Efficient Data Race Detection for Async-Finish Parallelism

Raghavan Raman[1], Jisheng Zhao[1], Vivek Sarkar[1], Martin Vechev[2], and Eran Yahav[2]

[1] Rice University
[2] IBM T. J. Watson Research Center
{raghav,jisheng.zhao,vsarkar}@rice.edu,
{mtvechev,eyahav}@us.ibm.com

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 larger) for use in mainstream software development.

In this paper, we present an efficient dynamic race detector algorithm targeting the async-finish task-parallel parallel programming model. The async and finish constructs are at the core of languages such as X10 and Habanero Java (HJ). These constructs generalize the spawn-sync constructs used in Cilk, while still ensuring that all computation graphs are deadlock-free.

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.

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 inter-leavings of operations in their program. The vast number of inter-leavings 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 make it easy to get things wrong.

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* [11] and *monitors*. Recently, additional support for fork-join task parallelism has been added in the form

G. Roşu et al. (Eds.): RV 2010, LNCS 6418, pp. 368–383, 2010.

of libraries [15,18] to existing programming environments and languages such as Java and .NET.

Parallel languages such as Cilk [5], 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 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 that the computation graphs generated in the language are deadlock-free [17] (unlike unrestricted fork-join computations).

Data Race and Determinism Detection. A central property affecting the correctness of parallel algorithms is data-race freedom. Data-race freedom is a desirable property as in some cases it can imply determinism [16,7]. For instance, in the absence of data races, all parallel programs with *async* and *finish*, but without *isolated* constructs, are guaranteed to be *deterministic*. Therefore, if we can prove data-race freedom of programs which do not contain *isolated* constructs, then we can conclude that the program is deterministic.

We present an efficient dynamic analysis algorithm that checks the presence of data races in async-finish style parallel computations. These constructs form the core of the larger X10, HJ and Cilk 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 [12] 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. Both, the SP-bags algorithm, as well as our extension to it, are sound for a given input: if a data race exists for that input, a violation will be reported.

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 for structured async-finish parallel programs. Our algorithm generalizes the classic SP-bags algorithm designed for the more restricted spawn-sync Cilk model.
- 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 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 perform data race detection 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.

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

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 broader sets of computation dags than those expressible with the spawn-sync constructs in the Cilk programming language.

2.1 Syntax

Fig. 1 shows the part of the language syntax for AFPL that is relevant to 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 **isolated** sections: no synchronization constructs such as async and finish are allowed inside isolated sections. However, isolated blocks may contain loops, conditionals, and other forms of sequential control flow.

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

Fig. 1. The syntax of synchronization statements for AFPL

To reflect that, we use the shortcut parametric macros ST and RT (to stand for standard statements and restricted statements respectively). $ST(s)$ will generate the set of usual statements and for any statement, it will replace its sub-statement, if necessary, with s. For instance, 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. The set of statements generated by RT includes all statements of ST except procedure calls. This restriction is placed to avoid synchronization constructs in methods called from within isolated sections.

While languages such as *X10* and *HJ* also allow for more expressive synchronization mechanisms such as futures, conditional isolated 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 [17]. FX10 considers a more restricted calculus (e.g. it has one large one-dimensional array for the global store) and does not support isolated sections. 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.

```
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
```

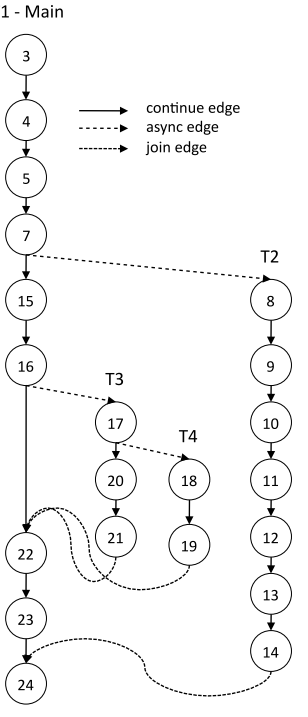

Fig. 2. An example AFPL program and its computation graph

2.2 Informal Language Semantics

Next, we briefly discuss the relevant semantics of the concurrency constructs. For formal semantics of the async and finish constructs, see FX10 [17]. 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 **isolated** { s } means that the statement s is executed atomically with respect to other isolated statements[2].

[2] As advocated in [14], we use the *isolated* keyword instead of *atomic* to make explicit the fact that the construct supports weak isolation rather than strong atomicity.

2.3 Cilk vs. AFPL

Our data race detection algorithm, ESP-bags, presented in later sections, is an adaptation of the SP-bags algorithm [12] developed for the Cilk programming language. Unfortunately, the SP-bags algorithm cannot be applied directly to our language and needs to be 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 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 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 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, with the use of nested finish operations in AFPL, it is possible for a task to join with *some* rather than 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 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 and not for T2, which is not possible using the 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* [6], while the computations generated by our language are called *terminally-strict* [2]. 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 ESP-Bags Algorithm

In this section, we briefly summarize 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 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

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. Each bag contains a set of task id's. 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 these 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 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 which is in a P-bag when W_1 occurs and it does not race with 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.

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. Each memory location is instrumented to contain two additional fields: a *reader* task id and a *writer* task id. 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, and its P bag is initialized to the empty set. 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, the P bag of that task is moved to its S bag.

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. 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 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 ESP-bags 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	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

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 Isolated Blocks

In this section, we briefly describe an extension to the ESP-bags algorithm to accommodate handling of 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.

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

1  Isolated 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.isolatedWriter is in a S–bag then L.isolatedWriter = t;

1  Read location L by Task t:
2      If L.writer is in a P–bag or L.isolatedWriter is in a P–bag
3          then Data Race;
4      If L.reader is in a S–bag then L.reader = 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.isolatedWriter is in a P–bag or L.isolatedReader is in a P–bag
4          then Data Race;
5      L.writer = t;
```

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

The extension to handle 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 hold the task that performs an *isolated* read or write on the location. We need to handle reads and writes from *isolated* blocks differently as compared to *non-isolated* operations. Fig. 4 shows the steps needed to be performed during each of the operations: *read*, *write*, *isolated-read*, and *isolated-write*.

5 Compiler 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. To test a given program for data-race freedom using the ESP-bags algorithm, we need a compiler transformation pass that instruments read and write operations on shared memory locations in the program with appropriate calls to the library. 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.

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. There is no need to instrument the operations in such sequential code regions. In an AFPL program, the sequential code regions are executed by the *main task*. Thus, in an AFPL program, there is no need to instrument the read and write operations in the sequential code regions of the main task.

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 [4] 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.

Escape Analysis. The input program may include many parallel tasks. A 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 data race. To ensure the task-local attribute, the compiler performs an inter-procedural escape analysis [10] 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.

Loop Invariant Check Optimization. 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.

Read/Write Check Elimination. 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.

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 post-dominates 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 post-dominance includes the possibility of exceptional control flow.

6 Evaluation

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 Java Grande Forum (JGF) benchmarks shown 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 get the results of size B for MolDyn since the both the versions (original and instrumented) runs out of memory. 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 and are available from [1].

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)

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 5. The outcome of the ESP-bags algorithm is also included in the table, which clearly shows there are no data races in any of the benchmarks. The same was observed for all the 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 3. Slowdown of ESP-bags Algorithm

Benchmark	Number of asyncs	Time (s)	ESP-bags Slowdown Factor w/o opts	ESP-bags Slowdown Factor w/ opts	Result
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 (Isolated 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 $4.86\times$ without optimization. When all the static optimizations are applied, the average slowdown drops to $3.05\times$. The slowdown of all the benchmarks except LUFact is less than $10\times$. 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 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. 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.

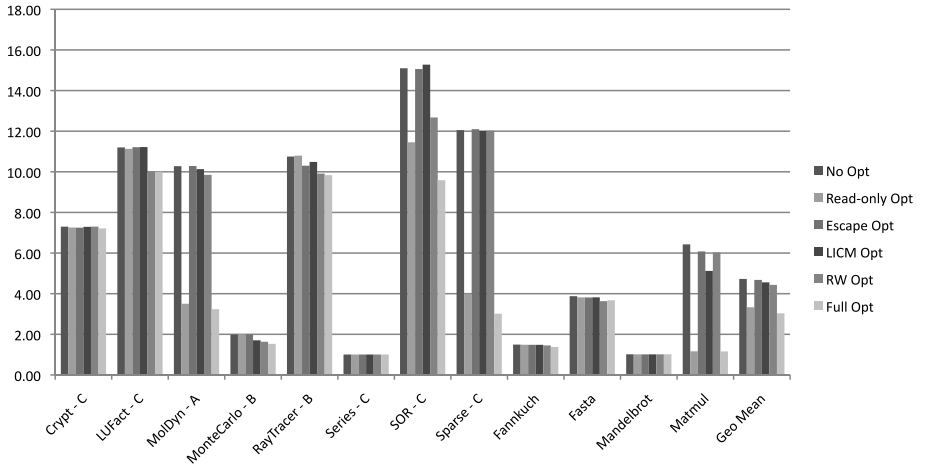

Fig. 5. Breakdown of static optimizations

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 breakdown of the effects of each of the static optimizations. The graph also shows the slowdown without any optimization and with the whole set of optimizations enabled. The Main Task Check Elimination optimization described in Section 5 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 excessive 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.

7 Related Work

The original Cilk paper [12] introduces SP-bags for spawn-sync computations. We extend that algorithm to the more general setting of async-finish computations. 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 $33\times$ - $78\times$ range for programs that follow their locking discipline, and upto $3700\times$ 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 just $3.05\times$ on average.

A recent result on detecting data races by Flanagan et al. [13] (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 Ackerman 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. [20] 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 hence reduce the overhead is similar to the compile-time analyses proposed by Mellor-Crummey [19]. 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 which are guaranteed to have already been instrumented in that task.

8 Conclusion

In this paper, we proposed a sound and efficient dynamic data-race detection algorithm called ESP-bags. ESP-bags targets the async-finish parallel programming model, which generalizes the spawn-sync model used in Cilk.

We have implemented ESP-bags in a tool called TASKCHECKER and augmented it with a set of static compiler optimizations that reduce the incurred overhead by $1.59\times$ on average. Evaluation of TASKCHECKER on a suite of 12 benchmarks shows that the dynamic analysis introduces an average slowdown of $4.86\times$ without compiler optimizations, and $3.05\times$ with compiler optimizations, making the tool suitable for practical use.

In future work, we plan to investigate the applicability of ESP-bags to the fork-join concurrency model.

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.

References

1. Habanero Java, `http://habanero.rice.edu/hj`
2. Agarwal, S., Barik, R., Bonachea, D., Sarkar, V., Shyamasundar, R.K., Yelick, K.: Deadlock-free scheduling of X10 computations with bounded resources. In: SPAA 2007: Proceedings of the 19th symposium on Parallel algorithms and architectures, pp. 229–240. ACM, New York (2007)
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.: The habanero multicore software research project. In: OOPSLA 2009: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, pp. 735–736. ACM, New York (2009)
4. Barik, R., Sarkar, V.: Interprocedural Load Elimination for Dynamic Optimization of Parallel Programs. In: PACT 2009, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, Washington, DC, USA, pp. 41–52. IEEE Computer Society, Los Alamitos (September 2009), `http://dx.doi.org/10.1109/PACT.2009.32`
5. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. In: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP, pp. 207–216 (October 1995)
6. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM 46(5), 720–748 (1999)
7. Bocchino, R., Adve, V., Adve, S., Snir, M.: Parallel programming must be deterministic by default. In: First USENIX Workship on Hot Topics in Parallelism, HOTPAR 2009 (2009)
8. Charles, P., Grothoff, C., Saraswat, V.A., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., 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, pp. 519–538 (October 2005)
9. Cheng, G.-I., Feng, M., Leiserson, C.E., Randall, K.H., Stark, A.F.: Detecting data races in cilk programs that use locks. In: Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 1998), Puerto Vallarta, Mexico, June 28–July 2, pp. 298–309 (1998)

10. Choi, J.-D., Gupta, M., Serrano, M.J., Sreedhar, V.C., Midkiff, S.P.: Stack allocation and synchronization optimizations for Java using escape analysis. ACM Trans. Program. Lang. Syst. 25(6), 876–910 (2003), http://doi.acm.org/10.1145/945885.945892
11. Dijkstra, E.W.: Cooperating sequential processes, 65–138
12. Feng, M., Leiserson, C.E.: Efficient detection of determinacy races in cilk programs. In: SPAA 1997: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures, pp. 1–11. ACM, New York (1997)
13. Flanagan, C., Freund, S.N.: Fasttrack: efficient and precise dynamic race detection. In: PLDI 2009: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, pp. 121–133. ACM, New York (2009)
14. Larus, J.R., Rajwar, R.: Transactional Memory. Morgan and Claypool (2006)
15. Lea, D.: A java fork/join framework. In: JAVA 2000: Proceedings of the ACM, conference on Java Grande, pp. 36–43. ACM, New York (2000)
16. Lee, E.A.: The problem with threads. Computer 39(5), 33–42 (2006)
17. Lee, J.K., Palsberg, J.: Featherweight x10: a core calculus for async-finish parallelism. In: PPoPP 2010: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel computing, pp. 25–36. ACM, New York (2010)
18. Leijen, D., Schulte, W., Burckhardt, S.: The design of a task parallel library. In: OOPSLA 2009: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications, pp. 227–242. ACM, New York (2009)
19. Mellor-Crummey, J.: Compile-time support for efficient data race detection in shared-memory parallel programs. In: PADD 1993: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging, pp. 129–139. ACM, New York (1993)
20. Sadowski, C., Freund, S.N., Flanagan, C.: SingleTrack: A dynamic determinism checker for multithreaded programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 394–409. Springer, Heidelberg (2009)