Test-Driven Repair of Data Races in Structured Parallel Programs

Rishi Surendran
Rice University
rishi@rice.edu

Raghavan Raman
Oracle Labs
raghavan.raman@oracle.com

Swarat Chaudhuri
Rice University
swarat@rice.edu

John Mellor-Crummey
Rice University
johnmc@rice.edu

Vivek Sarkar
Rice University
vsarkar@rice.edu

Abstract

A common workflow for developing parallel software is as follows: 1) start with a sequential program, 2) identify subcomputations that should be converted to parallel tasks, 3) insert synchronization to achieve the same semantics as the sequential program, and repeat steps 2) and 3) as needed to improve performance. Though this is not the only approach to developing parallel software, it is sufficiently common to warrant special attention as parallel programming becomes ubiquitous. This paper focuses on automating step 3), which is usually the hardest step for developers who lack expertise in parallel programming.

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

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Algorithms, Languages, Verification

Keywords Data race, Program repair, Structured parallelism, Async, Finish

PLDI '14, June 9–11, 2014, Edinburgh, United Kingdom.
Copyright © 2014 ACM 978-1-4503-2784-8/14/06…$15.00.
http://dx.doi.org/10.1145/2594291.2594335

1. Introduction

Today, inexpensive multicore processors are ubiquitous and the demand for parallel programming is higher than ever. However, despite advances in parallel programming models, it is widely acknowledged that reasoning about correct synchronization in parallel programs requires a level of expertise that many mainstream programmers lack. Our belief is that domain experts have a good intuition of which subcomputations can logically execute in parallel, but stumble when adding synchronization between subcomputations. With this premise in mind, we propose a novel approach to address the parallel programming problem by focusing on the following workflow: 1) start with a sequential program, 2) identify subcomputations that should be converted to parallel tasks, and 3) insert synchronization to achieve the same semantics as the sequential program while maximizing parallelism. In our approach, the programmer is responsible for steps 1) and 2). This paper focuses on automating step 3), which is usually the hardest step for developers who lack expertise in parallel programming.

Our approach targets *structured parallel* languages such as Cilk [2], OpenMP [19], Chapel [4], X10 [6], Habanero Java (HJ) [3], and Habanero-C [5]. We use a subset of the HJ and X10 languages as an exemplar of structured parallel languages in this paper. This subset focuses on two core parallel constructs, `async` and `finish`, which are used for task creation and termination. Consider the Mergesort example in Figure 1. Following steps 1) and 2), the programmer expressed their intuition that the two recursive calls in lines 4 and 5 can execute in parallel. For step 3), the algorithms introduced in this paper can determine that a finish statement is needed around lines 4 and 5 for correctness and maximal parallelism. Now, consider the Quicksort example in Figure 2. Again, the programmer expressed their intuition that the two recursive calls in lines 6 and 7 can execute in parallel. While inserting a finish statement around lines 6 and 7 would be correct, our algorithms can determine that inserting a finish around line 11 is better because it also prevents data races, yet yields more parallelism than a finish statement around lines 6 and 7 by avoiding the need for nested recursive finish constructs.

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

Figure 1. Mergesort program. A finish statement is needed around lines 4-5 for correctness and maximal parallelism.

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

Figure 2. Quicksort program. A finish statement is needed around line 11 for correctness and maximal parallelism.

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

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

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

The rest of the paper is organized as follows. In Section 2, we formulate the problem that we are solving. Sections 3-6 present our solution. Section 7 describes our experiments. Section 8 discusses related work. Finally, Section 9 summarizes our conclusions.

2. Problem Statement

A program execution contains a *data race* when there are two or more accesses to the same variable, at least one of which is a write, and the accesses are unordered by either synchronization or program order. The primary goal of test-driven repair tool is to ensure data race freedom[1] for the provided inputs. In addition to this, the repaired programs must be well formed and must provide good performance. Although the tool is applied iteratively for different test inputs, we define the problem statement for a single iteration of the tool as follows:

Problem 1. Given a program P and input, ψ, find a set of program locations in P where finish statements must be introduced such that

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

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

Definition 1. A program is said to have maximal parallelism, if it has minimum critical path length (CPL), where critical path is the longest path in the computation graph of the program. Critical path length could also be defined as the execution time of a program on a computer with unbounded number of processors.

2.1 Input Language

Our work focuses on a task-parallel programming model based on terminally-strict [11] parallelism as in Chapel, Habanero Java (HJ) [3], and X10. Specifically, we focus on the `async` and `finish` parallel constructs in HJ and X10, which are used for task creation and termination. The statement `async { S }` creates a child task S, which may execute asynchronously (before, after or in parallel with) with respect to the remainder of the parent task. The statement `finish { S }` causes the parent task to execute S and wait for the completion of all asynchronous tasks transitively created inside S.

2.2 Examples

Consider the program shown in Figure 3, where the execution times for each of the tasks is given. Let us assume that `D` is dependent on `B` and `F` is dependent on both `A` and `D`. Some of the possible finish placements to satisfy these dependences are given in Figure 4, along with their critical path lengths. For example `( A B C ) ( D ) E F` corresponds to inserting a finish statement which encloses `async A`, `async B`, `async C` and another finish statement which encloses `async D`. The choice of finish placements can have a big impact on the critical path length and available parallelism. The number of possible finish placements can grow exponentially with program size, and finding the best possible finish placement is a complex problem. The problem becomes harder in the presence of function calls and nested task parallelism, where asyncs are nested inside asyncs.

To demonstrate how a finish statement can violate the scope of the input program, let us consider the program given in Figure 5.

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

```
async A( ); //Execution Time = 500
async B( ); //Execution Time = 10
async C( ); //Execution Time = 10
async D( ); //Execution Time = 400
async E( ); //Execution Time = 600
async F( ); //Execution Time = 500
```

Figure 3. Async-Finish program with Execution Times. The dependences in the program are B $\rightarrow$ D, A $\rightarrow$ F and D $\rightarrow$ F

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

Figure 4. Few possible finish placements for the program in Figure 3 and their critical path lengths. Parentheses represent finish statements

```
if (...) {
  async {   ...    }  // A1
  async { x = ...  }  // A2
}
async { y   = ... }   // A3
async { ...  =  x+y } // A4
```

Figure 5. Async-finish code which demonstrates the scoping issues in finish insertion

There are two data races in this program: A2 $\rightarrow$ A4 and A3 $\rightarrow$ A4. There are two ways to fix these data races using finish statements:

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

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

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

3. Overview

In this section, we present an overview of our approach to test-driven repair of parallel programs. The overall approach is incremental by design. A single iteration of the tool takes as input an unsynchronized or under-synchronized parallel program with async and finish constructs and a test input. It executes the program in the canonical sequential (depth-first) order with the given input, and identifies all potential data races by employing a modified version of the ESP-bags algorithm [20] that builds an extended Dynamic Program Structure Tree (DPST) [21] for that execution. The output of the iteration identifies static points in the program where finish statements should be inserted to cover all data races for that particular execution.

A high level view of the tool is given in Figure 6. The three main steps in test-driven repair of data races are:

- **Data Race Detection:** Our tool executes the program sequentially with the provided input to identify data races in the program. To identify data races, the tool uses a modified version of the ESP-bags algorithm, which is explained in Section 4.

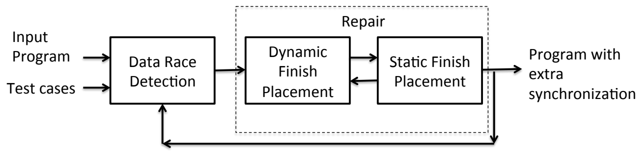

Figure 6. High level view of test-driven repair

```
async { ... = x}  // A1
async { ... = x } // A2
async { x = ... } // A3
```

Figure 7. Async-finish code with multiple data races

While the program executes, the data race detector constructs a data structure called Scoped Dynamic Program Structure Tree (S-DPST). S-DPST is an ordered rooted tree that captures the relationships among the async, finish, scope and step instances in the program, where a step instance is a maximal sequence of statement instances in a particular scope with no asyncs and finishes. Section 4.2 describes the S-DPST in detail.

- **Dynamic Finish Placement:** Our tool analyzes the S-DPST, which is annotated with the set of data races, to find the points in the S-DPST where additional finish statements are required. Section 5 presents our dynamic finish placement algorithm.
- **Static Finish Placement:** Our tool maps points in the S-DPST where additional finish statements are required to points in the program (AST). Section 6 presents our static finish placement algorithm.

We iteratively perform Dynamic and Static Finish Placements until all data races discovered in the program with the test input are repaired.

4. Data Race Detection

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

4.1 Multiple Reader-Writer ESP-Bags

Detecting data races is an important step in identifying the synchronization necessary to maintain correctness of parallel programs. In this work, we use the ESP-bags algorithm [20] to identify data races in parallel programs with `async` and `finish` constructs. The ESP-bags algorithm detects data races in a given program if and only if a data race exists. This algorithm performs a sequential depth first execution of the parallel program with the given input. By monitoring the memory accesses in the sequential execution, the algorithm identifies data races that may occur in some parallel execution of the program for that input. If the algorithm reports no data races for an execution of a program with an input, then no execution of the program for that input will encounter a data race. The sequential depth first execution of a parallel program is similar to the execution of an equivalent sequential program obtained by eliding the keywords `async` and `finish`. The ESP-bags algorithm maintains an access summary for each memory location monitored for data races; each location's access summary requires $O(1)$ space.

The ESP-bags algorithm reports only a subset of all the data races present in the program for a given input. This limitation is due to the constraint that ESP-bags keeps track of only one writer and one reader corresponding to each memory location. Consider the async-finish code given in Figure 7. There are two Read $\rightarrow$ Write data races in this code snippet due to parallel accesses to the global variable `x`. The first data race is from the async `A1` to the async `A3` and the second data race is from the async `A2` to the async `A3`. The ESP-bags algorithm reports only the data race `A1` $\rightarrow$ `A3`, because it keeps track of only one of the readers of a memory location. This data race could be fixed by enclosing `A1` inside a finish, but this will not fix the data race from `A2` to `A3`.

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

4.2 Scoped Dynamic Program Structure Tree

In this section, we present the principal data structure used for our analysis: Scoped Dynamic Program Structure Tree (S-DPST). S-DPST is an extension of Dynamic Program Structure Tree (DPST) [21], which is used in parallel data race detection for structured parallel programs.

Definition 2. The Scoped Dynamic Program Structure Tree (S-DPST) for a given execution is a tree in which all leaves are step instances, and all interior nodes are async, finish and scope instances. The parent relation is defined as follows:

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

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

A Scope node represents a scope encountered during the execution of the program. For instance, a scope node may represent an `if` statement, a `while` loop or a function call. The scope nodes ensures that the start and end points of a newly introduced finish statement are in the same scope of the input program (see Section 5). A *non scope node* refers to an async, finish or step node.

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

Example: Consider the incorrectly synchronized Fibonacci program in Figure 8. The program is incorrect because the async statement in line 12 can execute in parallel with line 14, both of them access the field `X.v` and one of them is write. Similarly there is a data race due to the access to field `Y.v`. Fig. 9 shows a subtree of S-DPST for the Fibonacci program. Each node is labeled with the type of the node and a number which indicates the order in which the node is visited in a depth first traversal of the tree. `Async0` corresponds to instances of the async statement in line 18, `Async1` corresponds to instances of the async in line 12 and `Async2` corresponds to instances of the async in line 13. The scope nodes in the S-DPST are labeled `Fib` and `If`, which corresponds to the scope of the `fib` function and the `if` statement in line 6 respectively. The data races due to the parallel accesses to `X.v` and `Y.v` are shown using the dotted directed edges.

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

Figure 8. Incorrectly synchronized Fibonacci program

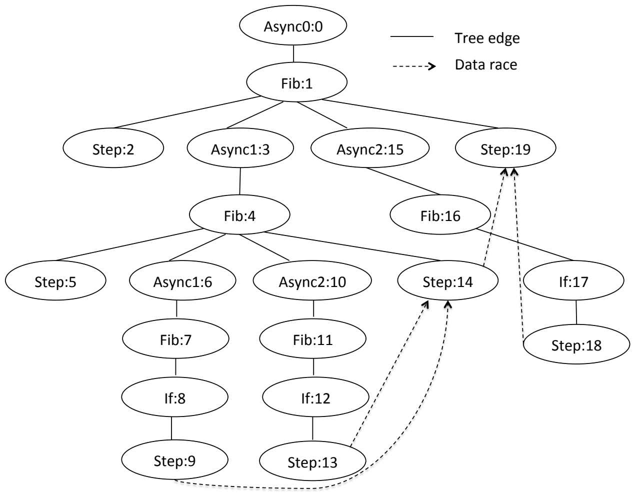

Figure 9. Subtree of S-DPST for Fibonacci

Definition 3. A non-scope child of a node p in S-DPST is a node c, which is a direct descendent of p with only scope nodes along the path from p to c.

Definition 4. The non-scope least common ancestor (NS-LCA) of two nodes n_i and n_j in S-DPST is a node l_{ij} such that if l'_{ij} is the Least Common Ancestor (LCA) of n_i and n_j in the S-DPST, then l_{ij} is the first non-scope node along the path from l'_{ij} to the root of the S-DPST.

Definition 5. The non-scope least common ancestor (NS-LCA) of a data race, D in S-DPST is the NS-LCA of n_i and n_j, where n_i is the source and n_j is the sink of the data race, D.

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

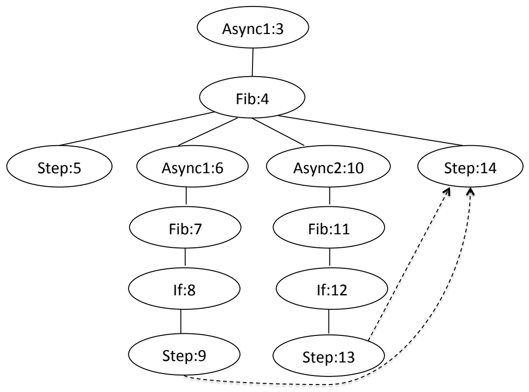

Figure 10. A subtree rooted at NS-LCA for Fibonacci

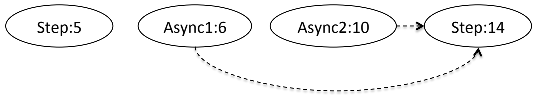

Figure 11. Dependence graph constructed from the subtree in Figure 10

5. Dynamic Finish Placement

In this section, we present the algorithm for dynamic finish placement, which involves two main steps: dependence graph construction which is presented in Section 5.1 and the application of a dynamic programming algorithm on the dependence graph, which is presented in Section 5.2.

5.1 Dependence Graph Construction

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

Example: The S-DPST in Figure 9 has two NS-LCAs: *Async0:0* and *Async1:3*. We demonstrate the dependence graph construction on the subtree rooted at *Async1:3*, which is given in Figure 10. Figure 11 shows the dependence graph constructed using the method presented above. The nodes in the graph are the non-scope children of *Async1:3* and the edges represent the data races between their descendent steps.

5.2 Algorithm

In this section, we present the core algorithm of our test-driven repair tool. The algorithm takes as input the dependence graph constructed from the subtree rooted at a NS-LCA by the method presented in Section 5.1 and finds the set of finishes needed to fix the data races represented in the dependence graph. The problem of optimal finish placement could be stated formally as follows: Let $G = (V, E)$ be a directed acyclic graph where $V = \{1..n\}$ is the set of vertices and $E = \{(x_1, y_1)..(x_m, y_m)\}$ is the set of edges. The set of edges, E satisfy the property $\forall(x_i, y_i) \in E, x_i < y_i$. The execution time of each vertex i is represented as t_i. We are interested in finding a set of points in the graph, where finish nodes need to be introduced such that it resolves all the data races and minimizes the execution time of G. The set of program points where finish nodes need to be introduced is represented using a set of ordered pairs, $FinishSet = \{(s_1, e_1)..(s_n, e_n)\}$, where each $(s_i, e_i) \in FinishSet$ represents a finish block which encloses the set of vertices $s_i..e_i$. To summarize, we need to compute $FinishSet$, such that

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

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

This problem exhibits *optimal substructure*. That is, the solution to the problem could be expressed in terms of solutions to smaller subproblems as shown in Figure 12. $OPT\ (i, j)$ represents the optimal cost for the subproblem involving the nodes $i..j$ and the corresponding edges. $succ(i)$ represents the set of nodes to which there is an edge from i, and $succ(i..k) = succ(i) \cup succ(i+1)..\cup succ(k)$. $OPT(i, j)$ is computed from two sub problems $i..k$ and

Algorithm 1 Dynamic finish placement algorithm

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

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

Figure 12. Optimal substructure of finish placement

$$EST(i+1,i..i) = \begin{cases} 0 & \text{if } i \text{ is an async} \\ t_i & \text{otherwise} \end{cases} \quad (2)$$

$$EST(i,j) = \begin{cases} OPT(i,k) + EST(k+1,i..k) & succ(i..k) \cap \{k+1..j\} \neq \emptyset \\ EST(k+1,i..k) + EST(j+1,k+1..j) & \text{otherwise} \end{cases} \; k \text{ is the optimal partitioning point for } i..j \quad (3)$$

Figure 13. Earliest start time computation

Algorithm 2 Checks the validity of a finish placement

```
 1: procedure VALIDHELP($node_1, node_2, left, right$)
 2:     $lca_{1l} = \text{LCA}(node_1, left)$
 3:     $lca_{12} = \text{LCA}(node_1, node_2)$
 4:     $lca_{2r} = \text{LCA}(node_2, right)$
 5:     $d_{1l} = lca_{1l}.depth$
 6:     $d_{12} = lca_{12}.depth$
 7:     $d_{2r} = lca_{2r}.depth$
 8:     if $(d_{1l} > d_{12}) \vee (d_{2r} > d_{12})$ then
 9:         return $false$
10:     end if
11:     return $true$
12: end procedure
13: procedure VALID($i, j$)
14:     return VALIDHELP($node[i]$, $node[j]$, $node[i-1]$, $node[j+1]$)
15: end procedure
```

Algorithm 3 Find the set of finishes from the output of Algorithm 1

Input: Partition, Finish
Output: Set of finishes

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

$k+1..j$. The value of k is chosen such that it gives us the minimum value of $OPT(i,j)$. We refer to k as the *optimal partitioning point* for the problem $i..j$. The possible *partitioning points* are $i, i+1, ..j-1$. The first case represents where there are edges from the first partition $(i..k)$ to the second partition $(k+1..j)$. In this case, a finish is required around the first partition to satisfy the dependence from the first partition to the second partition. The second case represents the case where there are no edges from the first partition to the second partition. In this case a finish is not required. Figure 13 shows the computation of EST.

Algorithm 1 shows the dynamic programming method used to compute the optimal finish placement. $Opt[i][j]$ holds optimal cost for the subproblem $i..j$. To help us keep track of how to construct an optimal solution, we save the optimal partitioning point of $i..j$ in $Partition[i][j]$. $Finish[i][j]$ keeps track of whether a finish is required around the block $i..k$.

The loop in lines 14-32 iterates through each of the possible partitioning points and finds the optimal one. Lines 16-20 handle the case where a finish is not required and lines 21-25 handle the case where a finish is required. Note that it considers only values of k for which (i,k) has a valid static finish placement. Procedure VALID(i,j) in Algorithm 2 checks the validity of a finish placement. (i,j) is a valid finish placement, only if there exists a point in the DPST where we can introduce a finish node, whose descendents include the nodes $i..j$, but do not include nodes $i-1$ or $j+1$. The array $node$ used in VALID contains all the nodes in the dependence graph, ordered from left to right.

Algorithm 1 computes the optimal solution in $O(n^3 \times d)$ time, by taking advantage of the overlapping-subproblems property, where d represents the height of the subtree rooted at LCA. There are only $\Theta(n^2)$ different subproblems in total. The solution for each of these subproblems is computed once in $O(n \times d)$ time.

Algorithm 1 computes the optimal cost for a finish placement, which satisfies all the dependences in the dependence graph. It does not directly show the partitioning points. This information can be easily determined from the arrays $Partition$ and $Finish$. $1..(Partition[1][n])$ and $(Partition[1][n]+1)..n$ are the two subproblems used to compute the optimal solution for $1..n$. The value of $Finish[1][n]$ determines whether a finish is required around $1..Partition[1][n]$. The finish placements for all the subproblems can be computed recursively as presented in Algorithm 3.

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

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

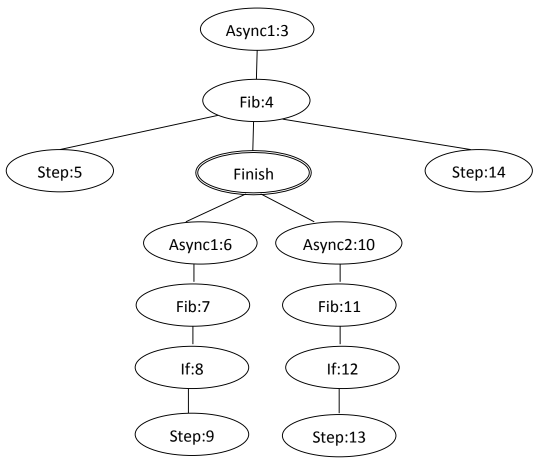

Figure 14. Subtree in Figure 10 after inserting finish

5.3 Correctness and Optimality

The dependence graph constructed using the method in Section 5.1 models the data races as dependences between the children of the NS-LCA. The dynamic programming algorithm given in Section 5.2 ensures that new finish nodes are inserted such that the source and sink of a data race may not happen in parallel.

Theorem 1. *Consider two leaf nodes S1 and S2 in a S-DPST, where S1 $\neq$ S2 and S1 is to the left of S2. Let N be the node denoting the non-scope least common ancestor of S1 and S2. Let node A be the ancestor of S1 that is the non-scope child of N. Then, S1 and S2 can execute in parallel if and only if A is an async node.*

Theorem 1 is an extension of a result from [21] which gives the necessary and sufficient condition for 2 steps to execute in parallel. To resolve a data race between 2 steps S1 and S2, we need to introduce a finish node, F in the S-DPST, such that

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

Theorem 2. *Consider a node L in S-DPST. Let $G = (V, E)$ be a directed acyclic graph (DAG) in which nodes $V = \{1, .., n\}$ represent the dynamic children of L and E represent the set of data races whose dynamic LCA is L. Algorithm 1 finds an optimal set of finish placement which resolves all the data races represented by E.*

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

6. Static Finish Placement

In this section, we present the algorithm for finding a static finish placement from the dynamic finish placements computed using the algorithms presented in Section 5.

6.1 Algorithm

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

```
static class BoxInteger {
      public int v;
   }

   void fib (BoxInteger ret, int n) {
       if ( n < 2) {
           ret.v = n;
           return;
       }
       final BoxInteger X = new BoxInteger();
       final BoxInteger Y = new BoxInteger();
       finish {  // Newly inserted finish
           async fib (X, n-1);     // Async1
           async fib (Y, n-2);     // Async2
       }
       ret.v = X.v + Y.v;
  }
       ....
       async fib(3);        // Async0
```

Figure 15. Fibonacci program from Figure 8 after finish insertion

1. Find the NS-LCA of the source and sink of each of the data races in the S-DPST
2. Group all the data races which have a common NS-LCA
3. For each unique NS-LCA , N
   (a) Reduce the subtree rooted at N to a DAG as described in Section 5.1.
   (b) Find the set of finish nodes needed to fix the data races with NS-LCA, N using the dynamic programming algorithm presented in Section 5.2.
   (c) Find the locations in the S-DPST where a finish needs to be inserted by a bottom-up traversal.
   (d) Insert the finish statements in the input program and update the S-DPST with the new finish nodes
   (e) Remove the data races which are fixed by the insertion of the finish nodes.
   (f) Update the data races for which the NS-LCA have changed due to the insertion of new finish nodes.

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

Example: The subtree rooted at *Async1:3* after dynamic finish placement is shown in Figure 14. The finish placement in this subtree are then propagated to the rest of the S-DPST. This will introduce another Finish node as the child of *Fib:1* and as the parent of *Async1:3* and *Async2:15* in Figure 9. At this point all the data races have been fixed and the program with the newly introduced finish statement is given in Figure 15.

6.2 Correctness & Conditions for Optimality

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

Theorem 3. *Consider a program P with n data races $\{D_1,..,D_n\}$. Let $\{L_1,..,Ln\}$ be the non-scope least common ancestors (NS-LCA) of the nodes corresponding to the steps involved in the data race in the S-DPST. A finish node in the S-DPST which resolves a data race D_i may resolve a data race D_j only if $L_i = L_j$.*

From Theorem 3, it follows that the problem of global finish placement could be solved by grouping data races by their NS-LCA and solving the problem locally. If the subproblems at different NS-LCAs are non-overlapping this leads to an optimal solution. If the subproblems are overlapping, the decisions made in the solution of earlier subproblems may lead to non-optimal solutions for later subproblems. In most real world programs, we observed that the solutions required at different NS-LCAs are identical or non-overlapping, which leads to a global optimal solution.

7. Experimental Results

In this section, we present experimental results for our test-driven repair tool. The different components of the tool shown in Figure 6 were implemented as follows. Programs were instrumented for race detection, S-DPST construction and computation of execution time of steps during a byte-level transformation pass on HJ's Parallel Intermediate Representation (PIR) [18]. The data race detector (modified version of ESP-Bags) was implemented as a Java library for detecting data races in HJ programs containing async and finish constructs, and generating trace files containing all the data races detected. The dynamic and static finish placement algorithms were implemented as subsequent compiler passes in the HJ compiler, that read the trace files generated by the data race detector, update the PIR representation of the program, and then output the source code positions where additional finish constructs should be inserted.

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

We evaluated the repair tool on a suite of 12 task-parallel benchmarks listed in Table 1. The fourth column of Table 1 shows the input sizes used for repair mode (which includes data race detection and S-DPST construction). The fifth column shows the input size used for performance evaluation of the repaired programs.

7.1 Repairing Programs

To evaluate our tool, we performed the following test. We removed all finish statements from the benchmarks in Table 1, and then ran the repair tool on each of the resulting buggy programs. For these programs, a single iteration of the tool with a single test case (input size shown in column 4 of Table 1) was sufficient to obtain a repair that satisfied all task dependences. Further, the tool's repair (insertion of finish statements) resulted in parallel performance that was almost identical to that of the original benchmark in each case. Figure 16 shows the execution times for the sequential, original parallel, and repaired parallel versions of each benchmark when executed on 12 cores. A visual inspection confirmed that the tool was able to insert finish statements so as to obtain comparable parallelism to that created by the experts who wrote the original benchmarks.

7.2 Time for Program Repair

Table 2 shows the time to repair each of the programs using input sizes given in column 4 of Table 1. HJ-Seq is the sequential runtime of the benchmarks. The third column shows the time taken for data race detection and S-DPST construction. The fourth and fifth column gives the number of S-DPST nodes and the number of data races reported by MRW ESP-Bags algorithm respectively. Repair time is the time taken for static and dynamic finish placements. We observed that, as the number of S-DPST nodes and the number of data races increases, the time taken for the program repair also increases. This is because the time to repair is dominated by the time taken to read the trace files generated by data race detector and building the S-DPST. Although the worst-case time complexity for dynamic finish placement is $O(n^3 \times d)$, the time taken in practice is very small because n and d are small in practice.

7.3 Comparison of SRW and MRW ESP-Bags

In this section, we compare the overheads and results produced by the two data race detectors. Our tool uses the MRW ESP-Bags algorithm for data race detection by default. This guarantees that all data races are reported in a single run, for a given test case. Using the SRW ESP-Bags algorithm may require multiple iterations of the tool for the same test case to ensure that the program does not contain data races that were not identified and fixed in a prior iteration. For the benchmarks used in this paper, only two SRW iterations were needed in each case (one for repair, and one to confirm that no data races remain). The main reason to consider SRW is that each SRW iteration may generate smaller trace files than that generated by a single MRW run, and smaller trace files directly result in a smaller memory footprint and execution time for repair. This hypothesis is confirmed in Table 4, which compares the number of data races detected by a single run of the SRW and MRW algorithms.

Table 3 compares the total repair time for the MRW and SRW algorithms (including two runs in the SRW case). We see that the execution times are comparable in many cases, but there is a big difference for mergesort for which MRW's repair time is more than $4\times$ slower than SRW's repair time. This can be explained by the large absolute number of reported data races for the MRW algorithm in Table 4, for the mergesort benchmark. The time for synthesizing a correct program from MRW race reports for mergesort is higher due to the cost of reading the large traces and adding the race edges to our internal representation.

7.4 Student Homework Evaluation

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

8. Related Work

Program repair and synthesis The last decade has seen much activity in repair and synthesis of programs, including concurrent programs. Vechev, Yahav, and Yorsh [25] use abstract interpretation to analyze atomicity violations and abstraction-guided synthesis to introduce minimal critical regions into a program to eliminate atomicity violations by restricting potential interleavings. A

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

Table 1. List of Benchmarks Evaluated

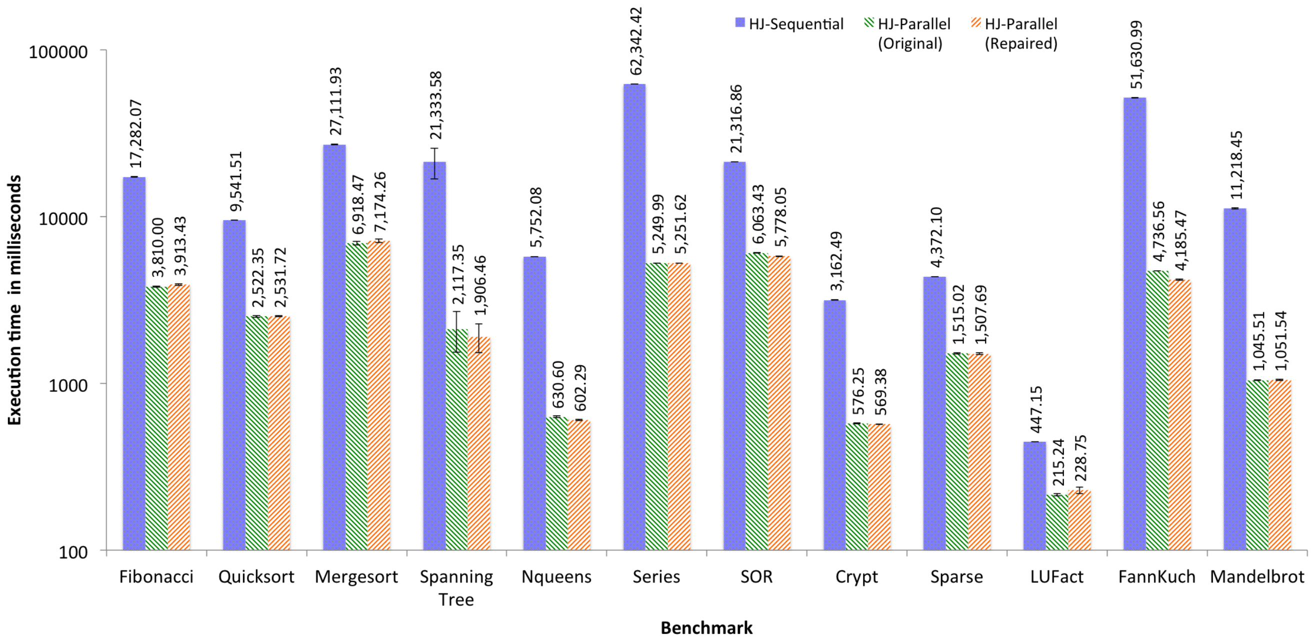

Figure 16. Average execution times in milliseconds and 95% confidence interval of 30 runs for sequential, original parallel, and repaired parallel versions, for “Performance” input size. Parallel versions were run on 12 cores.

prototype based on this approach has been demonstrated on program fragments consisting of tens of lines. Cerny et al. [24] present a method for using a performance model to guide refinement of a non-deterministic partial program into one that is both race and deadlock free. They use manually derived abstractions of programs and the SPIN model checker to reason about transitions. This system was also applied to tens of lines of code. Raychev et al. [22] used abstract interpretation to compute an over-approximation of the possible program behaviors. If the over-approximation is not free of conflicts, the algorithm synthesizes a repair that enforces conflict freedom. Solar-Lezama et al. [23] synthesizes concurrent data structures from high-level “sketches”.

The most closely related work on concurrent program repair is described in a pair of papers by Jin et al. [12, 13]. In a 2011 paper [12], they describe AFix—a system for detection and repair of concurrency bugs resulting from single-variable atomicity violations. Their system detects atomicity violations, designs a repair by adding a critical section protected by a lock to prevent problematic interleavings. A 2012 paper [13] describes CFix—a more comprehensive system for detecting and repairing several kinds of concurrency bugs, including atomicity violations, ordering violations, data races, and def-use errors. CFix relies on several different existing bug detectors to detect different types of concurrency bugs. They fix bugs by adding ordering and/or mutual exclusion. With respect to [12] and [13], which avoids atomicity violations by ordering accesses: we add finish constructs which (a) eliminate the observed races, and (b) ensure that the semantics of the accesses in the parallel program is the same as in the sequential version. [12] and [13] merely ensure that the accesses aren’t concurrent by adding mutual exclusion or pairwise ordering.

Kelk et al. uses a combination of genetic algorithms and program optimization to automatically repair concurrent Java pro-

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

Table 2. Time for Program Repair. Input size: Repair

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

Table 3. Comparison of SRW ESP-Bags and MRW ESP-Bags. Input size: Repair

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

Table 4. Number of data races detected by SRW ESP-Bags and MRW ESP-Bags. Input size: Repair

grams [14]. The usefulness of genetic algorithms in program repair was previously demonstrated by Le Goues et al. [9]. Other relevant program repair Griesmayer et al. [10], which uses a method based on model checking to repair boolean programs, and Logozzo and Ball [15], which describes a system for reasoning about .NET software that uses abstract interpretation to suggest program repairs. However, the focus in these approaches is not on concurrency bugs.

Race detection There are three general approaches for detecting concurrency bugs, including data races: *static analysis*, *dynamic analysis*, and a combination of the two. There is a large literature on each of these topic. For examples of static approaches to the problem, see Naik et al. [17] and Voung et al. [26]. In this paper, we utilize dynamic race detection techniques that identify races in an execution for a particular input. Work on dynamic detection of data races focuses either on structured parallelism or unstructured parallelism. For unstructured parallelism, vector clock algorithms are the standard, e.g., [1, 8]. For programs with structured fork-join parallelism (as in our setting), prior work has shown that for a single program input, in a single execution, one can pinpoint an example data race, or else there can be no data races with any interleaving based on that input [16]. Races can be detected in a single execution by tagging each variable with a constant number of labels that can be used to determine whether threads are concurrent or not; specifically labels a single reader and a single writer per variable suffice in a sequential execution [16]. Subsequent work on dynamic race detection by Feng and Leiserson achieves a similar qualitative result for Cilk's fully-strict parallelism using an algorithm that they call SP-Bags, but with lower asymptotic space and time overhead [7]. Raman et. al. [20] showed how to extend the SP-Bags algorithm to support terminally-strict parallelism of async-finish parallelism.

9. Conclusions

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

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

Acknowledgments

We thank members of the Habanero group at Rice for valuable discussions related to this work and contributions to the Habanero Java infrastructure used in this research. We also thank Karthik Murthy from Rice University for his feedback and suggestions. We are grateful to the anonymous reviewers for their comments and suggestions. This work was supported in part by NSF award 1242507.

References

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

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

[3] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: the new adventures of old x10. In *PPPJ '11*, pages 51–61, New York, NY, USA, 2011. ACM.

[4] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3): 291–312, Aug. 2007.

[5] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cavé, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan. Integrating asynchronous task parallelism with MPI. In *IPDPS*, pages 712–725, 2013.

[6] K. Ebcioğlu, V. Saraswat, and V. Sarkar. X10: an experimental language for high productivity programming of scalable systems (extended abstract). In *Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, February 2005.

[7] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *SPAA '97*, pages 1–11, New York, NY, USA, 1997. ACM.

[8] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI '09*, pages 121–133, New York, NY, USA, 2009. ACM.

[9] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[10] A. Griesmayer, R. Bloem, and B. Cook. Repair of Boolean programs with an application to C. In *CAV*, pages 358–371. Springer, 2006.

[11] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *IPDPS*, pages 1–12, 2009.

[12] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *PLDI '11*, pages 389–400, New York, NY, USA, 2011. ACM.

[13] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *OSDI'12*, pages 221–236, Berkeley, CA, USA, 2012. USENIX Association.

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

[15] F. Logozzo and T. Ball. Modular and verified automatic program repair. In *OOPSLA '12*, pages 133–146, New York, NY, USA, 2012. ACM.

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

[17] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*, pages 308–319, 2006.

[18] V. K. Nandivada, J. Shirako, J. Zhao, and V. Sarkar. A transformation framework for optimizing task-parallel programs. *ACM Trans. Program. Lang. Syst.*, 35(1), Apr. 2013.

[19] OpenMP. OpenMP specifications. http://www.openmp.org/specs.

[20] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Efficient data race detection for async-finish parallelism. *Formal Methods in System Design*, 41(3):321–347, Dec. 2012.

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

[22] V. Raychev, M. T. Vechev, and E. Yahav. Automatic synthesis of deterministic concurrency. In *SAS*, pages 283–303, 2013.

[23] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching concurrent data structures. In *PLDI*, pages 136–148, 2008.

[24] P. Černý, K. Chatterjee, T. A. Henzinger, A. Radhakrishna, and R. Singh. Quantitative synthesis for concurrent programs. In *CAV'11*, pages 243–259, Berlin, Heidelberg, 2011. Springer-Verlag.

[25] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL '10*, pages 327–338, New York, NY, USA, 2010. ACM.

[26] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *ESEC/FSE*, pages 205–214. ACM, 2007.

Appendix A: Software Artifact Supplement

The ACM digital library contains an artifact as a supplement to this paper. The artifact includes our tool to automatically fix data races in Habanero Java (HJ) programs written using async and finish constructs.

Components of the artifact are:

- the HJ compiler and runtime,
- instructions for setting up HJ,
- instructions for using the instrumenter and analyzer to identify where a program needs extra finishes to eliminate data races, and
- sample applications to demonstrate the capabilities of our tool.

One uses the artifact's software by performing three steps:

1. Instrument a program to dynamically detect data races.
2. Execute the instrumented program to pinpoint references involved in data races during the execution.
3. Apply our analyzer to the input program and trace files recorded by the data race detector during execution. The analyzer will identify program locations where additional finish statements are required to eliminate races.