Form Methods Syst Des (2012) 41:321–347
DOI 10.1007/s10703-012-0143-7

Efficient data race detection for async-finish parallelism

Raghavan Raman · Jisheng Zhao · Vivek Sarkar · Martin Vechev · Eran Yahav

Published online: 13 March 2012

Abstract A major productivity hurdle for parallel programming is the presence of *data races*. Data races can lead to all kinds of harmful program behaviors, including determinism violations and corrupted memory. However, runtime overheads of current dynamic data race detectors are still prohibitively large (often incurring slowdowns of 10× or more) for use in mainstream software development.

In this paper, we present an efficient dynamic race detection algorithm that handles both the async-finish task-parallel programming model used in languages such as X10 and Habanero Java (HJ) and the spawn-sync constructs used in Cilk.

We have implemented our algorithm in a tool called TASKCHECKER and evaluated it on a suite of 12 benchmarks. To reduce overhead of the dynamic analysis, we have also implemented various static optimizations in the tool. Our experimental results indicate that our approach performs well in practice, incurring an average slowdown of 3.05× compared to a serial execution in the optimized case.

Keywords Parallel programming · Program analysis · Data races · Determinism

E. Yahav is a Deloro Fellow.

R. Raman (✉) · J. Zhao · V. Sarkar
Rice University, 6100 Main St, Houston, TX 77005, USA
e-mail: raghav@rice.edu

J. Zhao
e-mail: jisheng.zhao@rice.edu

V. Sarkar
e-mail: vsarkar@rice.edu

M. Vechev
ETH Zürich, UNG H 14, Universitätstrasse 19, Zürich 8092, Switzerland
e-mail: martin.vechev@inf.ethz.ch

E. Yahav
Technion–Israel Institute of Technology, Taub Building 734, Haifa 32000, Israel
e-mail: yahave@cs.technion.ac.il

1 Introduction

Designing and implementing correct and efficient parallel programs is a notoriously difficult task, yet, with the proliferation of multi-core processors, parallel programming will play a central role in mainstream software development. One of the main difficulties in parallel programming is that programmers are often required to reason explicitly about the interleavings of operations in their programs. The vast number of interleavings makes this task difficult even for small programs and intractable for sizable applications. Unstructured and low-level frameworks such as Java threads allow the programmer to express rich and complicated patterns of parallelism but also to make mistakes.

Structured parallelism Structured parallelism makes it easier to determine the context in which an operation is executed and to identify other operations that can execute in parallel with it. This simplifies manual and automatic reasoning about the program, enabling the programmer to produce a program that is more robust and often more efficient.

Realizing these benefits, significant efforts have been made toward structuring parallel computations, starting with constructs such as *cobegin-coend* [10] and *monitors*. Recently, additional support for fork-join task parallelism has been added in the form of libraries [17, 20] to existing programming environments and languages such as Java and .NET.

Parallel languages such as Cilk [4], X10 [8], and Habanero Java (HJ) [3] provide simple, yet powerful, high-level concurrency constructs that restrict traditional fork-join parallelism yet are sufficiently expressive for a wide range of problems. The key restriction in these languages is in the flexibility of choosing which tasks a given task can join. The async-finish computations that we consider are desirable because the computation graphs generated in the language are deadlock-free [19] (unlike unrestricted fork-join computations).

Data race and determinism detection We present an efficient dynamic analysis algorithm, ESP-bags, that checks for the presence of data races (and proves data race freedom) in async-finish style parallel computations. In this work, we focus on the constructs *async*, *finish*, and *isolated*. The *async* construct is used to create a new task that can execute in parallel, the *finish* construct is used to specify a join point for a group of tasks, and the *isolated* construct is used for mutual exclusion. These constructs form the core of the larger X10 and HJ[1] parallel languages. Using *async*, *finish*, and *isolated*, one can express a wide range of useful and interesting parallel computations (both regular and irregular) such as factorizations and graph computations.

Our analysis is a generalization of Feng and Leiserson's SP-bags algorithm [11], which was designed for checking determinism of spawn-sync Cilk programs. The original algorithm cannot be applied directly to the async-finish style of programming because this model allows for a superset of the executions allowed by the traditional spawn-sync Cilk programs. Both the SP-bags algorithm and our extension to it are precise and sound for a given input[2]: if a violation is reported, then the race really exists (i.e., there are no false positives). Conversely, if a data race exists for that input, a violation will be reported (i.e., there are no false negatives).

Data race freedom affects the correctness of parallel algorithms and in some cases, it can imply determinism [6, 18]. For instance, in the absence of data races, all parallel programs

[1]The construct for mutual exclusion is called *atomic* in X10 and *isolated* in HJ.

[2]The ESP-bags algorithm is precise and sound when the program contains *async* and *finish* constructs only. When the program contains *isolated* constructs, it is precise but not sound (i.e., there may be false negatives).

$$\begin{array}{lrl} \textit{Program} & P ::= & \text{main}\{\textbf{finish}\{es\}\} \\ \textit{Extended Statement } es & ::= & \textbf{finish}\{es\} \mid \textbf{async}\{es\} \mid \textbf{isolated}\{s\} \\ & & \textbf{if}\ (b)\ es\ \textbf{else}\ es \mid es; es \mid \textbf{while}\ (b)\ es \mid \cdots \end{array}$$

Fig. 1 The syntax of AFIPL

with *async* and *finish*, but without *isolated* constructs, are guaranteed to be *deterministic*. Therefore, if we can prove data-race freedom of programs that do not contain *isolated* constructs, then we can conclude that the program is deterministic.

Main contributions To the best of our knowledge, this is the first detailed study of the problem of data race detection for async-finish task-parallel programs as embodied in the X10 and HJ languages. The main contributions of this paper are

– A dynamic analysis algorithm for efficient data race detection of structured async-finish parallel programs. Our algorithm generalizes the classic SP-bags algorithm designed for the spawn-sync Cilk model (we also show how any spawn-sync program can be checked with our algorithm).
– An implementation of our dynamic analysis in a tool named TASKCHECKER.
– Compiler optimizations to reduce the overhead incurred by the dynamic analysis algorithm. These optimizations reduce the average overhead by 37% with respect to the unoptimized version for the benchmarks used in our evaluation.
– An evaluation of TASKCHECKER on a suite of 12 benchmarks. We show that for these benchmarks, TASKCHECKER is able to perform data race detection with an average (geometric mean) slowdown of 4.86× in the absence of compiler optimizations, and 3.05× with compiler optimizations, compared to a sequential execution.

The rest of the paper is organized as follows. Section 2 introduces the structured parallel setting that our algorithm targets. Section 3 describes the ESP-bags algorithm for detecting data races in async-finish parallel programs. Section 4 proves the correctness of the ESP-bags algorithm. Section 5 describes the extensions needed in the ESP-bags algorithm to support isolated constructs. Section 6 outlines the compiler optimizations that are performed to reduce the overhead incurred by our algorithm. Section 7 describes the evaluation of our algorithm on a suite of 12 benchmarks. Section 8 discusses related work, and Sect. 9 concludes the paper.

2 Background

We present our approach to data-race detection using an abstract language AFIPL, *Async Finish Isolated Parallel Language*. We first present our language AFIPL and informally describe its semantics. To motivate the generalization of the traditional SP-bags algorithm, we give an example where our language allows for broader sets of behaviors than those expressible with the spawn-sync constructs in the Cilk programming language.

2.1 Syntax

Figure 1 shows the relevant statements of our language. The language extends any imperative sequential language with three statements: **async**, **finish** and **isolated**. The language allows for nesting of **finish** and **async** statements, but does not allow any of the new statements

```
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
```

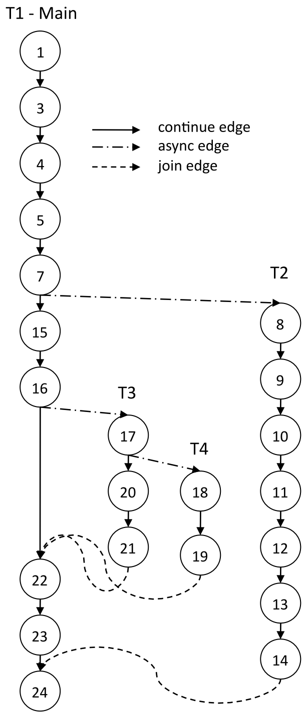

Fig. 2 An example AFIPL program and its computation graph. This code is the body of the main method in the program

to appear inside **isolated** statements: async and finish statements cannot appear inside isolated sections. However, isolated statements may contain any of the traditional statements: loops, conditionals, and so on. To reflect that, we use the name *es* to denote an extended statement and *s* to denote a traditional statement ($\cdots$ above is used to denote the remaining basic statements such as primitive assignments, heap assignments, etc). We refer to the subset of AFIPL without isolated sections as AFPL, the *Async Finish Parallel Language*. Our data race detection algorithm is largely independent of the sequential constructs in the language. For example, the sequential portion of the language can be based on the sequential portions of C, C++, Fortran, or Java.

2.2 Informal language semantics

Next, we briefly discuss the relevant semantics of the concurrency constructs. For a formal semantics of the async and finish constructs, see FX10 [19]. Initially, the program begins execution with the main task. When an **async** $\{s\}$ statement is executed by task A, a new child task, B, is created. The new task B can now proceed with executing statement s in parallel with its parent task A. For example, consider the AFIPL code of Fig. 2. The main task starts executing this code. The **async** statement in line 7 creates a new child task, which will now execute the block of code in lines 7–14 in parallel with the main task. When a **finish** $\{s\}$ statement is executed by task A, it means that task A must block and wait at the end of this statement until all descendant tasks created by A in s (including their recursively created children tasks), have terminated. That is, **finish** can be used to create a join point for all descendant tasks dynamically created inside its scope. In the example in Fig. 2, the **finish** in line 15 would wait for the tasks created by **async**s in lines 16 and 17 to complete. The statement **isolated** $\{s\}$ means that the statement s is executed atomically with respect to other isolated statements.[3] Note that in AFIPL, there is an implicit **finish** surrounding the body of the main method, which ensures that the program does not complete before all spawned tasks complete.

2.3 Cilk vs. AFIPL

Our data race detection algorithm, ESP-bags, presented in later sections, is an adaptation of the SP-bags algorithm [11] developed for the Cilk programming language. Unfortunately, the SP-bags algorithm cannot be applied directly to our language and needs to be extended because the async-finish constructs of AFIPL language supports a more relaxed concurrency model than the spawn-sync Cilk computations [13]. The static lexical scope of async-finish subsumes all of spawn-sync excluding *conditional syncs*.[4] 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, the task should join with *all* of its descendant tasks (including all recursive descendant tasks) created in between the start of the task and the join point. The join is accomplished by executing the statement **sync**. The semantics of spawn construct is exactly the same as the async construct.

The spawn-sync constructs can be translated to async-finish constructs as follows: each spawn construct can be directly replaced by 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 function or the previous sync, and the end of the finish block is the label of the sync statement. For instance, we can translate the following Cilk program,

$$\textbf{spawn}\ f1();\ \textbf{sync};\ \textbf{spawn}\ f2();\ \textbf{sync};\ s1;$$

into the following AFIPL program:

$$\textbf{finish}\{\textbf{async}\ f1();\ \};\ \ \textbf{finish}\ \{\textbf{async}\ f2();\ \};\ s1;$$

However, 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 AFIPL 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 dynamic *beginFinish* and *endFinish* scopes can be nested arbitrarily unlike 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 having started (in the same task). As a high-level analogy, the relationship between *beginFinish* / *endFinish* and AFIPL's lexical finish construct is akin to that of *MonitorEnter* / *MonitorExit*

[3] As advocated in [16], we use the *isolated* keyword instead of *atomic* to make explicit the fact that the construct supports weak isolation rather than strong atomicity.

[4] We refer to a sync that is executed under some condition in a function body as a *conditional sync*.

```
1  for (int i=0; i<size; i++ ) {
2    spawn f();
3    if (i == 3) {
4      sync;
5    }
6  } // for
7  sync;
```

(a)

```
1  beginFinish();
2  for (int i=0; i<size; i++ ) {
3    async f();
4    if (i == 3) {
5      endFinish();
6      beginFinish();
7    }
8  } // for
9  endFinish();
```

(b)

Fig. 3 (**a**) a Cilk program with conditional syncs and (**b**) its translation to AFIPL program

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).

Using *beginFinish* and *endFinish*, we can represent all of the sync constructs of Cilk (including *conditional syncs*) 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.

Figure 3 shows an example Cilk program with conditional syncs and its translation to AFIPL program. Note the conditional sync on line 4 in the Cilk program. It is translated into *endFinish(); beginFinish();* in the AFIPL program. This shows that the async-finish constructs subsume all of spawn-sync constructs. Our race detection algorithm works by intercepting the start and end of finish and async constructs. Hence, it 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 AFIPL, it is possible for a task to join with *some* rather than all of its descendant tasks. These descendant tasks are specified at the language level 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 Fig. 2 illustrates the differences between Cilk and AFIPL. Each vertical sequence of circles denotes a task. Here we have four sequences for four tasks. Each circle in the graph represents a program label, and an edge represents the execution of a statement at that label. Note that at label 22, the main task waits only for T3 and T4 but not for T2, which is not possible using the spawn-sync semantics in Cilk.

Another restriction in Cilk is that every task must execute a sync statement upon its return. That is, a task cannot terminate unless all of its descendants have terminated. In contrast, in AFIPL, 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 Fig. 2, 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. This need not be the case in AFIPL: task T3 can terminate before task T4 has finished.

More generally, the class of computations generated by the spawn-sync constructs is said to be *fully-strict* [5], while the computations generated by our language are called *terminally-strict* [1]. The set of terminally-strict computations subsumes the set of fully-strict computations. All of these relaxations mean that it is not possible to convert a AFIPL program directly into the spawn-sync semantics of Cilk, which in turn implies that we cannot use its SP-bags algorithm directly and that we need to generalize that algorithm to our setting. We show how that is accomplished in the next section.

3 ESP-bags algorithm

In this section, we first summarize the SP-bags algorithm used for spawn-sync computations. Then, we present our extension of SP-bags, called ESP-bags, for detecting data races in AFPL programs. For a given input, ESP-bags and SP-bags detect data races in a given program if and only if a data race exists (Theorem 5 in Sect. 4). That is, if the ESP-bags algorithm does not detect a data race for a given input, then it is guaranteed that there is no data race in any schedule of the program for that given input. On the other hand, if a race is found, the algorithm stops after the first one is detected. This means that there is some schedule of the program, with the given input, for which the reported race is the first one encountered. There may be other schedules with the given input that may encounter a different set of races in a different order.

The SP-bags algorithm was designed for Cilk's spawn-sync computations. As mentioned earlier, 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 are easily understood.

3.1 SP-bags

Although the program being tested for data races is a parallel program, the SP-bags algorithm is a serial algorithm that performs a sequential depth-first execution of the program on a single processor.

We assume that each dynamic task (async) instance is given a unique task ID. The basic idea behind the SP-bags algorithm is to attach two "bags", S and P, to each dynamic task instance (S stands for Serial and P for Parallel). Each bag contains a set of task IDs. When a statement E that belongs to a task A is being executed, 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. The P-bag of A holds all descendant tasks of A that may execute in parallel with E.

At any point during the depth-first execution of the program, a task ID will always belong to at most one bag. Therefore, all bags can be efficiently represented using a single disjoint-set data structure.

The intuition behind the algorithm can be stated as follows: when a program is executed in a depth-first manner, a write W_1 to a shared memory location L by a task τ_1 races with an earlier read/write to L by any task τ_2 that is in a P-bag when W_1 occurs, and it does not race with a read/write 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 following table shows the update rules for the SP-bags algorithm:

Async A	: $S_A \leftarrow \{A\}$, $P_A \leftarrow \emptyset$
Task A returns to Task B	: $P_B \leftarrow P_B \cup S_A \cup P_A$, $S_A \leftarrow \emptyset$, $P_A \leftarrow \emptyset$
EndFinish F in a Task B	: $S_B \leftarrow S_B \cup P_B$, $P_B \leftarrow \emptyset$

When a task A is created, its S bag 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 an 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

```
Read location L by Task t:
    If L.writer is in a P–bag then Data Race;
    If L.reader is in a S–bag then L.reader = t;
```

```
Write location L by Task t:
    If L.writer is in a P–bag or L.reader is in a P–bag
        then Data Race;
    L.writer = t;
```

Fig. 4 Instrumentation on shared memory access. Applies both to SP-bags and ESP-bags

A and hence, while executing this part of 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 thus, while executing this part of the code in A, all descendants of A before the join should be in an S bag.

In addition to the above steps, during the depth-first execution of a program, the SP-bags algorithm maintains two additional fields for each memory location: a *reader* task ID and a *writer* task ID, and takes an action on every read and write of a shared variable. Figure 4 shows the required instrumentation for *read* and *write* operations. For each operation on a shared memory location L, we only need to check those fields of L that could conflict with the current operation.

3.2 ESP-bags

Next, we present our extensions to the SP-bags algorithm. Recall that the key difference between AFPL and spawn-sync lies in the flexibility of selecting which of its descendant tasks a parent task can join. The following table shows the update rules for the ESP-bags algorithm. The extensions to SP-bags are highlighted in **bold**.

Async A—fork a new task A	: $S_A \leftarrow \{A\}$, $P_A \leftarrow \emptyset$
Task A returns to ***Parent*** *B*	: $P_B \leftarrow P_B \cup S_A \cup P_A$, $S_A \leftarrow \emptyset$, $P_A \leftarrow \emptyset$
StartFinish F	: $P_F \leftarrow \emptyset$
EndFinish F in a Task B	: $S_B \leftarrow S_B \cup P_F$, $P_F \leftarrow \emptyset$

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 an 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 an 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 therefore 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.

This extension generalizes the SP-bags presented earlier. 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 by 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 done in the original SP-bags algorithm).

3.3 Space overhead

The space overhead of this algorithm is O(1) for each memory location, since we only store the reader and writer task IDs for each memory location. In addition, 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 such a task to check if it is in an S or a P bag as part of some memory access. However, this space is generally insignificant compared to the space needed for each memory location.

3.4 Time overhead

In this algorithm, there are up to two look-ups for every memory access in the program. 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, happen on the disjoint-set data structure that contains all the tasks in the program. Tarjan showed that in 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 [24, 25]. 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 4 (the upper bound for all practical purposes). Since the number of memory accesses dominates the number of tasks in most programs, the total time complexity of the algorithm is proportional to four times the number of memory accesses in the program.

3.5 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 AFPL code in Fig. 2. Suppose that the main task, T_1, starts executing that code. We refer to the finish in line 4 as F_1 and the first instance of the finish in line 15 as F_2. Also, we refer to the first instance of the tasks generated by the asyncs in lines 7, 16, and 17 as T_2, T_3, and T_4, respectively.

Table 1 shows how the S and P bags of the tasks (T_1, T_2, T_3, and T_4) and the P bags of the finishes (F_1 and F_2) are modified by the algorithm as the code in Fig. 2 is executed. Each row shows the status of these S and P bags after the execution of a particular statement in the code. The PC refers to the statement number (from Fig. 2) that is executed. This table only shows the status corresponding to the first iteration of the for loop in line 5. The table also tracks the contents of the writer field of the memory location *B[0]*. The P bags of the tasks T_1, T_2, and T_4 are omitted here since they remain empty through the first iteration of the for loop.

Table 1 ESP-bags example

PC	T_1	F_1	T_2	F_2	T_3		T_4	B[0]
	S	P	S	P	P	S	S	Writer
1	$\{T_1\}$	–	–	–	–	–	–	–
4	$\{T_1\}$	$\emptyset$	–	–	–	–	–	–
7	$\{T_1\}$	$\emptyset$	$\{T_2\}$	–	–	–	–	–
8	$\{T_1\}$	$\emptyset$	$\{T_2\}$	–	–	–	–	T_2
14	$\{T_1\}$	$\{T_2\}$	$\emptyset$	–	–	–	–	T_2
15	$\{T_1\}$	$\{T_2\}$	$\emptyset$	$\emptyset$	–	–	–	T_2
16	$\{T_1\}$	$\{T_2\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{T_3\}$	–	T_2
17	$\{T_1\}$	$\{T_2\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{T_3\}$	$\{T_4\}$	T_2
*18	$\{T_1\}$	$\{T_2\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{T_3\}$	$\{T_4\}$	T_4
19	$\{T_1\}$	$\{T_2\}$	$\emptyset$	$\emptyset$	$\{T_4\}$	$\{T_3\}$	$\emptyset$	T_4
21	$\{T_1\}$	$\{T_2\}$	$\emptyset$	$\{T_4, T_3\}$	$\emptyset$	$\emptyset$	$\emptyset$	T_4
22	$\{T_1, T_4, T_3\}$	$\{T_2\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	T_4

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 the statement in line 8 is executed, the writer field of *B[0]* is set to the current task, T_2. Then, on completion of T_2 in line 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 line 18 (in Task T_4) is executed, the algorithm finds the task in its writer field, T_2, in a P bag (the P bag of F_1), and is reported as a data race. Further, when T_4 completes in line 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 line 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 line 22, the contents of its P bag are moved to the S bag of its parent T_1.

4 Correctness of the ESP-bags algorithm

In this section, we prove the correctness of the ESP-bags algorithm for AFPL. Section 5 shows the extension of ESP-bags for AFIPL. First, we start by defining the Computation Graph for an AFPL program. We then introduce the Dynamic Program Structure Tree (DPST) and how to construct the DPST for a given program with finish and async constructs. The DPST abstraction is used to establish the correctness proof and is not actually constructed by our algorithm. We then discuss the invariants that hold on the DPST when two tasks may-happen-in-parallel (Definition 3) and otherwise. 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; that is, the algorithm is precise and sound for the *given input*.

Definition 1 A Computation Graph (CG), $\Phi(N, E)$, for a schedule Ψ of an AFPL 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[5] in Ψ.
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* [4, 14]. 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 shows an AFPL 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 2 A *continue-edge-only* path in a computation graph is a path in which all the edges are of type *continue*.

Definition 3 Two statement instances s_1 and s_2 in a schedule Ψ of a program may-happen-in-parallel,[6] written as $\text{MHP}(s_1, s_2) = \text{true}$, 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 Ψ.

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

The Dynamic Program Structure Tree (DPST) is a runtime representation of the Program Structure Tree (PST) introduced in [2]. There are some important differences between a DPST and a PST. While the PST is a static data structure for a procedure in a program, the DPST is a dynamic data structure that spans the entire program. Since the DPST is a runtime data structure, it creates one node for every statement instance created during the program execution. A DPST node is one of three types: *finish*, *async*, and *statement*. The other three types of nodes in a PST, *root*, *loop*, and *isolated*, are not present in a DPST. The *root* is not present in a DPST because the implicit finish node in the main method is always the root of a DPST. Since the DPST is a runtime representation, the loops are unrolled and hence there is no need for a *loop* node. Because we restrict ourselves to AFPL in this section, we do not include *isolated* nodes.

Definition 4 A Dynamic Program Structure Tree (DPST), $\Gamma(N, E, \text{Par}, C)$, for a schedule Ψ of an AFPL program P for a given input ξ is an ordered rooted tree where

1. N is a set of nodes, one for every statement instance in Ψ. Every internal node belongs to one of the two types, *finish* and *async*, and every leaf node is of the type *statement*. The root node is always of the type *finish*, and it corresponds to the implicit finish surrounding the body of *main()* in the program.
2. Par defines the parent relation between nodes in Γ as follows:
   - $\text{Par}(n) = \alpha$, for every *finish* node α and every node n that satisfies the following condition: α is the immediately enclosing finish of n, and there is a *continue-edge-only* path from the start of the *finish* α to n in the computation graph corresponding to Ψ.

[5] We refer to an execution of a statement as either a dynamic statement instance or a statement instance.

[6] The definition of the static version of MHP can be found in [2].

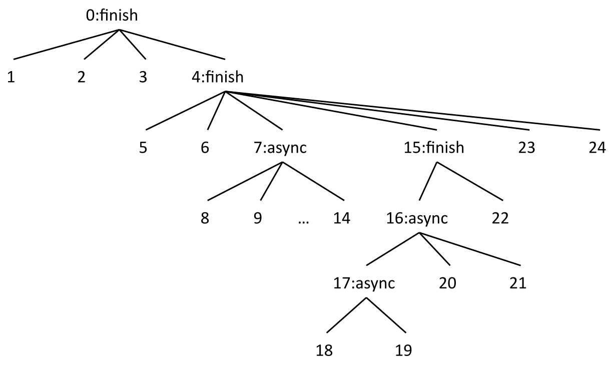

Fig. 5 Dynamic Program Structure Tree for the program and computation graph in Fig. 2

– $\mathrm{Par}(n) = \beta$, for every *async* node β and every node n that satisfies the following condition: n belongs to the task corresponding to β, and the immediately enclosing finish of n is not in β.

3. E is a set of tree edges that are obtained as follows: $E = \{(n_1, n_2) : \mathrm{Par}(n_2) = n_1\}$
4. C defines the children relation among nodes in Γ as follows: $C(\alpha) = \{n : \mathrm{Par}(n) = \alpha\}$
   Note that the set $C(\alpha)$ is ordered, which reflects the order of the children for every parent in Γ.

Definition 5 The Lowest Common Ancestor of two nodes s_1 and s_2, $\mathrm{LCA}(s_1, s_2)$, in a tree Γ is the node φ that is an ancestor[7] of both s_1 and s_2 with the greatest depth[8] in Γ.

Definition 6 In an ordered tree Γ, a node s_1 is said to be *to the left of* a node s_2 if and only if s_1 appears before s_2 in the inorder traversal of Γ. The relation *to the left of* is defined on two nodes s_1 and s_2 if and only if $\mathrm{LCA}(s_1, s_2) \neq s_1$ and $\mathrm{LCA}(s_1, s_2) \neq s_2$.

The set of edges from every internal node to its children in a DPST are arranged to reflect the program order, i.e., if a statement instance s_1 executes before a statement instance s_2 in a schedule Ψ, the node s_1 will appear *to the left of* the node s_2 in the DPST corresponding to Ψ.

Figure 5 shows the DPST for the AFPL program in Fig. 2. Note that the finish node 0 is the implicit finish in the body of *main()* (assuming that the code shown in Fig. 2 is the body of the main).

Theorem 1 *Every data-race-free AFPL 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 AFPL 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 ξ. Also every statement in P belongs to the same task across all possible executions of P for input ξ. Hence in every DPST of P that corresponds to different executions of P for an input ξ, the parent-child relationship

[7] A node is considered both an ancestor and a descendant of itself.

[8] The depth of a node in a tree is the length of the path from the root to the node.

is unique between nodes corresponding to all the instances of finish, async, and statements in P. In other words, if node α is the parent of node β in a DPST of P, then α is the parent of β in every DPST of P for an input ξ.

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 AFPL program with finish and async constructs for a given input. □

Theorem 2 *The sequential depth-first execution of an AFPL 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 AFPL 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. □

Theorem 3 *Let Γ be the DPST corresponding to the sequential depth-first execution Ψ of an AFPL program P with input ξ. Let s_1 and s_2 be two nodes in Γ. Let s_1 be to the left of s_2 in Γ. Let* $\mathrm{LCA}(s_1, s_2) = \varphi$, $\varphi \neq s_1$, $\varphi \neq s_2$. *Let A_1 denote the DPST ancestor of s_1 that is the child of φ. The following conditions hold*:

1. $\mathrm{MHP}(s_1, s_2) =$ *true if and only if A_1 is an async node.*
2. $\mathrm{MHP}(s_1, s_2) =$ *false if and only if A_1 is a finish node.*

Proof Let Φ denote the computation graph corresponding to Γ.

1. *if*: Let A_1 be an async node. Let F_1 denote the immediately enclosing finish of A_1. In Φ, any path starting at s_1 (that goes out of A_1) has to go to the end of A_1 and then directly to the end of F_1. But s_2 is outside the async A_1 and inside the finish F_1. Hence there can be no path from s_1 to s_2 in Φ. Since s_1 is to the left of s_2 in Γ, it follows from Theorem 2 that s_1 executes before s_2 in Ψ. Hence there can be no path from s_2 to s_1 in Γ. Thus, $\mathrm{MHP}(s_1, s_2) = \text{true}$.

   only if: Let $\mathrm{MHP}(s_1, s_2) = \text{true}$. By definition, there can be no path from s_1 to s_2 and from s_2 to s_1 in Φ. If A_1 is a finish node, then there is a path in Φ starting at s_1 that goes to the end of A_1 and then to s_2. Hence A_1 can not be a finish node. A_1 can not be a statement node because all statement nodes are leaf nodes. Thus A_1 is an async node.
2. *if*: Let A_1 be a finish node. There is a path in Φ that starts at s_1, goes to the end of A_1, and then to s_2. Hence $\mathrm{MHP}(s_1, s_2) = \text{false}$.

   only if: Let $\mathrm{MHP}(s_1, s_2) = \text{false}$. By definition, there is path from s_1 to s_2 or from s_2 to s_1 in Φ. Since s_1 is to the left of s_2 in Γ, it follows from Theorem 2 that s_1 executes before s_2 in Ψ. Hence there can be no path from s_2 to s_1 in Γ. If A_1 is an async node, then there can be no path from s_1 to s_2 as well. A_1 must be a finish node. □

Theorem 4 *Let Γ be the DPST corresponding to the sequential depth-first execution of an AFPL program P with an input ξ. Let statement instance s_1 be to the left of statement instance s_2 in Γ. During the sequential depth-first execution of P with input ξ in the ESP-bags algorithm, when s_2 is being executed, the ID of the task τ 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 Γ be the DPST of P for input ξ. Let $\mathrm{LCA}(s_1, s_2) = \varphi$. Consider the case when $\varphi \neq s_1$ and $\varphi \neq s_2$. If $\varphi = s_1$, then s_1 has to be an internal node, i.e., a finish or an async node. This case is not necessary because we are only interested in the MHP relation between two statement instances. The same holds when $\varphi = s_2$.

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 φ. We know from Theorem 3 that A_1 must be an async node. According to the rules of the ESP-bags algorithm from Sect. 3.2, 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 τ that owns s_1 will be put in the P-bag of φ, which is the parent of A_1 in Γ. The ID of τ will stay in the P-bag of φ until the execution completes the execution of the subtree under φ. By definition of φ 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 τ will be in a P-bag.

only if: Let us assume that the ID of the task τ 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 φ.

Case 1: A_1 is a finish node. In this case τ will be in a S bag when s_2 is executed, according to the rules from Sect. 3.2.

Case 2: A_1 is the node corresponding to s_1. Again in this case τ will be in a S bag when s_2 is executed, according to the rules from Sect. 3.2.

Hence A_1 can neither be a finish node nor the node corresponding to s_1. A_1 must be an async node. Following from Theorem 3, s_1 may-happen-in-parallel with s_2. □

Theorem 5 (Precision and Soundness) *The ESP-bags algorithm detects a data race in an AFPL program for a given input if and only if a data race exists.*

Proof Let us consider an AFPL program P that is executed with an input ξ. Let Γ denote the DPST corresponding to the sequential depth-first execution of P with input ξ.

if: Let us assume that there is a data race in some schedule of P with input ξ. 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 the ESP-bags's sequential depth-first execution of P. Thus s_1 will be to the left of s_2 in Γ. From Theorem 4 it follows that when s_2 is executed, the task τ 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 (according to the rules in Fig. 4). In this case, since τ 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 (according to the rules in Fig. 4). In this case, since τ 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 ξ. According to the rules of the algorithm in Fig. 4, a data race will be signaled only in two cases:

```
Isolated Read of location L by Task t:
    If L.writer is in a P—bag then Data Race;
    If L.isolatedReader is in a S—bag then L.isolatedReader = t;
```

```
Isolated Write of location L by Task t:
    If L.writer is in a P—bag or L.reader is in a P—bag
        then Data Race;
    If L.isolatedWriter is in a S—bag then L.isolatedWriter = t;
```

```
Read location L by Task t:
    If L.writer is in a P—bag or L.isolatedWriter is in a P—bag
        then Data Race;
    If L.reader is in a S—bag then L.reader = t;
```

```
Write location L by Task t:
    If L.writer is in a P—bag or L.reader is in a P—bag
            or L.isolatedWriter is in a P—bag or L.isolatedReader is in a P—bag
        then Data Race;
    L.writer = t;
```

Fig. 6 ESP-bags algorithm for AFIPL

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 τ, is in a P-bag. It follows from Theorem 4 that s_1 may-happen-in-parallel with s_2. Hence, there is a data race in some execution of P with input ξ.

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 τ, is in a P-bag. It follows from Theorem 4 that s_1 may-happen-in-parallel with s_2. Hence, there is a data race in some execution of P with input ξ. □

In summary, if there are races in the program for the given input, ESP-bags will find them and will never report races that do not exist.

5 Handling isolated blocks

In this section, we describe an extension to the ESP-bags algorithm for handling isolated sections. Isolated sections 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 an AFIPL program contains isolated sections, the data race detector must check for conflicts between isolated and non-isolated accesses to the same memory location that may execute in parallel. If an access a_1 to a memory location L in an isolated section conflicts with another access a_2 to L in a non-isolated section, then it is a data race.

Note that, accesses within isolated sections do not conflict with other accesses within isolated sections because of the mutual exclusion semantics guaranteed by isolated constructs. Hence, these isolated accesses themselves cannot cause data races.

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

Fig. 7 An example AFIPL program that depicts a scenario in which the ESP-bags algorithm is not sound

```
 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
```

We need to handle reads and writes in *isolated* blocks differently than *non-isolated* operations. Figure 6 shows the required steps during each of the operations: *read*, *write*, *isolated-read*, and *isolated-write*.

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

The problem is that with isolated sections, 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 sections in the program do not commute. In this case, for the same input, the isolated sections may produce a different result in some parallel schedule compared to the result produced in a depth-first execution, and there may be some code conditioned on this result that 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 7 shows an example AFIPL program that depicts a scenario in which the ESP-bags algorithm is not sound in the presence of isolated sections. In this example, during the depth-first execution of our algorithm, the isolated block in line 3 executes 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 does 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 sections 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 sections are used only with very small scopes, and it is easy to show that they commute (for instance, they use only commutative operations such as addition, to increment a counter).

In summary, when the isolated sections commute, the ESP-bags algorithm is precise and sound for the given input. When the isolated sections do not commute, it is precise but not sound.

6 Optimizations

The ESP-bags algorithm is implemented as a *Java* library. Recall that the ESP-bags algorithm requires that action is taken on every read and write to a shared memory location. It is during these actions that the algorithm checks if the current task can race with the task recorded in the reader or writer fields of the memory location. Now, to test a given program for determinism using the ESP-bags algorithm, we need 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. It would be naive to instrument every access to every shared memory location because some of these instrumentations may be redundant; i.e., removing them will not affect the process of checking for data races in the program. Because some read and write operations are guaranteed to never cause any additional data races in the program, such operations need not be instrumented.

As mentioned earlier, because the ESP-bags algorithm also keeps track of the *finish*, *async*, and *isolated* blocks in the program, it requires instrumentations for the start and end of every such block in the program. These instrumentations are all necessary to maintain the structure of parallelism at runtime in the ESP-bags algorithm.

In this section, we describe the static analyses that can be used to reduce the instrumentation and hence improve the runtime performance of the instrumented program. We also include an example that depicts how each of these static analyses are used to eliminate instrumentation points. Figure 8 shows a program in AFPL with all its read and write operations instrumented (*DJCRead* and *DJCWrite* 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.

6.1 Main task check elimination in sequential code regions

The first static optimization aims to eliminate redundant instrumentation points that are added in the sequential code regions. 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 trivial to show 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 AFPL program, the sequential code regions are the regions of the program that are outside the outermost *finish* blocks[9] and are executed by the *main task*. Thus, in an AFPL program, there is no need to instrument the read and write operations in such sequential code regions of the main task.

Figure 9 shows the result of eliminating the instrumentation points in the sequential code regions of the program in Fig. 8. The program in Fig. 8 contains a write to a heap location *p.x* in line 4 that 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.

6.2 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 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 inter-procedural side-effect analysis to detect potential write operations to

[9]This is assuming there are no *async*s outside any *finish* in the program. If there are any such *async*s, then the only sequential code regions in the program are the regions outside the outermost *finish* and before the first such *async*.

Fig. 8 An example AFPL program with all read and write operations instrumented

```
int[] A, B; Foo p;
... ...
DJCWrite(p, x);
p.x = 0;
finish {
  for (int i=0; i<size; i++ ) {
    final int ind = i;
    async {
      DJCRead(A, ind);
      DJCRead(B, ind);
      DJCWrite(p, x);
      p.x = A[ind] + B[ind];
      Foo q = new Foo();
      for (int j=0; j<ind; j++) {
        DJCRead(p, x);
        DJCWrite(q, x);
        q.x = p.x + 1;
        DJCRead(q, y);
        DJCWrite(B, j);
        B[i] = q.y + ind;
      }
    }
  }
}
```

Fig. 9 After applying the main task check elimination optimization on the program in Fig. 8

```
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         DJCRead(A, ind);
9         DJCRead(B, ind);
10        DJCWrite(p, x);
11        p.x = A[ind] + B[ind];
12        Foo q = new Foo();
13        for (int j=0; j<ind; j++) {
14          DJCRead(p, x);
15          DJCWrite(q, x);
16          q.x = p.x + 1;
17          DJCRead(q, y);
18          DJCWrite(B, j);
19          B[i] = q.y + ind;
20        }
21      }
22    }
23  }
```

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 Sect. 6.1.

The result for applying this optimization on the program in Fig. 9 is shown in Fig. 10. There is no write to array A within the parallel regions of the program in Fig. 9, so the instrumentation in line 8 corresponding to the read of A in line 11 can be removed.

6.3 Escape analysis

The input program may include many parallel tasks. A determinacy race occurs in the program only when two or more tasks access a shared memory location and at least one of them

Fig. 10 After applying the read-only check optimization on the program in Fig. 9

```
 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        DJCRead(B, ind);
 9        DJCWrite(p, x);
10        p.x = A[ind] + B[ind];
11        Foo q = new Foo();
12        for (int j=0; j<ind; j++) {
13          DJCRead(p, x);
14          DJCWrite(q, x);
15          q.x = p.x + 1;
16          DJCRead(q, y);
17          DJCWrite(B, j);
18          B[j] = q.y + ind;
19        }
20      }
21    }
22  }
```

Fig. 11 After applying the escape analysis and check elimination optimization on the program in Fig. 10

```
int[] A, B; Foo p;
...  ...
p.x = 0;
finish {
  for (int i=0; i<size; i++ ) {
    final int ind = i;
    async {
      DJCRead(B, ind);
      DJCWrite(p, x);
      p.x = A[ind] + B[ind];
      Foo q = new Foo();
      for (int j=0; j<ind; j++) {
        DJCRead(p, x);
        q.x = p.x + 1;
        DJCWrite(B, j);
        B[j] = q.y + ind;
      }
    }
  }
}
```

is a write. Suppose an object is created inside a task, and it never escapes that task; because no other task can access this object, it cannot lead to a determinacy race. In order to ensure the task-local attribute, the compiler performs an inter-procedural analysis that determines 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 Fig. 10 is created in line 11 within a task and it never escapes this task. No access to q can lead to a determinacy race, so the instrumentation points in lines 14 and 16 corresponding to access to q are eliminated. The resulting program is shown in Fig. 11.

6.4 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

Fig. 12 After applying the loop invariant check elimination optimization on the program in Fig. 11

```
 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        DJCRead(B, ind);
 9        DJCWrite(p, x);
10        p.x = A[ind] + B[ind];
11        Foo q = new Foo();
12        if (ind > 0)
13          DJCRead(p, x);
14        for (int j=0; j<ind; j++) {
15          q.x = p.x + 1;
16          DJCWrite(B, j);
17          B[j] = q.y + ind;
18        }
19      }
20    }
21  }
```

Fig. 13 After applying the read/write check elimination optimization on the program in Fig. 12

```
 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        DJCRead(B, ind);
 9        DJCWrite(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          DJCWrite(B, j);
15          B[j] = q.y + ind;
16        }
17      }
18    }
19  }
```

because other instrumentations will only add to the overhead with redundant 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 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 Fig. 11, the program contains a read of *p.x* in line 13 that is inside a sequential loop. 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 Fig. 12. Note the test for the non-zero trip count in line 12 guards this instrumentation outside the loop.

6.5 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 algorithm presented in this paper. Intuitively, if a read to a memory location M in a task τ causes a determinacy race, then a write to M in τ will definitely cause a determinacy 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). In practice, this rule tends to apply to fewer situations than the previous rule, because computation of postdominance includes the possibility of exceptional control flow.

Consider the program in Fig. 12 that contains 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 the latter is not a write, line 13's instrumentation can be eliminated.

7 Evaluation

We now present the experimental results of our race detection algorithm. We evaluated the ESP-bags algorithm on eight Java Grande Forum (JGF) benchmarks, three Shootout benchmarks, and one EC2 challenge benchmark, listed in Table 2. Though we performed our experiments on different sizes of the JGF benchmarks, we only report the results of the maximum size in each case. We were unable to obtain the results of size B for MolDyn since both versions (original and instrumented) ran out of memory. All the benchmarks were written in HJ using only the AFIPL constructs and are available from [15].

The ESP-bags algorithm was implemented as a Java library for detecting data races in HJ programs containing async, finish, and isolated constructs. The benchmarks written in HJ were instrumented for race detection during a bytecode-level transformation pass implemented on HJ's Parallel Intermediate Representation (PIR) [27]. The PIR extends Soot's Jimple IR [26] with parallel constructs such as async, finish, and isolated. The instrumentation pass adds the necessary calls to our race detection library at async and finish boundaries and also on reads and writes to shared memory locations.

We report the performance results of our experiments on a 16-way (quad-socket, quad-core per socket) Intel Xeon 2.4 GHz system with 30 GB memory, running Red Hat Linux (RHEL 5). The JVM used is the Sun Hotspot JDK 1.6 with a maximum heap size of 3 GB.

Results of ESP-bags algorithm Table 3 shows the results of applying the ESP-bags algorithm to our benchmarks. The table gives the original execution time for each benchmark without any instrumentation. It also shows the slowdown of the benchmark when instrumented for the ESP-bags algorithm, with and without the optimizations described in Sect. 6. The outcome of the ESP-bags algorithm is also included in the table and shows there are no data races in any of the benchmarks. The same was observed for all input sizes. Hence all the benchmarks are free of data races for the inputs considered. Note that though RayTracer has some *isolated* conflicts, it is free of data races since there were no conflicts between isolated and non-isolated accesses.

Table 2 List of benchmarks evaluated

Source	Benchmark	Description
JGF (Sect. 2)	Series	Fourier coefficient analysis
	LUFact	LU Factorization
	SOR	Successive over-relaxation
	Crypt	IDEA encryption
	Sparse	Sparse Matrix multiplication
JGF (Sect. 3)	MolDyn	Molecular Dynamics simulation
	MonteCarlo	Monte Carlo simulation
	RayTracer	3D Ray Tracer
Shootout	Fannkuch	Indexed-access to tiny integer-sequence
	Fasta	Generate and write random DNA sequences
	Mandelbrot	Generate Mandelbrot set portable bitmap file
EC2	Matmul	Matrix Multiplication (two 1000 × 1000 double matrix)

Table 3 Slowdown of ESP-bags algorithm

Benchmark	Number of asyncs	Time (s)	ESP-bags Slowdown Factor		Result
			w/o opts	w/opts	
Crypt—C	13000000	15.24	7.63	7.29	No Data Races
LUFact—C	1600000	15.19	12.45	10.08	No Data Races
MolDyn—A	510000	45.88	10.57	3.93	No Data Races
MonteCarlo—B	300000	19.55	1.99	1.57	No Data Races
RayTracer—B	500	38.85	11.89	9.48	No Data Races (Isolated conflict)
Series—C	1000000	1395.81	1.01	1.00	No Data Races
SOR—C	200000	3.03	14.99	9.05	No Data Races
Sparse—C	64	13.59	12.79	2.73	No Data Races
Fannkuch	1000000	7.71	1.49	1.38	No Data Races
Fasta	4	1.39	3.88	3.73	No Data Races
Mandelbrot	16	11.89	1.02	1.02	No Data Races
Matmul	1000	19.59	6.43	1.16	No Data Races
Geo Mean			4.86	3.05	

ESP-bags slowdown On average, the slowdown of the benchmarks with the ESP-bags algorithm is 4.86× without optimization. When all the static optimizations are applied, the average slowdown drops to 3.05×. The slowdown of all benchmarks except LUFact is less than 10×. The slowdown for benchmarks like MolDyn, MonteCarlo, and Sparse is less than 5×. There is no slowdown in the case of Series because most of the code uses stack variables. In *HJ* none of the stack variables can be shared across tasks, so we do not instrument any access to these variables. On the other hand, the slowdown for SOR and RayTracer benchmarks is around 9×.

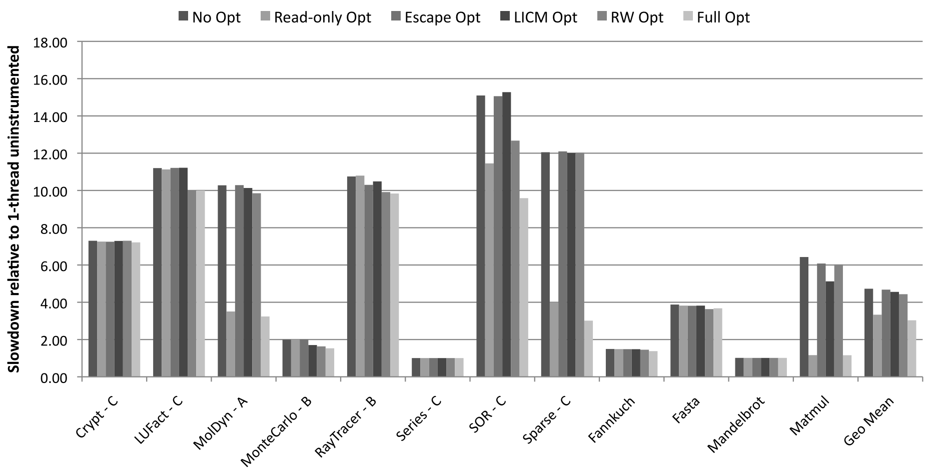

Fig. 14 Breakdown of static optimizations

Performance of optimizations We now discuss the effects of the compiler optimizations 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, loop invariant check motion, and read/write check elimination. The compile time overhead of instrumenting the input program for race detection with ESP-bags is 2% on an average. On the other hand, the compile time overhead of the static optimizations is 25% on an average across the benchmarks considered. This is because of the extra time required to perform the static analyses needed to eliminate redundant instrumentations.

As evident from the table, some of the benchmarks, such as SOR, Sparse, MolDyn, and Matmul, greatly benefit from the optimizations, with a maximum reduction in slowdown of about 78% for Sparse. On the other hand, for other benchmarks the reduction is relatively small. The optimizations do not reduce the slowdown much for Crypt and LUFact because very few instrumentation points are eliminated. In the cases of MonteCarlo and RayTracer, though a good number of instrumentation points are eliminated, a significant fraction of them still remain, so there is not much performance improvement in these benchmarks due to optimizations. On average, there is a 37% reduction in the slowdown of the benchmarks due to these optimizations.

Breakdown of the optimizations We now describe the effects of each of the static optimizations separately on the performance of the benchmarks. Figure 14 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 Sect. 6 is applied to all the versions discussed here, including the unoptimized version, because it is a basic optimization that avoids excessive instrumentations.

The read-only check elimination performs much better than the other optimizations for most of the benchmarks, such as MolDyn, SOR, and SparseMatmult. This is because in these benchmarks the parallel regions include reads to many arrays that 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 fully optimized version. The read-write optimization works well in the case of SOR but does not have much effect on other benchmarks. The loop invariant check motion helps improve the performance of MonteCarlo the most, and the escape analysis does not seem to help any of these benchmarks significantly.

Note that the performance of these four static optimizations do not directly add up to the performance of the fully optimized code. Because some of these optimizations create more chances for other optimizations, their combined effect is much more than their sum. For example, the loop invariant check motion creates more chances for the read-only and read-write optimization. So, when these two optimizations are performed after loop invariant check motion, their effect would be more than that is shown here. Finally, we only evaluated the performance of these optimizations on the set of benchmarks shown here. For a different set of benchmarks, their effects may vary. However, we believe that these static optimizations, when combined, can generally improve the performance of most of the benchmarks.

8 Related work

The Cilk paper [11] introduces SP-bags for spawn-sync computations. We generalize that algorithm so that it also applies to async-finish computations while still being able to check spawn-sync programs. An extension to SP-bags was proposed by Cheng et al. [9] to handle locks in Cilk programs. Their approach includes a data race detection algorithm for programs that satisfy a particular locking discipline. However, the slowdown factors reported in [9] were in the 33×–78× range for programs that follow their locking discipline, and up to 3700× for programs that don't. In this work, we detect data races in programs with async, finish, and isolated constructs. We outline and implement a range of static optimizations to reduce the slowdown factor to 3.05× on average.

A recent result on detecting data races by Flanagan et al. [12] (FastTrack) reduces the overhead of using vector clocks during data race detection. Their technique focuses on the more general setting of fork-join programs. The major problem with using vector clocks for race detection is that the space required for vector clocks is linear in the number of threads in the program, and hence any vector clock operation also takes time linear in the number of threads. In a program containing millions of tasks that can run in parallel, it is not feasible to use vector clocks to detect data races (if we directly extend vector clocks to tasks). Though FastTrack reduces this space (and thus the time for any vector clock operation) to a constant by using epochs instead of vector clocks, it needs vector clocks whenever a memory location has shared read accesses. Even a few such instances would make it infeasible for programs with millions of parallel tasks. On the other hand, our approach requires only a constant space overhead for every shared memory location and a time overhead proportional to the inverse Ackermann function for every shared memory access.

The other approach to use FastTrack for task-parallel languages is to fix the threads the program runs on to a small number (say eight) and use vector clocks of this fixed size. With this change, FastTrack would just check for data races in a particular schedule of a program. Our approach can guarantee the non-existence of data races for all possible schedules of a given input. However, the price we have to pay for this guarantee is that we have to execute the given program sequentially. Given that this needs to be done only during the development stage, we feel our approach is of value.

Sadowski et al. [23] propose a technique for checking determinism by using interference checks based on happens-before relations. This involves detecting conflicting races in threads that can run in parallel. Though they can guarantee the non-existence of races in all possible schedules of a given input, the fact that they use vector clocks makes these infeasible in a program with millions of tasks that can run in parallel.

The static optimizations that we use to eliminate the redundant instrumentations and thus reduce the overhead is similar to the compile-time analyses proposed by Mellor-Crummey [21]. 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 dependences 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 that are guaranteed to have already been instrumented in that task.

The Clara framework [7] also performs static analyses to reduce the overhead of runtime verification tools. It is a general framework for statically analyzing runtime monitors, which uses a finite-state-machine model of the property and generates runtime monitors in the form of AspectJ aspects. This framework has been used to eliminate all the runtime monitors for 68% of the cases considered, thereby completely obviating the need for runtime monitoring. In other cases, it reduces the overhead of the runtime monitors, similar to our static optimizations. To use this framework for our static optimizations, we need to specify data race detection as a finite-state-machine model. It would be interesting to see if Clara can eliminate all the runtime monitors for race detection. We would like to explore this in future models of our race detector.

Our static optimization that moves loop invariant checks out of the loop (outlined in Sect. 6.4) is similar in effect to the stutter-equivalent loop transformation described in [22]. They present a general framework for optimizing the monitoring of loops relative to a property. Their framework allows monitors inside a loop to be processed in a constant time rather than time that is proportional to the number of iterations of the loop. This is achieved by calculating the loop iteration after which the remaining iterations are said to be stutter relative to the property under consideration and transforming the loop accordingly to reduce the overhead of runtime monitoring. Again, this requires that the property to be monitored is specified as a finite-state-machine. In future, we plan to evaluate this approach to see if it reduces the overhead of our race detector even further.

9 Conclusion

In this paper we proposed a precise, sound, and efficient dynamic data-race detection algorithm called ESP-bags (i.e., there are neither any false positives nor any false negatives). ESP-bags supports both the async-finish parallel programming model as well as the spawn-sync model used in Cilk.

We implemented ESP-bags in a tool called TASKCHECKER and augmented it with a set of compiler optimizations that reduce the incurred average overhead by 37% with respect to the unoptimized version. Evaluation of TASKCHECKER on a suite of 12 benchmarks shows that the dynamic analysis introduces an average slowdown of 4.86× without compiler optimizations, and 3.05× with compiler optimizations, making the tool suitable for practical use.

In future, we plan to investigate the applicability of ESP-bags to the fork-join concurrency model. Also, we plan to explore data race detection by executing the input program in parallel, which is not possible with ESP-bags algorithm.

Acknowledgements We would like to thank Jacob Burnim and Koushik Sen from UC Berkeley, Jaeheon Yi and Cormac Flanagan from UC Santa Cruz, and John Mellor-Crummey from Rice University for their feedback on an earlier version of this paper. We thank Charles Leiserson for pointing out the conditional sync example. We are grateful to Jill Delsigne for her assistance with copy-editing the final version of this paper. We also thank the US-Israel Binational Foundation (BSF) for their support.

References

1. Agarwal S, Barik R, Bonachea D, Sarkar V, Shyamasundar RK, Yelick K (2007) Deadlock-free scheduling of X10 computations with bounded resources. In: SPAA '07: Proceedings of the 19th symposium on parallel algorithms and architectures. ACM, New York, pp 229–240
2. Agarwal S, Barik R, Sarkar V, Shyamasundar RK (2007) May-happen-in-parallel analysis of $\times 10$ programs. In: PPoPP '07: Proceedings of the 12th symposium on principles and practice of parallel programming. ACM, New York, pp 183–193
3. Barik R, Budimlic Z, Cave V, Chatterjee S, Guo Y, Peixotto D, Raman R, Shirako J, Tasirlar S, Yan Y, Zhao Y, Sarkar V (2009) The habanero multicore software research project. In: OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on object oriented programming systems languages and applications, New York, NY, USA. ACM, New York, pp 735–736
4. Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y (1995) Cilk: an efficient multithreaded runtime system. In: Proceedings of the fifth ACM SIGPLAN symposium on principles and practice of parallel programming, PPoPP, Oct 1995, pp 207–216
5. Blumofe RD, Leiserson CE (1999) Scheduling multithreaded computations by work stealing. J ACM 46(5):720–748
6. Bocchino R, Adve V, Adve S, Snir M (2009) Parallel programming must be deterministic by default. In: First USENIX workshop on hot topics in parallelism (HOTPAR 2009)
7. Bodden E, Lam P, Hendren L (2010) Clara: a framework for statically evaluating finite-state runtime monitors. In: 1st international conference on runtime verification (RV), Nov 2010. LNCS, vol 6418. Springer, Berlin, pp 74–88
8. Charles P, Grothoff C, Saraswat VA, Donawa C, Kielstra A, Ebcioglu K, von Praun C, Sarkar V (2005) X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the twentieth annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, OOPSLA, Oct, pp 519–538
9. Cheng G-I, Feng M, Leiserson CE, Randall KH, Stark AF (1998) 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, June 28–July 2 1998, pp 298–309
10. Dijkstra EW Cooperating sequential processes. 65–138
11. Feng M, Leiserson CE (1997) Efficient detection of determinacy races in Cilk programs. In: SPAA '97: proceedings of the ninth annual ACM symposium on parallel algorithms and architectures. ACM, New York, pp 1–11
12. Flanagan C, Freund SN (2009) Fasttrack: efficient and precise dynamic race detection. In: PLDI '09: proceedings of the 2009 ACM SIGPLAN conference on programming language design and implementation. ACM, New York, pp 121–133
13. Frigo M, Leiserson CE, Randall KH (1998) The implementation of the Cilk-5 multithreaded language. In: PLDI'98, NY, USA, 1998. ACM, New York, pp 212–223
14. Guo Y, Barik R, Raman R, Sarkar V (2009) Work-first and help-first scheduling policies for async-finish task parallelism. In: IPDPS '09: proceedings of the international symposium on parallel&distributed processing. IEEE Computer Society, Washington, pp 1–12
15. Habanero Java http://habanero.rice.edu/hj
16. Larus JR, Rajwar R (2006) Transactional memory. Morgan and Claypool, San Francisco
17. Lea D (2000) A java fork/join framework. In: JAVA '00: proceedings of the ACM 2000 conference on Java Grande. ACM, New York, pp 36–43
18. Lee EA (2006) The problem with threads. Computer 39(5):33–42
19. Lee JK, Palsberg J (2010) Featherweight $\times 10$: a core calculus for async-finish parallelism. In: PPoPP '10: proceedings of the 15th ACM SIGPLAN symposium on principles and practice of parallel computing. ACM, New York, pp 25–36
20. Leijen D, Schulte W, Burckhardt S (2009) The design of a task parallel library. In: OOPSLA '09: proceeding of the 24th ACM SIGPLAN conference on object oriented programming systems languages and applications. ACM, New York, pp 227–242
21. Mellor-Crummey J (1993) 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, 1993. ACM, New York, pp 129–139

22. Purandare R, Dwyer MB, Elbaum S (2010) Monitor optimization via stutter-equivalent loop transformation. In: Proceedings of the ACM international conference on object oriented programming systems languages and applications, New York, NY, USA, 2010, OOPSLA '10. ACM, New York, pp 270–285
23. Sadowski C, Freund SN, Flanagan C (2009) SingleTrack: A dynamic determinism checker for multithreaded programs. In: Programming languages and systems. Lecture notes in computer science, vol 5502. Springer, Berlin, pp 394–409
24. Tarjan RE (1975) Efficiency of a good but not linear set union algorithm. J ACM 22:215–225
25. Tarjan RE (1983) Data structures and network algorithms. Society for Industrial and Applied Mathematics, Philadelphia
26. Vallée-Rai R et al (1999) Soot—a Java optimization framework. In: Proceedings of CASCON 1999, pp 125–135
27. Zhao J, Sarkar V (2011) Intermediate language extensions for parallelism. In: VMIL'11, pp 333–334