Efficient Data Race Detection for Async-Finish Parallelism

Raghavan Raman Jisheng Zhao Vivek Sarkar
Rice University
Martin Vechev Eran Yahav
IBM T. J. Watson Research Center
{*raghav,jisheng.zhao, vsarkar*}*@rice.edu,* {*mtvechev, eyahav*}*@us.ibm.com*

Abstract. One of the major productivity hurdles for parallel programming is *non-determinism* — a parallel program may yield different results on different executions with the same input, depending on the order in which operations are interleaved. A major source of non-determinism is *data races*, and checking for the absence of data races is an important candidate for runtime verification. Past work on data race detection includes different techniques for different programming models such as SPMD, fork-join, and monitors. However, the runtime overheads incurred by past techniques are still prohibitively large (often a slowdown of $10\times$ or larger) for use in mainstream software development.

In this paper, we present a tool called TASKCHECKER that performs efficient runtime verification of the *data race freedom* property for async-finish task-parallel programs. The async and finish constructs are at the core of languages such as X10 and HJ, and generalize the spawn-sync constructs used in Cilk and the task-parallel constructs in OpenMP 3.0. Unlike programs written with traditional fork and join constructs, async-finish programs are guaranteed to be deadlock-free.

Our tool is based on runtime verification and is sound for a given *input*: if a potential data race exists for that input, the tool will report it. Further, the tool uses a range of static optimizations to reduce the overhead of the dynamic analysis.

We have evaluated TASKCHECKER on a suite of 12 benchmarks. Our experimental results indicate that our approach has a very reasonable overhead in practice, incurring an average slowdown of $3.05\times$ compared to a serial execution in the optimized case.

1 Introduction

Designing and implementing correct and efficient parallel programs is a notoriously difficult task, and yet, with the proliferation of multi-core processors, parallel programming will need to play a central role in mainstream software development. One of the main difficulties in parallel programming is that a programmer is often required to explicitly reason about the interleavings of operations in their program. The vast number of interleavings makes this task difficult even for small programs, and intractable for sizeable applications.

Unstructured and low-level frameworks such as Java threads allow the programmer to express rich and complicated patterns of parallelism, but also make it easy to

get things wrong. We believe that the arguments made by Dijkstra in favor of structured programming (and against the goto statement) [6] are just as relevant for parallel programming:

> *. . . our intellectual powers are rather geared to master static relations and [. . .] our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do [. . .] our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.*

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 towards structuring parallel computations, starting with constructs such as *cobegin-coend* and *monitors*. Recently, additional support for fork-join task parallelism has been added in the form of libraries [9, **?**] to existing programming environments and languages such as Java and .NET. Although fork-join parallel frameworks allow for a more disciplined style of parallel programming, they are still fairly unstructured. For instance, it is easy to obtain computations that deadlock by having two tasks join on each other.

Parallel languages such as Cilk [2], X10 [5], and Habanero Java (HJ) [13] 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 centered around the flexibility of choosing which tasks a given task can join to. The async-finish computations that we consider generalize the more restricted spawn-sync computations of Cilk, and similarly, have the desired property of being deadlock-free [11] (unlike unrestricted fork-join computations).

Data Race Detection To simplify reasoning about parallel programs, it is desirable to reduce the number of interleavings that a programmer has to consider [10, 4]. One way to achieve that is to require parallel programs to be *deterministic*. A major source of non-determinism is *data races*, which is why we focus on data race detection in this paper. In the absence of data races, all parallel programs with *async* and *finish* constructs are guaranteed to be deterministic. If *isolated* (atomic) constructs are included, then the programmer only needs to worry about reorderings of isolated blocks to establish determinism, and no other interleaving.

We present an efficient dynamic analysis tool that checks determinism of async-finish style parallel computations. These constructs form the core of the larger X10, HJ and Cilk parallel languages. Using async, finish and atomic, 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 the traditional Feng and Leiserson's SP-bags algorithm [7] which was designed for checking determinism of spawn-sync Cilk programs. The reason why the original algorithm cannot be applied directly to async-finish

style of programming is that this model allows for a superset of the executions allowed by the traditional spawn-sync Cilk programs. As with SP-bags, our analysis of programs with async-finish parallelism is sound for a given input: if a determinism violation exists for that input, regardless of the way parallel tasks interleave, our tool will report that violation.

Main Contributions To the best of our knowledge, this is the first detailed study of the problem of determinism checking 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 efficiently checking determinism of structured async-finish parallel programs. Our algorithm generalizes the classical SP-bags algorithm designed for the more restricted spawn-sync Cilk model.
- An implementation of our dynamic analysis in a tool named TASKCHECKER.
- Novel compiler optimizations to reduce the overhead incurred by the dynamic analysis algorithm. We show that our analysis reduces the overhead by $1.59\times$ on average for the benchmarks used in our evaluation.
- An evaluation of TASKCHECKER on a suite of 12 benchmarks written in the HJ programming language[1]. We show that for these benchmarks, TASKCHECKER is able to check determinism with an average (geometric mean) slowdown of $4.86\times$ in the absence of compiler optimizations, and $3.05\times$ with compiler optimizations, compared to a sequential execution.

2 Background

In this paper we present our approach to data race detection for an abstract language AFPL, *Async Finish Parallel Language*. We first present our language AFPL and informally describe its semantics. To motivate the generalization of the traditional SP-bags algorithm to our setting, we illustrate where our language allows for more computations than those expressible with the spawn-sync constructs in the Cilk programming language.

2.1 Syntax

Fig. 1 shows the relevant part of the language syntax for AFPL, that is, the portion that deals with parallelism. The language allows nesting of **finish** and **async** statements. That is, any statement can appear inside these two constructs. However, the language restricts the kind of statements that can appear inside **atomic** sections: no synchronization statements inside the atomic sections are allowed. To reflect that, and to avoid notational clutter by listing the usual statements of an imperative programming language such as assignments, reads, procedure calls, loops and conditionals, we use the shortcut parametric macro ST (to stand for standard statements). $ST(s)$ will generate the set of usual statements and for any statement, it will replace its sub-statement, if necessary, with s. That is, one of the several statements in the set for $ST(s)$ will be the conditional **if**(b) s **else** s, while for $ST(r)$, it will be **if**(b) r **else** r.

[1] These benchmarks also conform with version 1.5 of the X10 language.

While languages such as *X10* and *HJ* allow for more expressive synchronization mechanisms such as conditional atomic sections, clocks or phasers, the core of these languages is based around the constructs shown in Fig. 1. We note that a similar language, called Featherweight X10 (FX10) has been recently considered in [11]. FX10 considers a more restricted calculus (e.g. it has one large one-dimensional array for the global store) and does not support atomic sections. Our data race detection algorithm is largely independent of the sequential constructs in the language. That is, the sequential portion of the language can be based on the sequential portions of C, C++ or Java.

2.2 Language Semantics

Next, we briefly discuss the relevant semantics of the concurrency constructs. For formal semantics of the async and finish constructs, see FX10 [11].

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 AFPL code shown in Fig. 2. Suppose the main task starts executing this piece of 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 asyncs in lines 16 and 17 to complete. The statement **atomic** $\{\ s\ \}$ means that that the statement s is executed atomically with respect to other atomic task. As mentioned earlier, an atomic section cannot contain any synchronization constructs.

$$
\begin{array}{lrcl}
Program: & P & ::= & \text{main } \{\ \textbf{finish } \{\ s\ \}\ \} \\
Statement: & s & ::= & \textbf{finish } \{\ s\ \} \\
 & & | & \textbf{async } \{\ s\ \} \\
 & & | & \textbf{atomic } \{\ r\ \} \\
 & & | & ST(s) \\
 & & | & s\ ;\ s \\
Restricted & r & ::= & ST(r) \\
Statement & & | & r\ ;\ r
\end{array}
$$

Fig. 1. The syntax of synchronization statements for AFPL.

2.3 Cilk vs. AFPL

Our data race detection algorithm, ESP-bags, presented in later sections, is an adaptation of the SP-bags algorithm [7] developed for the Cilk programming language. Unfortunately, their algorithm cannot be applied directly to our language and needs to be

```
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
```

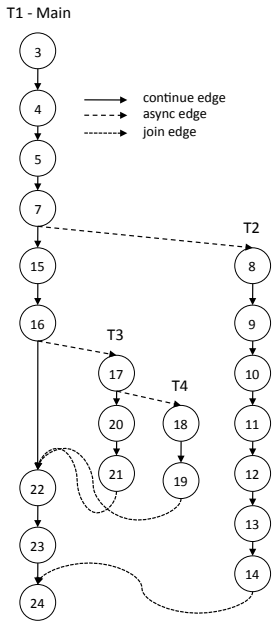

Fig. 2. An example AFPL program and its computation graph.

extended. The reason is that our language supports a more relaxed concurrency model than the spawn-sync Cilk computations. The key semantic relaxation lays 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 can 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 sync statement in Cilk can be directly translated to a standard **finish** block, where the start of the finish block is the start of the procedure and the end of the finish block is the label of the sync statement. For instance, we can trivially translate the following Cilk program:

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

into the following AFPL program:

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

That is, each spawn statement is replaced by an async statement and each sync statement is replaced with a finish block, where the scope of the finish ranges from the start of the task to the label of the corresponding sync.

In contrast, in AFPL, it is possible for a task to join to *some* and not all of its descendant tasks. The way these descendant tasks are specified at the language level is with the **finish** construct: upon encountering the end of a finish block, the task waits until all of the descendant tasks created inside the finish scope have completed.

The computation graph in Fig. 2 illustrates the differences between Cilk and AFPL. Each horizontal 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 how at a join point, the main task need not wait for all tasks which it created (via **async**), to complete. That is, at label 22, it only waits for T3 and T4 and not for T2. Such computations, where at a join point, a task need not wait for all of its descendants, are not allowed by the traditional spawn-sync semantics used in Cilk.

Further, 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 AFPL, 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. As we can see, this need not be the case in AFPL: 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* [3], 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 directly convert a AFPL program into the spawn-sync semantics of Cilk, which in turn implies that we cannot use its SP-bags algorithm immediately and we need to somehow generalize that algorithm to our setting. We show how that is accomplished in the next section.

3 Algorithm

In this section, we briefly survey the existing SP-bags algorithm used for spawn-sync computations. Then, we present our extension of that algorithm for detecting data races in AFPL programs.

The original SP-bags algorithm was presented for the spawn-sync computation of Cilk. 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. By having the original algorithm and our extension of it in the same language, we can clearly see exactly what the extension is.

3.1 SP-bags

The basic idea behind the SP-bags algorithm is to attach two bags, S and P, to each task. Each bag contains task id's. 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.

The SP-bags algorithm requires executing the program in a sequential depth-first manner. Note that a sequential depth-first execution of the program on a single processor satisfies the synchronization semantics of the program. Further, each memory location

is instrumented to contain two additional fields: a *reader* task id and a *writer* task id. As described below, these fields are updated during the depth-first execution of the program. During this depth-first execution, every time a shared memory location is accessed by a task, using the S and P bags, the algorithm checks whether that task can interfere with the task that is recorded in the reader and/or writer fields. Next, we show when and how the S and P bags are updated.

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 updated to contain itself and its P bag is reset. When a task A returns to a task B in the depth-first execution, then both of its bags, S and P, are moved to the P bag of its parent, B, and its bags are reset. When a join point is encountered in a task, then the P bag of that task is moved to its S bag. Note how in the last rule, the identifier of the finish, F, is actually not used in the update of the S and P bags.

The meaning of the S and P bags is as follows. 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. That is, given the S and P bags of all of the tasks, the contents of all these bags are actually disjoint. Therefore, all of these bags are maintained using a disjoint-set data structure. The disjoint-set data structure maintains the entire collection of task id's with support for operations like *MakeSet*, *Union*, and *Find-Set*. Tarjan [15] proved that any m of these operations on n bags take a total of $O(m\alpha(m, n))$ time.

In addition to the above steps, during the depth-first execution of a program, the SP-bags algorithm requires that action is taken on every read and write of a shared variable. Figure 3 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.

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

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

Fig. 3. Instrumentation on shared memory access. Applies both to SP-bags and ESP-bags

3.2 ESP-bags

Next, we present our extensions to the SP-bags algorithm. Recall that the key difference between AFPL and spawn-sync lays in the flexibility of selecting which of its descendent tasks a parent task can join to. In particular, with spawn-sync, once we translate the program to AFPL, we clearly see that the scope of any finish block begins at the start of a task, while in AFPL, a finish block can begin at any point in the task. Therefore, it is natural that these relaxations in the treatment of finish blocks will trigger extensions to the checking algorithm. Next, we present these extensions. 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 lays in attaching P bags, not only to tasks, but also to identifiers of finish blocks. At the start of a finish block F, the bag P_F is reset. Then, when a finish block ends in a task, the contents of its P bag are moved to the S bag of that task. Further, when during the depth-first execution a task returns to its parent, say B, B may be both a task *or* a finish scope. 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 algorithm can be applied directly to spawn-sync programs as well by first translating then to async-finish as shown earlier, and the 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 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 by F_1 and the first instance of the finish in line 15 by F_2. Also, we refer to the first instance of the tasks generated by the asyncs in lines 7, 16, and 17 by T_2, T_3, and T_4 respectively.

Table 1. ESP-bags Example

PC	T_1 S	F_1 P	T_2 S	F_2 P	T_3 P	T_3 S	T_4 S	B[0] 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

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.

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 (P bag of F_1). Hence this 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 Handling Atomic Blocks

In this section, we briefly describe an extension to the ESP-bags algorithm to accommodate handling of atomic sections. Atomic sections are useful since they allow the programmer to write deterministic parallel programs in which multiple tasks interact and update shared memory locations.

The extension to handle atomic sections includes checking that atomic and non-atomic access 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, *atomicReader*, and *atomicWriter*. These fields are used to hold the task that performs an *atomic* read or write on the location. We need to handle reads and writes from *atomic* blocks differently as compared to *non-atomic* operations. Fig. 4 shows the steps needed to be performed during each of the operations: *read*, *write*, *atomic-read*, and *atomic-write*. Note that we also need to modify the actions taken on a *read* and *write* to a memory location because they may result in a conflict with an *atomic* operation. As before, for each operation on

```
1  Atomic Read of location L by Task t:
2      If L.writer is in a P-bag then Data Race;
3      If L.atomicReader is in a S-bag then L.atomicReader = t;

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

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

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

Fig. 4. ESP-bags algorithm for AFPL, with support for *atomic* blocks

a shared memory location, we only need to check those fields of the location that could conflict with the current operation.

5 Evaluation

Table 2. List of Benchmarks Evaluated

Source	Benchmark	Description
JGF (Section 2)	Series	Fourier coefficient analysis
	LUFact	LU Factorisation
	SOR	Successive over-relaxation
	Crypt	IDEA encryption
	Sparse	Sparse Matrix multiplication
JGF (Section 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	sMatrix Multiplication (two 1000*1000 double matrix)

We report the performance results of our experiments on a 16-way (quad-socket, quad-core per socket) Intel Xeon 2.4GHz system with 30 GB memory, running Red Hat Linux (RHEL 5). The JVM used is the Sun Hotspot JDK 1.6. We applied the ESP-bags algorithm to a set of 8 JGF benchmarks shown in Table 2. We report the results for different input sizes of these benchmarks since our algorithm is guaranteed to be sound for a given input. We also evaluated our algorithm on 3 Shootout benchmarks and 1 EC2 challenge benchmark. All the benchmarks used were written in HJ using only the AFPL constructs.

Results of ESP-bags algorithm Table 3 shows the results of applying the ESP-bags algorithm on our benchmarks. This table gives the original time taken for each benchmark, i.e., the time taken to execute the 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 Section A. The outcome of the ESP-bags algorithm is also included in the table, which clearly shows there are no determinacy races in any of the benchmarks for any input size. Hence all the benchmarks are deterministic for the inputs considered. Note that though RayTracer has some *atomic* conflicts, it is deterministic since all the *atomic* blocks in it were tagged to indicate that they *commute*.

Table 3. Slowdown of ESP-bags Algorithm

Benchmark	Number of asyncs	Time (s)	ESP-bags Slowdown		Result
			w/o opts	w/ opts	
Crypt - C	1.3e7	15.24	7.63	7.29	No Data Races
LUFact - C	1.6e6	15.19	12.45	10.08	No Data Races
MolDyn - A	5.1e5	45.88	10.57	3.93	No Data Races
MonteCarlo - B	3.0e5	19.55	1.99	1.57	No Data Races
RayTracer - B	5.0e2	38.85	11.89	9.48	No Data Races (Atomic conflict)
Series - C	1.0e6	1395.81	1.01	1.00	No Data Races
SOR - C	2.0e5	3.03	14.99	9.05	No Data Races
Sparse - C	6.4e1	13.59	12.79	2.73	No Data Races
Fannkuch	1.0e6	7.71	1.49	1.38	No Data Races
Fasta	4.0e0	1.39	3.88	3.73	No Data Races
Mandelbrot	1.6e1	11.89	1.02	1.02	No Data Races
Matmul	1.0e3	19.59	6.43	1.16	No Data Races
Geo Mean			4.86	3.05	

ESP-bags slowdown On an average, the slowdown of the benchmarks with the ESP-bags algorithm is $8.44\times$ without optimization. When all the static optimizations are applied, the average slowdown drops to $5.29\times$. The slowdown of all the benchmarks except LUFact is less than $10\times$. In fact it is the smaller sizes A and B of LUFact that have higher slowdown. Since the actual execution times in these cases are very small, 0.26s and 1.76s, the overheads are exaggerated.

The slowdown for benchmarks like MolDyn, MonteCarlo and Sparse are less than $5\times$. 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 and hence we do not instrument any access to these variables. On the other hand, the slowdown for SOR and RayTracer benchmarks are around $9\times$.

Performance of Optimizations We now discuss the effects of the optimizations cumulatively on the benchmarks. Note that these optimizations only eliminate instrumentations corresponding to the read and write of share memory locations. Hence they do have any effect on the size of the memory used by the benchmarks. As is evident from the table,

some of the benchmarks like SOR, Sparse, MolDyn, and Matmul benefit a lot 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 less. The optimizations does not reduce the slowdown much for Crypt and LUFact because in these benchmarks very few instrumentations are eliminated as a result of the optimizations. In the case of MonteCarlo and RayTracer, though a good number of instrumentations are eliminated, a significant fraction of them still remain and hence there is not much performance improvement in these benchmarks due to optimizations. On an average, there is a 37% reduction in the slowdown of the benchmarks due these optimizations.

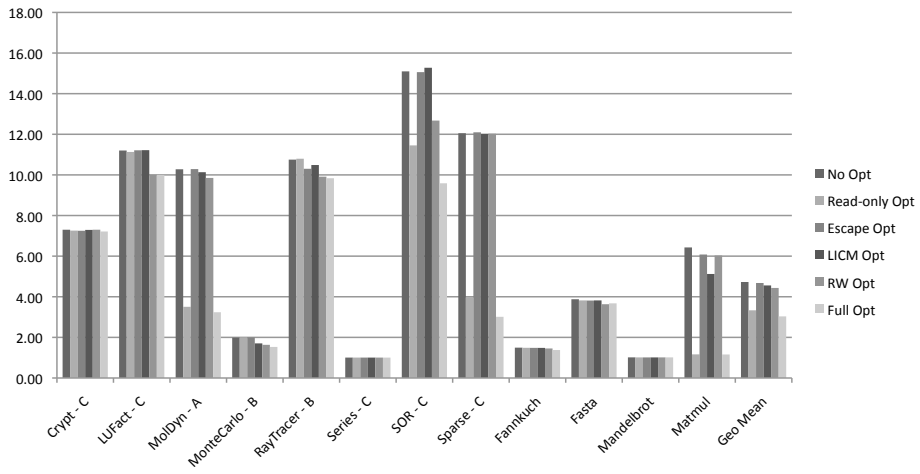

Fig. 5. Breakdown of static optimizations

Breakdown of the Optimizations We now describe the effects of each of the static optimizations separately on the performance of the benchmarks. Figure 5 shows the break up of the effects of each of the static optimizations, that includes read-only check elimination, escape analysis to remove redundant instrumentations, loop invariant code motion, and read-write check elimination. The graph also shows the slowdown without any optimization and with the whole set of optimizations enabled. Note that the optimization described in Section A.1, that identifies instructions within parallel constructs so that only those instructions need to be instrumented, is applied to all the versions included here, including the unoptimized version. This is because we consider that optimization as a basic step without which there could be unnecessary instrumentations.

The read-only check elimination performs much better than the other optimizations for most of the benchmarks, like MolDyn, SOR, and SparseMatmult. This is because in these benchmarks the parallel regions include reads to many arrays which are written only in the sequential regions of the code. Hence, this optimization eliminates the instrumentation for all these reads. It contributes the most to the overall performance

improvement in the full optimized version. The read-write optimization works well in the case of SOR, but does not have much effect on other benchmarks. The Loop invariant code motion helps improve the performance of Montecarlo the most and the Escape analysis does not seem to help any of these benchmarks to a great extent.

Note that the performance of these four static optimizations do not directly add up to the performance of the fully optimized code. This is because some of these optimizations creates more chances for other optimizations. Hence their combined effect is much more than their sum. For example, the loop invariant code motion creates more chances for the Read-only and Read-Write optimization. So, when these two optimizations are performed after loop invariant code 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 could be different. But we believe that these static optimizations, when applied in combination, are in general good enough to improve the performance of most of the benchmarks.

6 Related Work

The original Cilk paper [7] introduces SP-bags for spawn-sync computations. We extend that algorithm to the more general setting of async-finish computations. We also handle atomic sections and detect data races between atomic and non-atomic accesses. Moreover, we outline and implement a range of static optimizations to reduce instrumentation costs.

A recent work on detecting data races by Flanagan et al. [8] (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 hence 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 one such instance would make it infeasible for programs with millions of parallel tasks. On the other hand, our approach requires only a constant space for every memory location and a time proportional to the inverse ackermann function. Also, FastTrack just checks for data races in a particular execution of a program, whereas our approach can guarantee the non-existence of data races for all possible schedules of a given input. The price we have to pay for this soundness guarantee is that we have to execute the given program sequentially. But given that this needs to be done only during the development stage we feel our approach is of value.

Sadowski et al. [14] proposes 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. Again the downside of our approach is that we have to run the program sequentially whereas they can run the program in parallel.

The static optimizations that we use to eliminate the redundant instrumentations and hence reduce the overhead is similar to the compile-time analyses proposed by John Mellor-Crummey [12]. 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. This 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 which are guaranteed to have already been instrumented in that task.

7 Conclusion

In this paper we proposed a dynamic analysis algorithm, ESP-bags, for efficiently checking determinism of structured async-finish parallel programs. The algorithm is sound for a given input: if a deterministic violation exists, it will be reported. Our algorithm generalizes the SP-bags algorithm developed for the more restricted spawn-sync Cilk model.

We have implemented our algorithm in a tool called TASKCHECKER. Further, TASKCHECKER is augmented with static compiler optimizations that reduce the incurred overhead by $1.51\times$ on average. Evaluation of TASKCHECKER on a suite of 12 benchmarks shows that the dynamic analysis introduces an average slowdown of $5.58\times$ without compiler optimizations, and $3.68\times$ with compiler optimizations, making the tool suitable for practical use.

8 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 this work on a short notice.

References

1. AGARWAL, S., BARIK, R., BONACHEA, D., SARKAR, V., SHYAMASUNDAR, R. K., AND YELICK, K. Deadlock-free scheduling of X10 computations with bounded resources. In *SPAA '07: Proceedings of the 19th symposium on Parallel algorithms and architectures* (2007), ACM, pp. 229–240.
2. BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: an efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and P ractice of Parallel Programming, PPoPP* (Oct. 1995), pp. 207–216.
3. BLUMOFE, R. D., AND LEISERSON, C. E. Scheduling multithreaded computations by work stealing. *J. ACM 46*, 5 (1999), 720–748.

4. BOCCHINO, R., ADVE, V., ADVE, S., AND SNIR, M. Parallel programming must be deterministic by default. In *First USENIX Workship on Hot Topics in Parallelism (HOTPAR 2009)* (2009).
5. CHARLES, P., GROTHOFF, C., SARASWAT, V. A., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the Twentieth Annual ACM SIGPLAN Conference on Obj ect-Oriented Programming, Systems, Languages, and Applications, OOPSLA* (Oct. 2005), pp. 519–538.
6. DIJKSTRA, E. W. Letters to the editor: go to statement considered harmful. *Commun. ACM 11*, 3 (1968), 147–148.
7. FENG, M., AND LEISERSON, C. E. Efficient detection of determinacy races in cilk programs. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures* (1997), ACM, pp. 1–11.
8. FLANAGAN, C., AND FREUND, S. N. Fasttrack: efficient and precise dynamic race detection. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (2009), ACM, pp. 121–133.
9. LEA, D. A java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande* (2000), ACM, pp. 36–43.
10. LEE, E. A. The problem with threads. *Computer 39*, 5 (2006), 33–42.
11. LEE, J. K., AND PALSBERG, J. Featherweight x10: a core calculus for async-finish parallelism. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel computing* (2010), ACM, pp. 25–36.
12. MELLOR-CRUMMEY, J. 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, pp. 129–139.
13. Habanero Multicore Software Research project. http://habanero.rice.edu/hj.
14. SADOWSKI, C., FREUND, S. N., AND FLANAGAN, C. SingleTrack: A dynamic determinism checker for multithreaded programs. In *Programming Languages and Systems* (2009), vol. 5502 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 394–409.
15. TARJAN, R. E. Efficiency of a good but not linear set union algorithm. *J. ACM 22*, 2 (1975), 215–225.

A 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. 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. A naive way to perform this is to instrument every access to every shared memory location. But some of these instrumentations may be redundant, i.e., all determinacy races that can be detected when these are present can also be detected after they have been removed. This is because some read and write operations are guaranteed to never affect the deterministic behavior of the program and hence such operations need not be instrumented.

As mentioned earlier, the ESP-bags algorithm also keeps track of the *finish*, *async*, *atomic* blocks in the program. Hence, it requires instrumentations for the start and end

of every such block in the program. But these instrumentations are all necessary to maintain the parallelism structure at runtime in the ESP-bags algorithm.

In this section, we describe the static analyses that we 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 is used to eliminate instrumentations. Fig. 6 shows a part of the example program in Fig. 2 (lines 15 - 22 deleted) written in AFPL with all its read and write operations instrumented (*DJCRead* and *DJCWrite* are library calls). 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.

A.1 Main Task Check Elimination in 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 affect the deterministic behavior of the program. Hence, there is no need to instrument the operations in such sequential code regions. In an AFPL program, the sequential code regions are the regions of the program that are executed by the *main task*. Thus, in an AFPL program, there is no need to instrument the read and write operations in the main task. Fig. 7 shows the result of eliminating the instrumentations in the main task of the program in Fig. 6. The program in Fig. 6 contains a write to a heap location *p.x* in line 4 which will be executed by the main task. Hence the corresponding call to the library in line 3 can be eliminated.

A.2 Read-only Check Elimination in Parallel Code Regions

The input program may have shared memory locations that are written by the sequential regions of the program and only read within parallel regions of the program. Such read operations within parallel regions of the program need not be instrumented because parallel tasks reading from the same memory location will never lead to a conflict. To perform this optimization, the compiler implements an inter-procedural side-effect analysis to detect potential write operations to shared memory locations within the parallel regions of the given program. If there is no possible write to a shared memory location M in the parallel regions of the program, that clearly shows that all accesses to M in the parallel regions must be read-only and hence the instrumentations corresponding to these reads can be eliminated. (The checks for the writes in the sequential regions, if any, will be eliminated by the rule in Section A.1).

The result of applying this optimization on the program in Fig. 7 is shown in Fig. 8. There is no write to array *A* within the parallel regions of the program in figure Fig. 7. Hence the instrumentation in line 8 corresponding to the read of *A* in line 11 can be removed.

A.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 is a write. Suppose an object is created inside a task and it never escapes that task, then no other task can access this object and hence it cannot lead to a determinacy race. To ensure the task-local attribute, the compiler performs an inter-procedural analysis that identifies if an object is shared among tasks. This also requires an alias analysis to ensure that no alias of the object escapes the task. Thus, if an object O is proven to not escape a task, then the instrumentations corresponding to all accesses to O can be eliminated.

The object q in the program in Fig. 8 is created in line 11 within a task and it never escapes this task. Thus no access to q can lead to a determinacy race. Hence, the instrumentations in line 14 and 16 corresponding to access to q are eliminated and the resulting program is shown in Fig. 9.

A.4 Loop Invariant Check Motion

Recall that the instrumentation corresponding to a memory access to M will first check if the task that previously accessed M conflicts with the current task and also update the information that the current task now accessed M. If there are multiple accesses of the same type (read or write) to M by a task, then it is sufficient to instrument one such access because other instrumentations will only add to the overhead by unnecessarily repeating the steps. Suppose the input program accesses a shared memory location M unconditionally inside a loop, the instrumentation corresponding to this access to M can be moved outside the loop 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. 9, 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. 10. Note the test for the non-zero trip count in line 12 guarding this instrumentation outside the loop.

A.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. An intuitive argument for this is that, 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). This rule tends to be applicable in fewer situations than the previous rule in practice, because computation of postdominance includes the possibility of exceptional control flow.

Consider the program in Fig. 10. There is an instrumentation for the write to *p.x* in line 9 and an instrumentation corresponding to the read of the same memory location in line 13. Since the instrumentation in line 9 dominates the one in line 13 and latter is not a write, the latter can be eliminated.

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

Fig. 6. An example AFPL program with all read and write operations instrumented. Time = 17.13 s (Uninstrumented Time = 0.56 s)

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

Fig. 8. After applying the read-only check optimization on the program in Fig. 7. Time = 13.35 s

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

Fig. 10. After applying the loop invariant check elimination optimization on the program in Fig. 9. Time = 1.24 s

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

Fig. 7. After applying the main task check elimination optimization on the program in Fig. 6. Time = 17.12 s

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

Fig. 9. After applying the escape analysis and check elimination optimization on the program in Fig. 8. Time = 6.78 s

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

Fig. 11. After applying the read/write check elimination optimization on the program in Fig. 10. Time = 0.83 s