SPD3 relative to 16-thread HJ-Base version for all benchmarks. Note that the ESP-bags version runs on 1-thread while the SPD3 version runs on 16-threads.

### 8.4.3 Comparison of SPD3 with Eraser and FastTrack

Now, we compare SPD3 with two parallel dynamic data race detectors from the past, Eraser and FastTrack. For Eraser and FastTrack, we use the implementations included in the RoadRunner tool [57]. Since the performance of the FastTrack implementation available in the public RoadRunner download yielded worse results than those described in [1], we communicated with the implementers and received an improved implementation of FastTrack which was used to obtain the results reported in [11] and in this dissertation.

We only use the JGF benchmarks for comparisons with these algorithms since those are the only common benchmarks with past work on Eraser and FastTrack. However, since Eraser and FastTrack work on multithreaded Java programs rather than task-parallel variants like HJ, they used the original coarse-grained one-chunk-per-thread approach to loop parallelism in the JGF benchmarks and created one thread per core. Converting these programs to fine-grained parallel versions that create larger numbers of Java threads quickly leads to `OutOfMemoryError`'s due to the memory required by `Java` threads and the larger size of vector clocks.

So, to enable an apples-to-apples comparison in this section, we created coarse-grained `async-finish` versions of the JGF benchmarks with chunked loops for the HJ versions to match the multithreaded `Java` versions, even though it is more natural for the programmer to write the fine-grained HJ versions. Since Eraser and FastTrack were implemented in RoadRunner, we used the execution of the `Java` versions of these benchmarks on RoadRunner without instrumentation (RR-Base) as the baseline for calculating the slowdowns for Eraser and FastTrack. The differences between RR-Base and HJ-Base arise from the use of array views in the HJ version, and from the use of finish operations instead of barriers as discussed below.

As we mentioned earlier, our algorithms found data races in the barrier implementations of four of the JGF benchmarks. We observed that the default Eraser and FastTrack tools in the RoadRunner implementation did not report most of these data races. The only race reported was by FastTrack for SOR. After communication with the implementers of RoadRunner, we learned that RoadRunner recognizes a number of common barrier class implementations by default and generates special “Barrier Enter” and “Barrier Exit” events for them which in turn enables Eraser and FastTrack to take the barriers into account for race detection (even though the barriers are technically buggy). Further a “-nobARRIER” option can be used to suppress this barrier detection. We confirmed that all races were indeed reported by RoadRunner with the “-nobARRIER” option. However, all RoadRunner performance measurements reported here were obtained with the default settings (i.e., without the “-nobARRIER” option) to match RoadRunner’s intended handling of barrier idioms.

To undertake a performance comparison, we converted the four benchmarks to race-free HJ programs by replacing the buggy barriers by finish operations. In some cases, this caused the HJ-base version to be slower than the RR-base version as a result (since RR-base measures the performance of the unmodified JGF benchmarks with custom barriers).

It is also worth noting that the implementation of Eraser and FastTrack in RoadRunner include some optimizations that are orthogonal to the race detection algorithm used [57]. Similarly, the static optimizations discussed in Chapter 7 included in our implementation of SPD3 are also orthogonal to the race detection algorithm. Both these sets of optimizations could be performed on any race detection algorithm to improve its performance.

## Performance Comparison

Table 8.4 : Relative slowdown of Eraser, FastTrack and SPD3 for JGF benchmarks on 16 threads. The slowdown of Eraser and FastTrack was calculated relative to RR-Base while the slowdown of SPD3 was calculated relative to HJ-Base. For benchmarks marked with \*, race-free versions were used for SPD3 but the original (buggy) versions were used for Eraser and FastTrack.

Benchmark	RR-Base	Eraser	FastTrack	HJ-Base	SPD3
	Time(s)	Slowdown		Time(s)	Slowdown
Crypt	0.36	122.40×	133.24×	0.59	1.84×
LUFact*	1.47	17.95×	26.41×	5.41	1.08×
MolDyn*	16.19	8.39×	9.59×	3.75	13.56×
MonteCarlo	2.88	10.95×	13.54×	5.61	1.86×
RayTracer*	2.19	20.23×	17.45×	19.97	5.84×
Series	112.52	1.00×	1.00×	88.77	1.00×
SOR*	0.91	4.26×	8.36×	2.60	4.53×
SparseMatMult	2.75	14.29×	20.59×	4.61	1.72×
GeoMean	-	11.21×	13.87×	-	2.63×

Table 8.4 shows the slowdowns of Eraser, FastTrack, and SPD3 for all the JGF benchmarks on 16 threads. Note that the slowdowns of Eraser and FastTrack were calculated relative to RR-Base (with 16 threads), and the slowdown of SPD3 was calculated relative to HJ-Base (with 16 threads). For benchmarks marked with \*, race-free versions were used for SPD3 but the original versions were used for Eraser and FastTrack; this accounts for differences in the execution times of RR-Base and HJ-Base for some benchmarks since the HJ-Base versions include more synchronization to correct the bugs in the RR-Base versions.

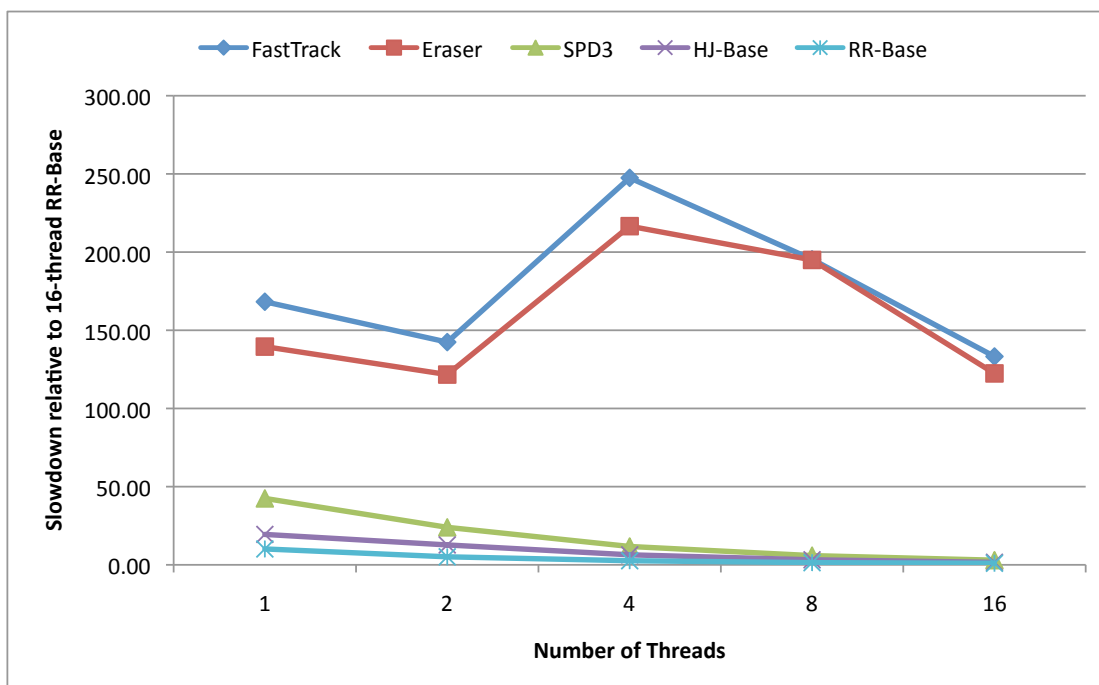


Figure 8.7 : Slowdown (relative to 16-threads RR-Base) of RR-Base, Eraser, FastTrack, HJ-Base, and SPD3 for Crypt benchmark (chunked version) on 1-16 threads

Table 8.4 shows that the relative slowdowns for Eraser and FastTrack are much larger than those for SPD3. On average (geometric mean), the slowdown for SPD3 relative to HJ-base is  $2.70\times$  while that for Eraser and FastTrack are  $11.21\times$  and  $13.87\times$  respectively relative to RR-base. There is also a large variation. While the slowdowns are within a factor of 2 for SOR, there is more than a  $60\times$  gap in slowdowns for Crypt and quite a significant difference for LUFact, MonteCarlo, and SparseMat-Mult as well. The slowdown for SPD3 on MolDyn is larger than the slowdowns for Eraser and FastTrack because the baseline for SPD3 is more than  $4\times$  faster than the baseline for Eraser and FastTrack. For FastTrack, these slowdowns are consistent with the fact that certain data access patterns (notably, shared reads) can lead to large overheads because they prevent the use of optimized versions of vector clocks.

For the case with the largest gap in Table 8.4 (Crypt), Figure 8.7 shows the



slowdown (scaled execution time) of RR-Base, Eraser, FastTrack, HJ-Base, and SPD3 for the chunked version of the Crypt benchmark on 1-16 threads relative to the 16-thread RR-Base execution time. In this benchmark, RR-Base is the fastest for 16 threads as expected. The execution time of HJ-Base is  $1.9\times$  slower than RR-Base in the 1-thread case and  $1.6\times$  slower than RR-Base in the 16-thread case. Similarly, the execution time of SPD3 version is also very close; it is  $4.2\times$  slower in the 1-thread case and  $3\times$  slower in the 16-thread case. The execution time of Eraser and FastTrack are  $13.7\times$  and  $16.6\times$  slower than RR-Base in the 1-thread case but they increase to more than  $100\times$  for 8-threads and 16-threads. This example shows that for some programs the performance overheads for Eraser and FastTrack can increase dramatically with the number of threads (cores).

### Memory Usage Comparison

We now compare the memory usages of the Eraser, FastTrack and SPD3 algorithms on the coarse-grained JGF benchmarks. Again, the baseline for Eraser and FastTrack was RR-Base and the baseline for SPD3 was HJ-Base. To obtain a coarse estimation of the memory used, we used the *-verbose:gc* option in the JVM and picked the maximum heap memory used over all the GC executions in a single JVM instance. All three instrumented versions trigger GC frequently, so this is a reasonable estimate of the memory overhead.

Table 8.5 shows the estimated memory usage of these three algorithms and their baselines for JGF benchmarks on 16 threads. The table shows that the memory usage of HJ-Base is lower than that of RR-Base in all the benchmarks except Series. In all cases, the memory usage is lower for SPD3, compared to Eraser and FastTrack with significant variation in the gaps. The memory usage of Crypt with SPD3 is quite

Table 8.5 : Peak heap memory usage of RR-Base, Eraser, FastTrack, HJ-Base, and SPD3 for JGF benchmarks on 16 threads. For benchmarks marked with \*, race-free versions were used for SPD3 but the original versions were used for Eraser and FastTrack.

Benchmark	Memory (in MB)				
	RR-Base	Eraser	FastTrack	HJ-Base	SPD3
Crypt	209	8539	8535	149	6009
LUFact*	80	1790	2455	47	203
MolDyn*	382	1048	1040	9	35
MonteCarlo	1771	9316	9292	557	584
RayTracer*	1106	4475	4466	43	88
Series	80	1067	1062	162	177
SOR*	81	1161	1551	47	202
SparseMatMult	225	2120	2171	88	714

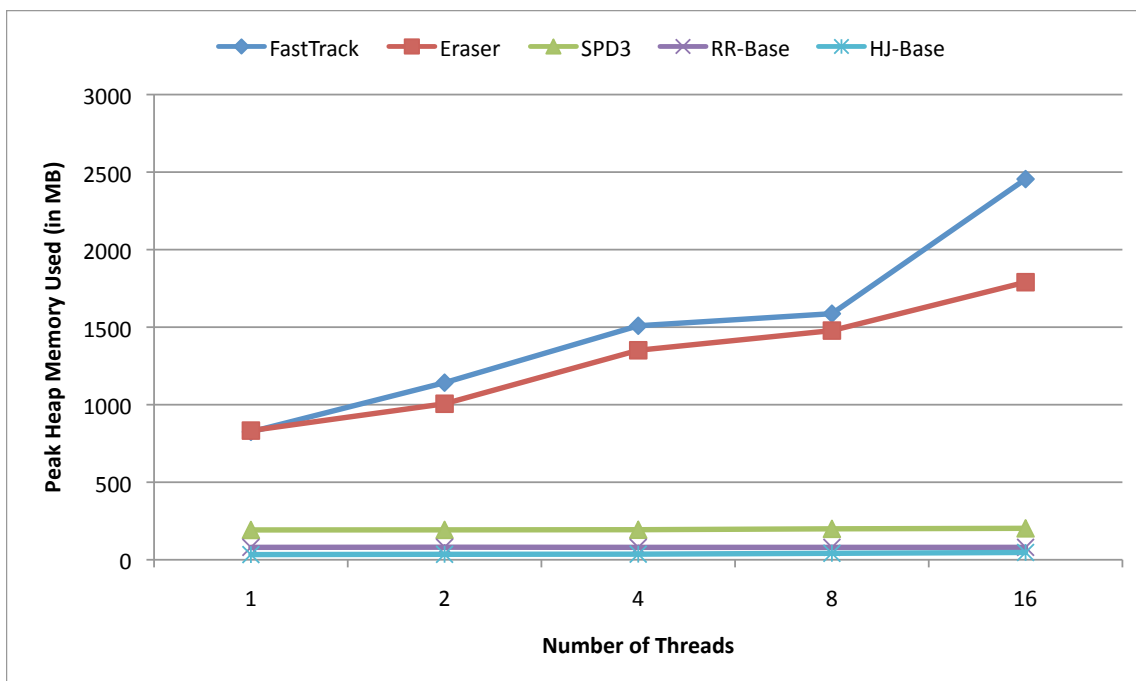


Figure 8.8 : Estimated heap memory usage (in MB) of RR-Base, Eraser, FastTrack, HJ-Base, and SPD3 for LUFact benchmark

high because the benchmark has arrays of size 20 million and our algorithm maintains shadow locations for all elements of these arrays. But the memory used by SPD3 for Crypt is still less than that of Eraser and FastTrack. The high memory usage for Eraser and FastTrack is not surprising because Eraser has to maintain all the locks held while accessing a particular location, and FastTrack's vector clocks may grow linearly in the number of threads in the worst case.

For one of the benchmarks in Table 8.5 (LUFact), Figure 8.8 shows the estimated memory usage of the three algorithms and their baselines as a function of the number of threads/cores used. Note that both the baselines (RR-Base and HJ-Base) are very close. While the estimated heap usage of RR-Base remains constant at  $80M$ , the estimated usage of HJ-Base varies from  $33M$  to  $47M$  as we go from 1 thread to 16 threads. The estimated heap usage of SPD3 is about  $6\times$  larger than HJ-Base: it varies

between  $192M$  and  $203M$  across 16 threads. The estimated heap usage of Eraser increases from  $833M$  for 1 thread to  $1790M$  for 16 threads ( $2.1\times$  increase). Similarly, the estimated heap usage of FastTrack increases from  $825M$  for 1 thread to  $2455M$  for 16 threads ( $3\times$  increase). This clearly shows the increase in the memory usage for Eraser and FastTrack as we increase the number of threads for this benchmark.

## Chapter 9

### Related Work

Data races are a major cause of incorrect and unintended behaviors in parallel programs. Since data races typically occur only in a few of the possible interleavings, it is extremely difficult to detect and reproduce them. But data race detection is important because it helps improve productivity. Consequently, there has been a lot of prior work on data race detection. While there has been a lot of work on data race detection in software, there has also been some hardware-based data race detectors. Software data race detectors are classified into *static* and *dynamic* race detectors based on the type of the analysis they perform to detect data races in a given program. There has also been some approaches that combine *static* and *dynamic* analysis (*hybrid* techniques) to detect data races in a given program [58, 59].

This chapter discusses some related work on data race detection. First, we describe some related dynamic data race detection techniques from the past. Then, we present some past work on static analysis for data race detection and avoidance. Then, we describe some existing hardware-based data race detection. Finally, we describe static and dynamic analysis for checking determinism in parallel programs.

#### 9.1 Dynamic Data Race Detection

Dynamic data race detection involves analyzing an execution of the input parallel program to detect data races at runtime. In this section, we describe some related

work on dynamic data race detection. First, we start with the dynamic data race detection algorithms based on locksets. Then, we describe the happens-before based dynamic data race detection algorithms.

### 9.1.1 Lockset based Algorithms

Eraser [3] is a dynamic data race detector that targets lock-based synchronization used in multithreaded programs. Eraser performs a lockset analysis to ensure that the program strictly follows a *locking discipline*. A simple locking discipline is that each variable shared between threads is protected by a particular lock. When the program being monitored does not follow the locking discipline, Eraser flags a data race. While this approach identifies data races that may occur in schedules other than the one that is examined, a drawback is that it may have false positives. This is because there could be regions in the program where accesses to a variable are unprotected by locks, like initialization, and also the lock protecting a variable could change over time. Another major drawback of this approach is that it only handles lock-based synchronization. It does not handle other forms of synchronizations like fork-join, barriers, post-wait, etc., which are used extensively in multithreaded programs.

Choi et.al. [59] presented a dynamic data race detector for multithreaded object-oriented programs. Their data race detection technique is designed for locks like Eraser. But unlike Eraser, they extend their detector to support thread start and join operations by introducing an ownership model. Though it reduces the number of false positives considerably compared to Eraser, it is still not precise. The advantage of this technique is that it reduces the overhead of race detection to under 50% by applying static and dynamic optimization techniques.

There are other lockset based data race detectors [60, 61] that improve the Eraser

algorithm to reduce the number of false positives. Though these data race detectors reduce the space and time overhead of race detection, they are still not precise, i.e., they may have false positives. The main limitation of these algorithms is due to the fact that the locksets associated with variables only become smaller.

Goldilocks [38, 62] eliminates the limitation in earlier algorithms by allowing the locksets associated with variables to grow. It is a precise lockset-based dynamic data race detection algorithm. It also computes a precise happens-before relation (defined in the next section) using the locksets. Since this algorithm does not use any expensive data structure to compute the happens-before relation, it is expected to be much faster than happens-before based data race detectors. But it has been shown subsequently that there are happens-before data race detectors, like FastTrack [1], that are faster than Goldilocks. In this thesis, we show that our SPD3 algorithm performs much better than FastTrack for structured parallel programs. In Goldilocks, a `DataRaceException` is thrown just before the access causing the data race is executed. This allows the programmers to handle data races explicitly in their programs. It also guarantees that when an execution does not throw any `DataRaceException` it is sequentially consistent.

### 9.1.2 Happens-Before based Algorithms

The *happens-before* relation defined by Lamport [63] has been used extensively for data race detection. The *happens-before* relation defines a partial order among all the operations in all the threads of a parallel program. All the operations within a thread are ordered by program order. Two operations in two different threads are ordered if there is some synchronization operation that ensures that one of these operations is guaranteed to complete before the other one begins. In this case, the first operation

*happens-before* the second operation. There has been a lot of work on dynamic data race detection based on the *happens-before* relation [4, 36, 37, 64, 1].

The *may-happen-in-parallel* relation that we use in our SPD3 algorithm is the inverse of the *happens-before* relation. In other words, when two events have no *happens-before* relation between them, they *may-happen-in-parallel* with each other.

Schonberg [4] presented one of the earliest dynamic data race detection algorithm for nested fork-join and synchronization operations that uses *happens-before* relation to detect data races. In this algorithm, a shared variable set is associated with each sequential block in every task. There is also a concurrency list associated with each shared variable set which keeps track of the concurrent shared variable sets that will complete at a later time. The algorithm detects anomalies by comparing complete concurrent shared variable sets at each time step. The space required to store read information in the shared variable sets is bounded by  $V \times N$ , where  $V$  is the number of variables being monitored and  $N$  is the number of execution threads.<sup>1</sup> One of the limitations of this work is that this space requirement increases linearly with the number of threads the program is executed on. This is clearly not useful in practice because its memory requirement will exceed the available memory when executed on a large number of threads. Also, this algorithm's guarantees apply only to the schedule of the program that was examined, which is again not very useful. Another limitation is that since access anomalies are detected at synchronization points, it does not identify the actual read and write operations involved in the data races and hence, it does not help improve the productivity.

---

<sup>1</sup>If  $N$  refers to the maximum number of threads possible in all executions of a program for a given input, then this algorithm can guarantee data race freedom for all executions of the program for that input. If not, then this guarantee will not hold.



Offset-Span (OS) labeling [37] is an optimized version of the English-Hebrew (EH) labeling technique [36] for detecting data races. The idea behind both these techniques is to attach a label to every thread in the program and use these labels to check if two threads can execute concurrently. They also maintain the access history for every shared variable that is monitored which is then used to check for conflicts. The length of the labels associated with each thread can grow arbitrarily long in EH labeling,<sup>2</sup> whereas the length of the labels in OS labeling is bounded by the maximum nesting depth of fork-join in the program. While the EH labeling technique needs an access history of size equal to the number of threads for every monitored variable in the program, the OS labeling technique only needs constant size to store access history. The OS labeling algorithm works only for the nested fork-join constructs. The computations generated by async-finish constructs are a strict superset of those generated by nested fork-join constructs. Hence, this algorithm cannot be directly applied to programs with async-finish. Also, the OS labeling algorithm has been evaluated only on sequential executions [58] and the performance of this algorithm on parallel executions is not clear.

LiteRace is a light-weight dynamic data race detector that reduces the overhead of race detection by sampling [65]. It maintains a happens-before relation to precisely identify data races. The novelty of the approach is in the use of sampling to reduce the race detection overhead. They sample regions of the code that are more likely to result in a data race at a higher rate than those regions of the code that are less likely to result in a data race. While this technique does not report any false positives, it may miss data races.

---

<sup>2</sup>Note that the length of the labels is bounded by the maximum nesting level of fork-join in EH labeling in the presence of an effective heuristic as reported in [36]

## Algorithms using Vector Clocks

A vector clock [66] is a data structure that is used to maintain the *happens-before* relation between operations during program execution. Every thread's execution is considered as a sequence of time-steps split according to the synchronization operations performed by that thread. Every thread maintains a vector (or an array), called a vector clock, of numbers that represent the time-steps of all the other threads in the program. The number in position  $q$  in the vector clock of thread  $p$  represents the latest time-step of thread  $q$  after which there was a synchronization between  $p$  and  $q$ . In other words, the number in position  $q$  refers to the most recent time-step of thread  $q$  that is *visible* to thread  $p$ . A basic data race detector using vector clocks would maintain two vector clocks for each memory location, one for read and one for write, and compare the vector clocks to check for data races on every access.

DJIT [67] is one of earliest dynamic data race detectors that use vector clocks to detect data races in distributed shared memory (DSM) systems. DJIT is designed to detect data races for global synchronization constructs like barriers and two-way synchronization constructs like locks. It can also be extended to support k-way synchronization constructs. The disadvantages of DJIT is that it works only on a sequentially consistent system and reports only the first data race found in an execution. The DJIT<sup>+</sup> algorithm [68] overcomes these restrictions.

MultiRace [68, 69] combines DJIT<sup>+</sup> and an improved lockset algorithm to detect data races. This combination is effective because they complement each other. While DJIT<sup>+</sup> detects only those data races that happen in the examined execution, the improved lockset algorithm can detect data races across all executions of the program for an input. While the improved lockset algorithm reports false positives, DJIT<sup>+</sup> is precise. In this algorithm, the granularity of data race detection changes dynam-

ically, and can also be controlled by the users. Since the race detection granularity varies, this algorithm can generate false positives when detecting data races at coarse granularities.

Perković and Keleher [70] present an on-the-fly data race detector that detects data races in programs executing on a distributed shared memory system. The main idea in this technique is to utilize the mechanisms that support lazy release consistency to detect data races. The implementations of lazy release consistency models maintain enough information to detect data races by identifying concurrent accesses in constant time. This technique uses vector timestamps [66] to check if two “intervals” are concurrent. The advantage of this technique is that it does not require any support from the compiler. But the downside is that it detects data races that occur in the particular examined execution only.

RaceTrack [71] is a dynamic data race detection algorithm that combines lockset and happens-before analysis. It uses vector clocks to maintain the happens-before relation between different operations in the program. While RaceTrack may miss data races, it adaptively focuses on areas in the program that are more suspicious to report more accurate data races with much less overhead. It does this by estimating the current number of parallel accesses to every memory location and resetting the access history of those memory locations that are not expected to have any data races, thereby reducing the space and time overhead associated with those memory locations. In summary, RaceTrack reduces the overhead of data race detection by adaptively changing the granularity of race detection and the technique used for race detection.

The FastTrack algorithm [1] is a precise dynamic data race detection algorithm that can run in parallel and handles classic (unstructured) fork-join programs with

locks and barriers. This algorithm uses vector-clocks to maintain the access history for every memory location that is monitored in the program. A key drawback of FastTrack is that, in the worst-case, it requires  $O(n)$  space per instrumented memory location, where  $n$  is the number of threads. Note that this drawback applies to all vector-clock based race detection techniques. This means that the algorithm can only be used with a small number of threads. Increasing the number of threads can quickly cause space overheads and slowdowns that render the algorithm impractical. FastTrack applies an optimization to reduce the overhead by requiring only  $O(1)$  space per memory location when it is local to a thread, but for memory locations that are read shared, the algorithm requires  $O(n)$  space. Unfortunately, in domains where structured parallelism dominates, programs typically use a massive number of lightweight tasks (e.g. consider a parallel-for loop on a GPU) and often the parallel tasks share read-only data.

## Summary

We now summarize and compare our race detection algorithms, ESP-bags and SPD3, with some of the past work discussed so far that are closely related. Recall the discussion from Section 2.3.4 about the race detection algorithms that were designed for Cilk. The SP-bags algorithm was designed to detect data races in Cilk programs with `spawn` and `sync` constructs. It requires that the input program is executed sequentially. The SP-hybrid algorithm is a parallel race detection algorithm for `spawn` and `sync` constructs of Cilk.

Table 9.1 compares five of the past works on dynamic data race detection with our ESP-bags and SPD3 algorithms. The past algorithms that we compare with are *On The Fly Detection of Access Anomalies* (OTFDAA) [4], Offset-Span Labeling [37],

Table 9.1 : A comparison of related work. OTFDAA refers to “On The Fly Detection of Access Anomalies”,  $n$  refers to the number of threads executing the program and  $N$  refers to the maximum logical concurrency in the program.

	<b>Target Language</b>	<b>Space Overhead per Memory Location<sup>a</sup></b>	<b>Guarantees</b>	<b>Empirical Evaluation</b>	<b>Parallel Execution</b>	<b>Scheduling Dependency</b>
OTFDAA	Nested Fork-Join	$O(n)$	Per-Schedule	No	Yes	No
Offset-Span	Nested Fork-Join	$O(1)$	Per-Input	Minimal	Yes	No
SP-bags	Spawn-Sync	$O(1)$	Per-Input	Yes	No	No
SP-hybrid	Spawn-Sync	$O(1)$	Per-Input	No	Yes	Yes
FastTrack	Fork-Join	$O(N)$	Per-Input	Yes	Yes	No
ESP-bags	Async-Finish	$O(1)$	Per-Input	Yes	No	No
SPD3	Async-Finish	$O(1)$	Per-Input	Yes	Yes	No

---

<sup>a</sup>A discussion about the full space overhead of the SPD3 algorithm can be found in Section 4.1.5

SP-bags [5], SP-hybrid [2], and FastTrack [1]. We focus on six properties of dynamic race detection algorithms in this comparison.

While some algorithms target nested fork-join and spawn-sync constructs for race detection, our algorithms work for a more relaxed concurrency model, `async-finish`. FastTrack supports the most relaxed model, the unstructured fork-join frameworks. The worst-case space overhead per memory location in most of the algorithms, including ESP-bags and SPD3, are constant. The worst-case space overhead per memory location is linear in the number of threads executing the program in OTFDAA and is linear in the maximum logical concurrency of the program in FastTrack, which is the major downside of these algorithms. All the algorithms except OTFDAA have per-input guarantees.

There has been no empirical evaluation for OTFDAA and SP-hybrid. The empirical evaluation of Offset-Span labeling has been done only on sequential executions [58]. The SP-bags and ESP-bags algorithms need to execute the input program sequentially, but the other algorithms can execute the program in parallel. While the SP-hybrid algorithm is closely tied with the work-stealing scheduler, no other algorithm depends on a scheduling technique.

### Other Dynamic Data Race Detectors

Veeraraghavan et.al. [72] introduce a new approach to detecting data races called *outcome-based data race detection*. In this approach, when a parallel program executes, multiple replicas of the program are executed such that two of these replicas follow complementary schedules. Two replicas with complementary schedules are constructed such that conflicting operations execute in opposite order in these two replicas with high probability. By comparing the states of replicas executed with

complementary schedules, data races can be detected. This technique has very low overhead compared to existing dynamic data race detectors but it may miss data races. This technique also includes a mechanism to “survive” data races by choosing the replica that is expected to be correct when a data race occurs.

Process races are those that occur when shared operating systems resources are accessed in parallel by different processes without any synchronization. RacePro is an offline technique to detect process races on real world applications [73]. In a deployed system, RacePro records the necessary information during execution and then analyzes the recorded information to detect process races offline. It has various methods to reduce the overhead of recording all the necessary information during execution. It also has some methods to validate the detected process races, thereby reducing the number of false positives and false negatives.

## 9.2 Static Analysis for Data Race Detection and Avoidance

Static data race detection involves analyzing the input parallel program statically at compile-time to detect data races. Static data race detection is attractive because it can reason about the program completely unlike dynamic analysis, where the reasoning holds only for the particular input considered. The major drawback of static race detection is that it generates large number of false positives. We now describe two approaches that are commonly used in static race detection and also briefly describe few specific techniques in static race detection.

One approach to static race detection is to explore all possible states of a parallel program beginning with the initial state that correspond to the input to the program. Whenever a state with conflicting accesses is reached, a data race is signaled between these conflicting accesses. Some techniques explore these states by building them

explicitly [74, 75, 76, 77, 78, 79] and others build abstractions to represent the state space [80, 81]. The problem with this approach is that the number of possible states could be exponential even for simple programs. In other words, both the space and time requirement could be exponential for this approach.

The other approach to static race detection involves performing data flow analysis to compute the ordering between memory accesses in a program and reporting data races when there are conflicting accesses that are not ordered [82, 83, 84, 85, 86, 87, 88]. The advantage of this approach is that it usually takes polynomial space and time. But the downside is that, when no ordering between two memory accesses can be proved, they are said to result in a data race. Thus, these approaches are conservative in nature which may lead to false positives.

The type based data race detection for Java [89] identifies data races in a program by combining user annotations and a type based analysis. This work focuses on lock-based synchronization by identifying the lock protecting every field in the program and tracking them to check if every access to that field happens when that lock is held. Loginov et.al. combine inter-thread control flow analysis and points-to analysis [90] to statically detect data races in multithreaded object-oriented programs. Aiken and Gay present a static analysis to detect data races in SPMD programs [29]. SPMD programs perform synchronization using constructs like barriers and hence, this work focuses on barrier synchronization.

One of the earliest work on avoiding data races was using a *monitor*, which was introduced by Hoare [91]. A monitor associates a group of procedures and a lock with every shared variable. The lock is meant to guard the variable. The shared variable is accessible only within the procedures associated with it. The lock associated with the variable is acquired at the entry to these procedures and released at their exits. With



this set up, there is no way for a data race to occur because all accesses to the variable are guarded by a lock. Though it gives static guarantees that the program is free of data races, a drawback is that it can only handle global variables. Specifically, it cannot handle dynamically allocated memory locations. Also, it serializes all accesses to a variable even if it is read-shared by multiple threads.

Young and Taylor [92] present a technique that combines static analysis for data race detection with symbolic execution to reduce the false positives generated by static analysis. In this technique, symbolic execution is used to discover more information about execution paths, which is then used to prune the results of static race detection. Also, this technique uses static analysis to select the paths for symbolic execution and hence, it does not incur the complete overhead of symbolic execution as well. Earlier, Taylor showed that the problem of computing the ordering between accesses in Ada programs is NP-complete [78, 93].

### 9.2.1 Static Analysis to Improve Dynamic Race Detectors

Static analysis can be used to improve the performance of dynamic data race detectors. Static analysis can prove that some regions of a parallel program will never result in a data race and hence, these regions of the program need not be monitored for races during dynamic data race detection [94, 33, 95, 79].

In chapter 7, we present some static optimizations to eliminate the redundant instrumentations for race detection thereby reducing the overhead of the algorithms. These static optimizations are similar to compile-time analysis used by Mellor-Crummey [58]. His work uses a dependence graph that contains edges for all data dependences to eliminate instrumentations for variable references that are not part of these data dependences. His technique is applicable for loop carried data depen-

dences across parallel loops and also for data dependences across parallel blocks of code. In our approach, we concentrate on the instrumentations within a particular task and try to eliminate redundant instrumentations for memory locations which are guaranteed to have already been instrumented in that task.

Harris et.al. [96] present an implementation of Software Transactional Memory (STM) that improves the performance of `atomic` blocks in a parallel program. They present some compiler optimizations to eliminate the need for logging on some variables and regions of the program because they are redundant. These optimizations are similar to our static optimizations to reduce the runtime overhead of our race detection algorithms.

Parallel programs sometimes use *ad hoc synchronizations* between different threads for flexibility or performance reasons. This kind of synchronization makes it hard for tools like data race detectors to detect bugs in parallel programs. SyncFinder is a tool that performs static analysis to automatically identify such ad hoc synchronizations in parallel programs [97]. When ad hoc synchronizations in parallel programs are annotated, debugging tools like data race detectors can utilize this extra information to improve their performance. Specifically, data race detectors can reduce their false positives by using the knowledge of ad hoc synchronizations. Similarly, SyncFinder can be used to improve the accuracy of other concurrency bug detectors.

### 9.3 Data Race Detection in Hardware

Min and Choi present a cache-based approach to data race detection [98]. This technique uses the underlying cache coherence protocol to minimize the overhead of data race detection. The execution of the program proceeds without any data race detection until there is a cache miss on a read access or an invalidation on a write

access. When one such event is observed data race detection is enabled.

Prvulovic and Torrellas propose the idea that the rollback capabilities of thread-level speculation can be used to re-execute a buggy piece of code until the bug is characterized [99]. Their ReEnact tool uses the thread-level speculation mechanisms to detect, characterize, and even repair data races automatically.

HARD is a hardware implementation of the lockset algorithm for race detection [100]. The candidate sets used in the lockset algorithm is stored in hardware bloom filters. It improves upon the lockset algorithm by handling barriers. This helps in reducing the number of false positives reported.

Nistor et.al. present a hardware-based race detector that detects data races during systematic testing of parallel programs [101]. Their tool, Light64, computes a hash of program execution history during systematic testing. It detects data races by comparing the hashes of two executions of the program. It depends on the fact that two executions in which the accesses causing a data race are reversed will mostly likely have different execution histories. The advantage of this technique is that it does not report any false positives.

SigRace is a hardware-based data race detector that uses hardware address signatures [102]. As a thread executes on a processor, the addresses of the locations accessed by that thread are encoded in a signature. Then, a hardware module computes the intersection of signatures from different threads (processors). If the intersection of signatures from any two threads is not null, then there may have been a data race. This technique eliminates the limitations due to the reliance on caches in the earlier hardware-based race detectors and also reduces the overhead of race detection a lot.

A recent work on demand driven software race detection uses hardware performance counters to enable race detection as needed [103]. It monitors the cache events

that are indicative of shared memory accesses across threads. It enables the software race detection only when such events occur. The downside of this approach is that it may miss data races because some shared memory accesses may not lead to any cache events.

## 9.4 Determinism Checking

Data races and determinism are very closely related. This is because data race freedom may imply determinism in some cases. We now describe some techniques to check for or guarantee determinism in parallel programs.

Burnim and Sen present an assertion framework to specify deterministic regions of a parallel program [104]. The programmers specify assertions in a parallel program that could involve program states from different executions of the program. The assertions can be viewed as pre-conditions and post-conditions that span multiple executions of a program. The assertion framework executes the program for some inputs and verifies that the assertions are satisfied during these executions. While this technique identifies non-deterministic behavior if the executions monitored exhibit non-determinism, it can not provide any guarantees about determinism otherwise.

Sadowski et.al. present a dynamic analysis to verify conflict freedom and external serializability of multithreaded programs [105]. They define determinism as a combination of conflict freedom and external serializability properties. Though they can prove determinism of multithreaded programs, the main drawback of their approach is that they use vector clocks to model the happens-before relation. Since the space overhead due to vector clocks is linear in the number of threads for every memory location, this cannot be applied to programs with large number of threads.

Vechev et.al present a static analysis to verify determinism for structured parallel

programs [27]. They identify fragments in a parallel program that may execute in parallel and prove that these fragments access independent memory locations. While this technique can prove determinism in some of the programs, they can not handle programs where the memory locations accessed are based on the input specified. This is an inherent drawback of static analysis.

Dthreads is an efficient multithreading system designed to provide deterministic executions for multithreaded C/C++ programs [106]. In this technique, every thread is implemented as a separate process. Since every process has its own private address space, there is no interaction between different threads except at synchronization points. At synchronization points, the updates to the shared memory from every thread (process) are applied in sequence. Thus, a deterministic execution is guaranteed even in the presence of data races.

Deterministic Multithreading (DMT) is another technique to preserve determinism in parallel program executions. Tern is a deterministic multithreading system that performs schedule memoization to deterministically execute parallel programs [107]. It maintains a cache of inputs and their corresponding working schedules. When the program execution begins with an input, it checks if the input is present in the cache. If there is a match, then the corresponding schedule is used to execute the program. The execution will be deterministic because the same schedule is always used to execute the program for a given input.

Peregrine [108] is an efficient deterministic multithreading system which records the trace during the first execution of a parallel program on an input. It then computes a *hybrid* schedule of the program for that input, i.e., a schedule that contains a total-order on all the synchronization events in the program execution and an ordering of all the memory events in the regions of the program that contain data races. Then,

the system uses this schedule to deterministically execute the program on all “similar” inputs.

## Chapter 10

### Conclusions and Future Work

While increasing emphasis is being placed on parallel programming to utilize the processing power available in parallel processors, writing parallel programs still remains a challenge. The main cause for this difficulty is the need to reason about large numbers of interleavings of statements in a parallel program. Since data races are a major source of bugs in parallel programs, data race detection plays an important role in improving the productivity of programmers. Structured parallelism is an attractive and emerging trend in parallel programming. Though there has been a lot of work on data race detection in the past, existing dynamic data race detectors suffer from a number of limitations due to which it has not been effective to apply them to structured parallel programs.

In this dissertation, we introduce two new dynamic data race detection algorithms for structured parallel programs. The first is the ESP-bags algorithm that detects data races in HJ programs with `async`, `finish`, `isolated`, and a restricted form of `future`, by executing the program sequentially. This algorithm is a generalization of the SP-bags algorithm that was designed for data race detection in Cilk programs with `spawn` and `sync` constructs. Though this algorithm is sound and precise for a given input, the main drawback of this algorithm is that it requires a sequential execution of the input program.

The second algorithm that we present, called the SPD3 algorithm, can detect data races by executing the input program in parallel. It uses a new data structure called

the Dynamic Program Structure Tree (DPST) to check if two accesses may execute in parallel. The DPST has some nice properties that enable it to be accessed and updated in parallel without any synchronization. The algorithm uses only constant space per memory location irrespective of the number of parallel tasks accessing that location. This algorithm incurs an average slowdown of under  $3\times$  over the original execution time on a suite of 15 benchmarks executing on a 16-core smp system. This is in contrast with an average slowdown of over  $10\times$  from past work (Eraser, FastTrack).

With these algorithms, we show that structured parallelism can enable simpler analysis of concurrency. Specifically, we show that structured parallelism can enable efficient dynamic data race detectors that are useful in practice. We believe that our parallel SPD3 algorithm is the first practical dynamic data race detection algorithm for async-finish parallel programs, that can execute the input program in parallel and use constant space per memory location. This algorithm provides a promising foundation for future data race detection tools. It takes us closer to our goal of designing dynamic data race detectors that can be “always-on” while developing parallel applications.

## Future Work

This dissertation on dynamic data race detection paves the way for a variety of future research directions. Some of them are listed below.

1. The next logical step with the SPD3 algorithm is to extend it to support other parallel constructs like Phasers [16], Data Driven Futures [109], and Actors [110].
2. Since our data race detection algorithms are dynamic, their guarantees hold only for the input that is considered. We could augment our algorithms with



some static analysis to extend the guarantees beyond one input, say to a class of inputs.

3. The race detection algorithms presented in this thesis can also be used to synthesize the synchronization needed in the program. We could start with a program where the parallelism is specified (e.g., using `async` statements) but is devoid of any synchronization (e.g., `finish` statements). In our model, such a program would consist of `asyncs` and no `finishes`. Using our race detection algorithm on such programs, whenever a data race occurs, an appropriate synchronization construct can be placed so as to prevent the data race. This way all the synchronization in the program can be automatically generated and the program will also be free of data races.
4. The work on Permission Regions in HJ [111, 112] introduces the notion of permissions for read / write of memory locations for regions of code. It reports a violation when there are two conflicting permissions that are acquired by tasks executing in parallel. Currently, this work reports a violation only if that happens in the execution that is monitored. We can use the DPST based approach to extend the guarantees beyond the monitored execution and also enable the detection of high level data races.
5. The DPST introduced in the SPD3 algorithm can be used to check for *determinism* in parallel programs. The concurrency libraries used by parallel programs often consist of conflicting methods, i.e., methods which when executed in parallel lead to non-determinism. We can model these conflicting methods in the same form as reads, writes, isolated-reads, and isolated-writes in our SPD3 algorithm. Then, we can use the DPST to show that there are no calls to con-

flicting methods in parallel thereby proving determinism of the program for a given input.

## Bibliography

- [1] C. Flanagan and S. N. Freund, “FastTrack: efficient and precise dynamic race detection,” in *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pp. 121–133, ACM, 2009.
- [2] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson, “On-the-Fly Maintenance of Series-Parallel Relationships in Fork-Join Multithreaded Programs,” in *Proceedings of the Sixteenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, (Barcelona, Spain), pp. 133–144, June 27–30 2004.
- [3] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: a dynamic data race detector for multithreaded programs,” *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.
- [4] E. Schonberg, “On-the-fly detection of access anomalies,” in *In Proceedings of the SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pp. 285–297, 1998.
- [5] M. Feng and C. E. Leiserson, “Efficient detection of determinacy races in Cilk programs,” in *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pp. 1–11, ACM, 1997.
- [6] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark, “Detecting Data Races in Cilk Programs that Use Locks,” in *Proceedings of the Tenth*

- Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*, (Puerto Vallarta, Mexico), pp. 298–309, June 28–July 2 1998.
- [7] 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 *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, pp. 207–216, Oct. 1995.
- [8] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar, “X10: An object-oriented approach to non-uniform cluster computing,” in *OOPSLA 2005 Onward! Track*, 2005.
- [9] V. Cave, J. Zhao, J. Shirako, and V. Sarkar, “Habanero-Java: the New Adventures of Old X10 ,” in *PPPJ'11*, 2011.
- [10] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, “Efficient data race detection for async-finish parallelism,” in *Proceedings of the First international conference on Runtime verification, RV'10*, (Berlin, Heidelberg), pp. 368–383, Springer-Verlag, 2010.
- [11] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, “Scalable and precise dynamic datarace detection for structured parallelism,” in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation, PLDI '12*, (New York, NY, USA), pp. 531–542, ACM, 2012.
- [12] “OpenMP Application Program Interface v 3.0,” 2008.
- [13] M. Frigo, C. E. Leiserson, and K. H. Randall, “The implementation of the Cilk-5 multithreaded language,” in *PLDI'98*, (NY, USA), pp. 212–223, ACM, 1998.

- [14] “UPC Language Specifications, version 1.1, 2003..” <http://www.gwu.edu/upc/documentation.html>.
- [15] R. W. Numrich and J. Reid, “Co-array fortran for parallel programming,” *ACM FORTRAN FORUM*, vol. 17, no. 2, pp. 1–31, 1998.
- [16] 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 *Proceedings of the 22nd annual international conference on Supercomputing, ICS '08*, (New York, NY, USA), pp. 277–288, ACM, 2008.
- [17] J. K. Lee and J. Palsberg, “Featherweight X10: a core calculus for async-finish parallelism,” in *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel computing*, pp. 25–36, ACM, 2010.
- [18] J. R. Larus and R. Rajwar, *Transactional Memory*. Morgan and Claypool, 2006.
- [19] Y. Guo, R. Barik, R. Raman, and V. Sarkar, “Work-first and help-first scheduling policies for async-finish task parallelism,” in *IPDPS '09: Proceedings of the International Symposium on Parallel&Distributed Processing*, pp. 1–12, IEEE Computer Society, 2009.
- [20] R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.
- [21] S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. Yelick, “Deadlock-free scheduling of X10 computations with bounded re-

- sources,” in *SPAA '07: Proceedings of the 19th symposium on Parallel algorithms and architectures*, pp. 229–240, ACM, 2007.
- [22] M. B. Dwyer and L. A. Clarke, “Data flow analysis for verifying properties of concurrent programs,” in *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '94, (New York, NY, USA), pp. 62–75, ACM, 1994.
- [23] D. Engler and K. Ashcraft, “RacerX: effective, static detection of race conditions and deadlocks,” in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, (New York, NY, USA), pp. 237–252, ACM, 2003.
- [24] M. Abadi, C. Flanagan, and S. N. Freund, “Types for safe locking: Static race detection for java,” *ACM Transactions on Programming Languages and Systems*, vol. 28, p. 2006, 2006.
- [25] M. Naik, A. Aiken, and J. Whaley, “Effective static race detection for Java,” in *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pp. 308–319, ACM, 2006.
- [26] J. W. Vounq, R. Jhala, and S. Lerner, “Relay: static race detection on millions of lines of code,” in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, (New York, NY, USA), pp. 205–214, ACM, 2007.
- [27] M. Vechev, E. Yahav, R. Raman, and V. Sarkar, “Automatic verification of determinism for structured parallel programs,” in *Proceedings of the 17th international conference on Static analysis*, SAS'10, (Berlin, Heidelberg), pp. 455–471,

Springer-Verlag, 2010.

- [28] R. Agarwal and S. D. Stoller, “Type inference for parameterized race-free Java,” in *In Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, pp. 149–160, Springer-Verlag, 2004.
- [29] A. Aiken and D. Gay, “Barrier inference,” in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’98, (New York, NY, USA), pp. 342–354, ACM, 1998.
- [30] C. Boyapati and M. Rinard, “A parameterized type system for race-free Java programs,” in *OOPSLA ’01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 56–69, ACM, 2001.
- [31] D. Grossman, “Type-safe multithreading in Cyclone,” in *TLDI ’03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pp. 13–25, ACM, 2003.
- [32] D. P. Helmbold, C. E. McDowell, and J.-Z. Wang, “Analyzing traces with anonymous synchronization,” in *ICPP (2)*, pp. 70–77, 1990.
- [33] P. A. Emrath, S. Chosh, and D. A. Padua, “Event synchronization analysis for debugging parallel programs,” in *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Supercomputing ’89, (New York, NY, USA), pp. 580–588, ACM, 1989.
- [34] B. P. Miller and J.-D. Choi, “A mechanism for efficient debugging of parallel programs,” in *Proceedings of the ACM SIGPLAN 1988 conference on Program-*

- ming Language design and Implementation*, PLDI '88, (New York, NY, USA), pp. 135–144, ACM, 1988.
- [35] R. H. B. Netzer and S. Ghosh, “Efficient race condition detection for shared-memory programs with post/wait synchronization,” in *ICPP (2)*, pp. 242–246, 1992.
- [36] A. Dinning and E. Schonberg, “An empirical comparison of monitoring algorithms for access anomaly detection,” in *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, PPOPP '90, (New York, NY, USA), pp. 1–10, ACM, 1990.
- [37] J. Mellor-Crummey, “On-the-fly detection of data races for programs with nested fork-join parallelism,” in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, (New York, NY, USA), pp. 24–33, ACM, 1991.
- [38] T. Elmas, S. Qadeer, and S. Tasiran, “Goldilocks: a race and transaction-aware Java runtime,” in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, (New York, NY, USA), pp. 245–255, ACM, 2007.
- [39] T. C. Karunaratna, “Nondeterminator-3: A provably good data-race detector that runs in parallel,” Master’s thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Sept. 2005.
- [40] “Intel Cilk++ Programmer’s Guide.” <http://software.intel.com/en-us/articles/download-intel-cilk-sdk/>.



- [41] R. E. Tarjan, “Efficiency of a good but not linear set union algorithm,” *J. ACM*, vol. 22, no. 2, pp. 215–225, 1975.
- [42] R. E. Tarjan, *Data structures and network algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1983.
- [43] R. Lubliner, J. Zhao, Z. Budimlić, S. Chaudhuri, and V. Sarkar, “Delegated isolation,” in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications, OOPSLA ’11*, (New York, NY, USA), pp. 885–902, ACM, 2011.
- [44] G. J. Narlikar and G. E. Blelloch, “Space-efficient scheduling of nested parallelism,” *ACM Transactions on Programming Languages and Systems*, vol. 21, 1999.
- [45] J. Zhao and V. Sarkar, “Intermediate language extensions for parallelism,” in *VMIL ’11*, pp. 333–334, 2011.
- [46] N. Nystrom, M. R. Clarkson, and A. C. Myers, “Polyglot: an extensible compiler framework for Java,” in *Proceedings of the 12th international conference on Compiler construction, CC’03*, (Berlin, Heidelberg), pp. 138–152, Springer-Verlag, 2003.
- [47] R. Vallée-Rai *et al.*, “Soot - a Java Optimization Framework,” in *Proceedings of CASCON 1999*, pp. 125–135, 1999.
- [48] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to algorithms*. MIT Press, 2009.

- [49] L. Lamport, “Concurrent reading and writing,” *Commun. ACM*, vol. 20, pp. 806–811, November 1977.
- [50] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff, “Escape analysis for Java,” in *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA ’99, (New York, NY, USA), pp. 1–19, ACM, 1999.
- [51] J. Whaley, “Partial method compilation using dynamic profile information,” in *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA ’01, (New York, NY, USA), pp. 166–179, ACM, 2001.
- [52] A. Salcianu and M. Rinard, “Pointer and escape analysis for multithreaded programs,” in *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, PPOPP ’01, (New York, NY, USA), pp. 12–23, ACM, 2001.
- [53] L. A. Smith and J. M. Bull, “A Parallel Java Grande Benchmark Suite,” in *In Supercomputing’01*, p. 8, ACM Press, 2001.
- [54] A. Duran *et al.*, “Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP,” in *ICPP’09*, pp. 124–131, 2009.
- [55] “The Computer Language Benchmarks Game.”  
<http://shootout.alioth.debian.org/>.
- [56] Y. Guo, J. Zhao, V. Cavé, and V. Sarkar, “Slaw: A scalable locality-aware adaptive work-stealing scheduler,” in *IPDPS*, 2010.

- [57] C. Flanagan and S. N. Freund, “The RoadRunner Dynamic Analysis Framework for Concurrent Programs,” in *PASTE’10*, (NY, USA), pp. 1–8, ACM, 2010.
- [58] J. Mellor-Crummey, “Compile-time support for efficient data race detection in shared-memory parallel programs,” in *PADD ’93: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, (New York, NY, USA), pp. 129–139, ACM, 1993.
- [59] J.-D. Choi, K. Lee, A. Loginov, R. O’Callahan, V. Sarkar, and M. Sridharan, “Efficient and precise datarace detection for multithreaded object-oriented programs,” in *PLDI*, pp. 258–269, 2002.
- [60] H. Nishiyama, “Detecting data races using dynamic escape analysis based on read barrier,” in *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2004.
- [61] C. von Praun and T. R. Gross, “Object race detection,” in *OOPSLA ’01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 70–82, ACM, 2001.
- [62] S. T. Tayfun Elmas, Shaz Qadeer, “Goldilocks: Efficiently computing the happens-before relation using locksets,” Tech. Rep. MSR-TR-2006-163, Microsoft Research, 2006.
- [63] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” in *ACM Operating Systems*, 1978.
- [64] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer, “Detecting data races

- on weak memory systems,” in *ISCA '91: Proceedings of the 18th international symposium on Computer architecture*, pp. 234–243, ACM, 1991.
- [65] D. Marino, M. Musuvathi, and S. Narayanasamy, “LiteRace: effective sampling for lightweight data-race detection,” in *PLDI '09: Proceedings of the 2009 conf. on Programming language design and implementation*, pp. 134–143, ACM, 2009.
- [66] F. Mattern, “Virtual time and global states of distributed systems,” in *Parallel and Distributed Algorithms*, pp. 215–226, North-Holland, 1989.
- [67] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai, “Towards Integration of Data Race Detection in DSM Systems,” *Journal of Parallel and Distributed Computing*, vol. 59, pp. 180–203, 1999.
- [68] E. Pozniansky and A. Schuster, “Efficient on-the-fly data race detection in multithreaded C++ programs,” in *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 179–190, ACM, 2003.
- [69] E. Pozniansky and A. Schuster, “MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs: Research Articles,” *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 3, pp. 327–340, 2007.
- [70] D. Perkovic and P. J. Keleher, “Online data-race detection via coherency guarantees,” in *Proceedings of the second USENIX symposium on Operating systems design and implementation, OSDI '96*, (New York, NY, USA), pp. 47–57, ACM, 1996.
- [71] Y. Yu, T. Rodeheffer, and W. Chen, “RaceTrack: efficient detection of data race conditions via adaptive tracking,” in *Proceedings of the twentieth ACM*

- symposium on Operating systems principles*, SOSP '05, (New York, NY, USA), pp. 221–234, ACM, 2005.
- [72] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy, “Detecting and surviving data races using complementary schedules,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 369–384, ACM, 2011.
- [73] O. Laadan, N. Viennot, C.-C. Tsai, C. Blinn, J. Yang, and J. Nieh, “Pervasive detection of process races in deployed systems,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 353–367, ACM, 2011.
- [74] W. F. Appelbe and C. E. McDowell, “Integrating tools for debugging and developing multitasking programs,” in *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, PADD '88, (New York, NY, USA), pp. 78–88, ACM, 1988.
- [75] D. P. Helmbold and C. E. McDowell, “Computing reachable states of parallel programs,” in *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, PADD '91, (New York, NY, USA), pp. 76–84, ACM, 1991.
- [76] E. Morgan and R. Razouk, “Interactive state-space analysis of concurrent systems,” *IEEE Transactions on Software Engineering*, vol. 13, pp. 1080–1091, 1987.
- [77] R. N. Taylor, “A general-purpose algorithm for analyzing concurrent programs,” *Commun. ACM*, vol. 26, pp. 361–376, May 1983.

- [78] R. N. Taylor, *Static analysis of the synchronization structure of concurrent programs*. PhD thesis, Boulder, CO, USA, 1980. AAI8114013.
- [79] R. N. Taylor, “Analysis of concurrent software by cooperative application of static and dynamic techniques,” in *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, (New York, NY, USA), pp. 127–137, Elsevier North-Holland, Inc., 1984.
- [80] O. Herzog, “Static Analysis of Concurrent Processes for Dynamic Properties Using Petri Nets,” in *Proceedings of the International Symposium on Semantics of Concurrent Computation*, (London, UK, UK), pp. 66–90, Springer-Verlag, 1979.
- [81] K. R. Apt, “A Static Analysis of CSP Programs,” in *Proceedings of the Carnegie Mellon Workshop on Logic of Programs*, (London, UK, UK), pp. 1–17, Springer-Verlag, 1984.
- [82] G. Bristow, C. Drey, B. Edwards, and W. Riddle, “Anomaly detection in concurrent programs,” in *Proceedings of the 4th international conference on Software engineering, ICSE '79*, (Piscataway, NJ, USA), pp. 265–273, IEEE Press, 1979.
- [83] V. Balasundaram and K. Kennedy, “Compile-time detection of race conditions in a parallel program,” in *Proceedings of the 3rd international conference on Supercomputing, ICS '89*, (New York, NY, USA), pp. 175–185, ACM, 1989.
- [84] D. Callahan and J. Sublok, “Static analysis of low-level synchronization,” in *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging, PADD '88*, (New York, NY, USA), pp. 100–111, ACM, 1988.

- [85] K. M. Olender and L. J. Osterweil, “Cecil: A sequencing constraint language for automatic static analysis generation,” *IEEE Trans. Softw. Eng.*, vol. 16, pp. 268–280, Mar. 1990.
- [86] L. Osterweil, “Integrating the testing, analysis and debugging of programs,” in *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, (New York, NY, USA), pp. 73–102, Elsevier North-Holland, Inc., 1984.
- [87] J. H. Reif, “Data flow analysis of communicating processes,” in *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL ’79, (New York, NY, USA), pp. 257–268, ACM, 1979.
- [88] R. N. Taylor and L. J. Osterweil, “Anomaly detection in concurrent software by static data flow analysis,” *IEEE Trans. Softw. Eng.*, vol. 6, pp. 265–278, May 1980.
- [89] C. Flanagan and S. N. Freund, “Type-based race detection for Java,” in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI ’00, (New York, NY, USA), pp. 219–232, ACM, 2000.
- [90] A. Loginov, V. Sarkar, J. deok Choi, J. deok Choi, A. Logthor, and I. V. Sarkar, “Static datarace analysis for multithreaded object-oriented programs,” tech. rep., IBM Research Division, Thomas J. Watson Research Centre, 2001.
- [91] C. A. R. Hoare, “Monitors: an operating system structuring concept,” *Commun. ACM*, vol. 17, pp. 549–557, Oct. 1974.

- [92] M. Young and R. M. Taylor, “Combining static concurrency analysis with symbolic execution,” *IEEE Trans. Softw. Eng.*, vol. 14, pp. 1499–1511, Oct. 1988.
- [93] R. N. Taylor, “Complexity of analyzing the synchronization structure of concurrent programs,” *Acta Informatica*, vol. 19, pp. 57–84, 1983. 10.1007/BF00263928.
- [94] P. A. Emrath and D. A. Padua, “Automatic detection of nondeterminacy in parallel programs,” in *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging, PADD '88*, (New York, NY, USA), pp. 89–99, ACM, 1988.
- [95] P. A. Emrath, S. Ghosh, and D. A. Padua, “Detecting nondeterminacy in parallel programs,” *IEEE Softw.*, vol. 9, pp. 69–77, Jan. 1992.
- [96] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi, “Optimizing memory transactions,” in *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, PLDI '06*, (New York, NY, USA), pp. 14–25, ACM, 2006.
- [97] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma, “Ad hoc synchronization considered harmful,” in *Proceedings of the 9th USENIX conference on Operating systems design and implementation, OSDI'10*, (Berkeley, CA, USA), pp. 1–8, USENIX Association, 2010.
- [98] S. L. Min and J.-D. Choi, “An efficient cache-based access anomaly detection scheme,” in *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems, ASPLOS-IV*, (New York, NY, USA), pp. 235–244, ACM, 1991.



- [99] M. Prvulovic and J. Torrellas, “ReEnact: using thread-level speculation mechanisms to debug data races in multithreaded codes,” in *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, (New York, NY, USA), pp. 110–121, ACM, 2003.
- [100] P. Zhou, R. Teodorescu, and Y. Zhou, “Hard: Hardware-assisted lockset-based race detection,” in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, (Washington, DC, USA), pp. 121–132, IEEE Computer Society, 2007.
- [101] A. Nistor, D. Marinov, and J. Torrellas, “Light64: lightweight hardware support for data race detection during systematic testing of parallel programs,” in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 541–552, ACM, 2009.
- [102] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas, “SigRace: signature-based data race detection,” in *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, (New York, NY, USA), pp. 337–348, ACM, 2009.
- [103] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. Austin, “Demand-driven software race detection using hardware performance counters,” in *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, (New York, NY, USA), pp. 165–176, ACM, 2011.
- [104] J. Burnim and K. Sen, “Asserting and checking determinism for multithreaded programs,” in *ESEC/FSE '09: Proceedings of the 7th conference on the foundations of software engineering*, pp. 3–12, ACM, 2009.

- [105] C. Sadowski, S. N. Freund, and C. Flanagan, “SingleTrack: A dynamic determinism checker for multithreaded programs,” in *Programming Languages and Systems*, vol. 5502 of *Lecture Notes in Computer Science*, pp. 394–409, Springer Berlin / Heidelberg, 2009.
- [106] T. Liu, C. Curtsinger, and E. D. Berger, “Dthreads: efficient deterministic multithreading,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP ’11, (New York, NY, USA), pp. 327–336, ACM, 2011.
- [107] H. Cui, J. Wu, C.-C. Tsai, and J. Yang, “Stable deterministic multithreading through schedule memoization,” in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI’10, (Berkeley, CA, USA), pp. 1–13, USENIX Association, 2010.
- [108] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang, “Efficient deterministic multithreading through schedule relaxation,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP ’11, (New York, NY, USA), pp. 337–351, ACM, 2011.
- [109] S. Tasirlar and V. Sarkar, “Data-driven tasks and their implementation,” in *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP ’11, (Washington, DC, USA), pp. 652–661, IEEE Computer Society, 2011.
- [110] S. Imam and V. Sarkar, “Integrating task parallelism with actors,” in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) (To Appear)*, 2012.

- [111] E. M. Westbrook, J. Zhao, Z. Budimlic, and V. Sarkar, “Practical permissions for race-free parallelism,” in *ECOOP*, pp. 614–639, 2012.
- [112] E. M. Westbrook, J. Zhao, Z. Budimlic, and V. Sarkar, “Permission regions for race-free parallelism,” in *RV*, pp. 94–109, 2011.

## Appendix A

### Anomaly in Crypt with Optimizations

This section explores the performance anomaly seen with Crypt benchmark when using static optimizations. Section 8.3 discusses the performance of the ESP-bags algorithm along with the static optimizations. In general, the slowdown of ESP-bags with optimizations is lower than the slowdown of ESP-bags without optimizations. But for Crypt benchmark, the slowdown of ESP-bags with optimizations is  $12.93\times$  (relative to uninstrumented 1-thread execution time of Crypt), which is about  $1.5\times$  slower than the performance of ESP-bags without optimizations, as shown in Table 8.2.

All performance results in Chapter 8 use the smallest time measured in 3 executions repeated in the same JVM instance. For Crypt, when we repeat the experiment by performing 30 executions in the same JVM instance, the optimized version performs better than the unoptimized version.

Table A.1 shows the execution time of Crypt on 1-thread by performing 30 executions in the same JVM instance for three cases: uninstrumented, ESP-bags without optimizations, and ESP-bags with optimizations. Though the optimized version is slower than the unoptimized version for the first few iterations, beyond 15 iterations, the optimized version is consistently better than or same as the unoptimized version. This occurs because, in the case of Crypt, the memory footprint of the uninstrumented program is very high. Consequently, the memory footprint of the instrumented version is even higher. So, it takes a longer time to warm up the hardware resources used

Table A.1 : Crypt execution repeated for 30 iterations

Iteration #	Execution Time on 1-thread (s)		
	Uninstrumented	ESP-bags w/o Opts	ESP-bags w/ Opts
1	10.91	165.82	209.50
2	10.41	122.93	146.69
3	10.82	89.98	138.41
4	10.69	63.88	153.42
5	10.68	58.88	79.61
6	10.67	50.24	99.71
7	10.67	63.35	64.76
8	10.82	41.85	47.64
9	10.57	38.38	15.00
10	10.65	58.22	60.07
11	10.64	68.84	88.69
12	10.64	24.69	59.43
13	10.55	25.18	45.26
14	10.61	26.07	15.45
15	10.51	25.39	77.57
16	10.58	37.57	18.18
17	10.62	42.12	23.44
18	10.61	34.41	25.22
19	10.61	34.80	24.99
20	10.56	34.89	24.73
21	10.58	34.39	25.13
22	10.60	34.81	24.88
23	10.79	24.51	24.70
24	10.46	34.52	24.93
25	10.46	24.69	25.10
26	10.66	24.79	25.34
27	10.42	25.55	25.63
28	10.45	24.77	25.76
29	10.47	25.35	15.18
30	10.46	24.87	15.63

by the instrumented version. The minimum execution time of the optimized version is 15 seconds while that of the unoptimized version is 24.51 seconds.

Table A.2 shows the result of repeating the above experiment with some additional JVM parameters. The additional parameters used are: `-XX:MaxPermSize=256m -XX:+UseParallelGC -XX:+UseParallelOldGC -XX:NewRatio=2 -XX:-UseGCOverheadLimit`. With these additional JVM parameters, the execution times of both the optimized and the unoptimized versions are lower than earlier in the initial iterations. Now, the execution times seem to reach a steady state with fewer iterations. Also, note that, with these additional parameters, we see lower execution times more often than before. This improvement over the execution times in Table A.1 is because there were frequent executions of the garbage collector in the earlier experiment. But, in this experiment, the number of executions of the garbage collector is reduced with these JVM options. In particular, the `-XX:-UseGCOverheadLimit` option specifies a policy that limits the proportion of the time that the JVM spends in garbage collection before an `OutOfMemory` error is thrown. Now, the minimum execution time of the optimized version is 11.54 seconds whereas that of the unoptimized version is 21.03 seconds.

Table A.2 : Crypt execution repeated for 30 iterations with additional JVM parameters: `-XX:MaxPermSize=256m -XX:+UseParallelGC -XX:+UseParallelOldGC -XX:NewRatio=2 -XX:-UseGCOverheadLimit`

Iteration #	Execution Time on 1-thread (s)		
	Uninstrumented	ESP-bags w/o Opts	ESP-bags w/ Opts
1	11.52	66.89	130.91
2	10.15	60.75	69.46
3	9.85	42.72	57.15
4	9.90	49.78	11.55
5	9.88	21.03	33.24
6	9.99	30.14	65.33
7	10.01	36.70	31.10
8	9.45	70.46	27.34
9	9.52	24.81	30.02
10	9.57	37.21	28.54
11	9.55	36.96	30.79
12	9.48	37.01	27.42
13	9.55	24.65	26.85
14	9.80	37.12	24.69
15	9.78	24.62	24.54
16	9.73	24.61	16.02
17	9.63	25.16	15.64
18	9.91	24.94	15.50
19	9.57	24.75	16.17
20	9.79	24.69	15.55
21	9.50	24.62	15.82
22	9.51	24.67	15.87
23	9.54	24.62	15.48
24	9.37	24.62	24.62
25	9.74	24.60	15.35
26	9.50	25.05	15.53
27	9.51	24.63	15.73
28	9.84	24.57	15.81
29	9.55	25.03	15.87
30	9.87	24.54	16.40