Automatically Inserting Synchronization Statements in Divide-and-Conquer Programs

Pieter Hijma, Rob V. van Nieuwpoort, Ceriel J.H. Jacobs and Henri E. Bal
Vrije Universiteit Amsterdam
The Netherlands
Email: {pieter,rob,ceriel,bal}@cs.vu.nl

***Abstract*—Divide-and-conquer is a well-known and important programming model that supports efficient execution of parallel applications on multi-cores, clusters and grids. In a divide-and-conquer system such as Satin or Cilk, recursive calls are automatically transformed into jobs that execute asynchronously. Since the calls are non-blocking, consecutive calls are the source of parallelism. However, the programmer has to manually enforce synchronization with `sync` statements that indicate where the system has to wait for the result of the asynchronous jobs.**

In this paper, we investigate the possibility to automatically insert `sync` statements to relieve the programmer of the burden of thinking about synchronization. We investigate whether correctness can be guaranteed and to what extent the amount of parallelism is reduced. We discuss the code analysis algorithms that are needed in detail.

To evaluate our approach, we have extended the Satin divide-and-conquer system, which targets efficient execution on grids, with a sync generator. The fact that Satin uses Java as a base language helps the sync generator to reason about control flow and aliasing of references to objects. Our experiments show that, with our analysis, we can automatically generate synchronization points in virtually all real-life cases: in 31 out of 35 real-world applications the `sync` statements are placed optimally. The automatic placement is correct in all cases, and in one case the sync generator corrected synchronization errors in an application (FFT).

I. Introduction

Writing parallel programs is difficult in general. Writing parallel programs that execute efficiently on multiple clusters or clouds is even more demanding. Satin [1] makes cross-grid computing accessible to programmers who are not parallel programming experts. It allows programmers to write parallel programs without much effort by offering a sequential divide-and-conquer programming model.

In Satin, programmers annotate recursive methods to indicate that calls to these methods are spawnable, which means that they can be executed asynchronously. Consecutive spawnable method calls create parallelism in the program. These method calls are transformed into jobs that are executed efficiently on grids or clouds using the Ibis platform [2].

However, since spawnable method calls are non-blocking, programmers also have to annotate where in the program the system has to block until the results of the jobs are available. Programmers indicate this by carefully inserting `sync()` statements. Placing `sync()` statements too soon results in less parallelism than possible and placing them too late gives incorrect results. Our goal is to make grid computing even more accessible to programmers who are not parallel programming experts by making sync insertion automatic.

This would mean that programmers no longer have to think about synchronization, but that a sync generator would solve this. The following questions arise: whether syncs can always be automatically inserted in such a way that the resulting program is correct; how much parallelism can be obtained; whether this can be done without complicated analysis.

We found that the implementation of the sync generator program is able to insert sync statements in such a way that the resulting program is always correct, but alias and control flow analysis are needed to do this. More extensive analysis will not lead to optimal placement in all cases: programmers sometimes *deliberately* use unsynchronized variables for performance reasons. An automatic generator cannot determine this.

In practice, our sync generator achieves excellent results. We tested the sync generator on 35 pre-existing real-world Satin applications. In 31 of them, the sync generator found the optimal places for the sync statements. In all but one of the remaining cases, the sync generator gave a warning that the placement was likely suboptimal. In one case, it even corrected an originally incorrect application (FFT).

Our contributions are three-fold:

- We make implementing parallel divide-and-conquer applications even more effortless than before. Programmers only have to indicate parallel methods, but not the synchronization points.
- We offer a good understanding of the problems involving automatically inserting synchronization statements.
- We provide a real implementation in the form of a compilation pass for the Satin compiler that generates synchronization statements automatically.

The following section discusses the Satin programming model and explains some basic concepts. Section III defines the problem in detail and section IV discusses the implementation of the sync generator. We evaluate the sync generator in section V using 35 real-world Satin applications and discuss the results in section VI. Section VII describes related work after which the paper concludes.

II. The Satin programming model

This section briefly introduces the Satin programming model. Satin [1] is a divide-and conquer-framework similar to

Cilk [3]. The main differences are that programs are written in Java instead of Cilk, a C derivative, and that programs are deployed on a grid and not on shared memory systems.

The Satin compiler will rewrite Satin programs in such a way that they can run in parallel on grids and clouds using Ibis [2]. Ibis is an environment that provides communication primitives to compute nodes in a grid. Rewriting Satin programs is performed with help of bytecode rewriting library BCEL [4].

In order to create parallelism in a Satin program, programmers have to make some annotations. They have to indicate which methods are spawnable to ensure that the Satin compiler will rewrite those methods to versions that spawn jobs on the grid. The second annotation the programmer has to make is a special `sync()` statement which is a barrier local to the method. The system will block at the sync statement until all results of the spawnable calls in the current method become available. A node that blocks in a sync may execute jobs resulting from spawnable calls in the meantime.

The Java type and class systems provide all means to annotate Satin programs. Programmers create spawnable classes by extending the class `ibis.satin.SatinObject`, that provides methods such as `sync()`. Programmers can also create an interface that extends `ibis.satin.Spawnable`. The method calls of the methods that are declared in this interface will be rewritten by the Satin compiler and executed in parallel.

The procedure from writing to deploying a Satin program is as follows: programmers write a sequential recursive program in Java and annotate the (possibly recursive) calls that have to be spawnable. They also place sync statements in places where the program has to wait for the results of the spawnable calls. The program is compiled to normal, sequential Java bytecode. This serves as input for the Satin compiler that rewrites the bytecode in such a way that Satin jobs are spawned. The application is now ready to be deployed on the grid.

The procedure in combination with the sync generator differs slightly and is depicted in Fig. 1. Programmers again write a sequential recursive program but leave out the sync statements. They compile the program to ordinary bytecode. The Satin compiler takes the Java bytecode as input, and generates sync statements as an extra compiler pass. Then it rewrites the bytecode to make it grid aware.

Throughout the rest of the document the following terms will be used: A *spawning class* is a class that contains spawning methods. A *spawning method* is a method that contains *spawnable calls*. A *spawnable call* is an invocation of a method with a *spawnable method signature*. This is the signature of the method annotated by the programmer to be spawnable. A *sync statement* is a local barrier synchronization primitive which makes sure that all spawnable calls have returned their values.

Fig. 2 depicts a typical Satin program. On line 3 `spawningMethod()` is marked to be a spawnable method as it is defined in an interface that extends `ibis.satin.Spawnable`. The method on line 6 is a

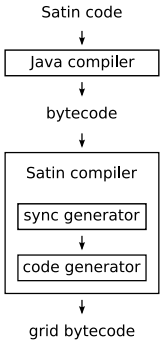

Fig. 1. Compiler procedure with automatic sync generation.

```
1 interface SpawnProg extends
2     ibis.satin.Spawnable {
3   int spawningMethod();
4 }
5
6 int spawningMethod() {
7   if (stopCondition) return 0;
8
9   int result1 = spawningMethod();
10  int result2 = spawningMethod();
11
12  sync();
13  return result1 + result2;
14 }
```

Fig. 2. A basic Satin program.

spawning method, because it contains two *spawnable calls* on line 9 and 10 (two recursive calls). These calls are non-blocking and as a result the two calls run in parallel. The system blocks on the `sync()` statement on line 12 until the parallel calls have finished and returned their results into `result1` and `result2`. The method can now safely return the sum of `result1` and `result2`.

Besides returning values using the `return` statement, a spawning method can also return using exceptions. Fig. 3 illustrates this. The non-blocking spawning method is called on line 6 in the `try` block and the program continues immediately beyond the `catch` clause. It executes the second `try` block, calling the second spawnable call on line 13 in parallel. The program continues beyond the `catch` clause and blocks on the `sync()`. When the spawnable calls have finished, the results are thrown and stored into local variables inside the `catch` block. Control flow returns from the `catch` block for both the first and second call. Now that the spawnable calls have finished and the results are stored in the local variables, the main process continues and throws the final result. Exceptions are used to asynchronously return results, for instance with speculative parallelism.

III. Problem description

The implementation of the sync generator focuses first of all on correctness. The main question is: Is it possible to create a sync generator compilation pass that inserts sync statements

```
void spawningMethod() throws Result {
  if (stopCondition) throw new Result();

  int result1, result2;
  try {
    spawningMethod();
  }
  catch (Result r) {
    result1 = r.result;
    return;
  }
  try {
    spawningMethod();
  }
  catch (Result r) {
    result2 = r.result;
    return;
  }

  sync();

  Result finalResult = new Result();
  finalResult.result = result1 + result2;
  throw finalResult;
}
```

Fig. 3. A basic Satin program throwing exceptions.

in such a way that the resulting program is guaranteed to be correct? Correctness in this sense means that the placement of sync statements is such that the parallel version delivers the same results as the sequential version.

The second issue is then: Is it possible to find a place for the sync statements in such a way that parallelism is created? If this is true, then the question is how optimal the placement is.

Other questions are whether this can be achieved in a way that does not require complicated analysis or places many spurious sync statements.

The sync generator operates under the following assumptions. It requires the interfaces that extend `ibis.satin.Spawnable` to find spawnable signatures. With this information it determines which classes are spawning classes. We assume that within a single run of the sync generator, the sync generator is given both the classes that need to be rewritten and the classes that implement `ibis.satin.Spawnable` to indicate which methods are spawnable.

In addition, there are some assumptions about spawnable calls. Figures 2, 3 and 4 show parts of typical Satin programs using return values, exceptions and loops respectively. A spawnable call can return in three ways. The spawnable method signature can be of type `void`. It then returns with a `return` statement. The second way is by returning a value using the `return` keyword. This happens in Fig. 2 and Fig. 4. Finally, it can return using exceptions as in Fig. 3. Return values from spawnable calls are likely to be stored in local variables to allow parallelism, but this may not be the case. Also, these local variables may not be loaded again. In this case and if the spawnable call is of type `void`, the sync

```
int spawningMethod() {
  if (stopCondition) return 0;

  int[] results = new int[NR_SPAWNS];
  int finalResult;

  for (int i = 0; i < results.length; i++) {
    results[i] = spawningMethod();
  }

  sync();
  for (int i = 0; i < results.length; i++) {
    finalResult += results[i];
  }

  return finalResult;
}
```

Fig. 4. A basic Satin program using loops.

generator needs to place a sync as last instruction of the spawning method.

To conclude the boundaries for the sync generator, it is not an issue when multiple syncs are placed behind each other. The system will notice during the sync that there are no spawnable calls running and will just continue without any problem. It is also no problem to have sync statements when no spawnable call will be called. The overhead of these calls is negligible.

IV. Implementation

The first subsection describes the basic algorithm of the sync generator compilation pass. The second subsection discusses several analysis strategies and gives insight into which problems need to be solved to automatically generate sync statements.

A. Basic algorithm

The basic algorithm of the sync generator is composed of three phases, the recording phase, the analysis phase and the generator phase. During the recording phase, the sync generator reads in all class files. Next, it finds all spawnable method signatures from the interfaces. It then tries to create spawning classes for every class file. This succeeds if the class contains spawning methods.

For a spawning method, the recording phase records all spawnable calls, and for every spawnable call, it keeps track of:

- the invoke instruction of the spawnable call
- the load instruction of the object reference on which the spawnable call is invoked
- the indices of the local variables in which the results are stored

A spawnable call tracks multiple local variables in case an exception is thrown. The `catch` block may store results in multiple local variables and subsequent loads of these local variables need a sync.

The analysis phase (discussed in detail in the following subsection) takes as input a spawning method and proposes

```
1 result1 = spawnableMethod();
2 result2 = spawnableMethod();
3
4 if (someCondition) {
5   sync();
6   return result1;
7 } else {
8   return result2;
9 }
```

Fig. 5. Jumping over sync().

one or more places to insert a sync statement. The generator phase will then insert a sync statement at those places.

B. Analysis phase

On the basis of four strategies we employ in the analysis phase, we discuss the problems involved in automatically generating sync statements. The analysis phase takes as input a spawning method and the spawnable calls with the information provided by the recording phase. The analysis phase returns the instructions in front of which sync statements need to be inserted.

1) Fallback strategy: The fallback strategy is used when all other analysis fails. It proposes sync statements immediately behind the spawnable calls. This provides us guaranteed correctness as the resulting program is equivalent to the sequential program. However, this also means that there is no parallelism.

2) On-first-load strategy: This strategy increases parallelism by postponing the sync statement until the first load of a variable in which one of the spawnable calls stores. In Fig. 2 that would be exactly where the sync is now, on line 12, because `result1` is loaded first.

Unfortunately, this places multiple syncs in case of loops. In Fig. 4 this would mean that the sync statement is placed inside the loop between lines 12 and 13. It is preferable to put the sync in front of the `for`-loop, but it is not a problem, since multiple syncs are allowed.

The strategy fails in the example in Fig. 5. When `someCondition` evaluates to `false`, there would be no sync and the result would be incorrect.

This can easily be solved by inserting sync statements in front of all loads. However, there is a larger problem. The correctness is based on the assumption that loads always happen in the part of instructions behind the instruction of the spawnable call. In many cases this will be true, but there may be situations with a backward jump after a spawnable call. The analysis has to be control-flow aware to place sync statements in these situations. However, within basic blocks, where no control flow occurs, the on-first-load strategy suffices.

3) Control-flow-aware strategy: The problems discussed above are solved with this strategy. It succeeds in placing the sync in front of both loads in Fig. 5 and, for example, just in front of a loop that accesses the result of a spawn (Fig. 4).

With help of control flow information provided by the BCEL library [4], a graph of *basic blocks* is constructed. A *basic block* is a sequence of instructions with only one entry and one exit point. So, within the basic block there is no branch instruction other than the last instruction and no instruction is targeted by any branch instruction. A basic block keeps track of the basic blocks it targets and whether the basic block is an *ending basic block* in the graph. This is the case if the last instruction is a `return` or a `throw` instruction.

A *path* is a sequence of basic blocks. An *ending path* is a path where the last basic block is an ending basic block. For every spawnable call in the method the following analysis takes place: find all *ending paths* from the spawnable call on. This includes all possible loops.

Then, for every ending path, the implementation of this strategy tries to construct a *store-to-load path* for every local variable index in which the spawnable call stores. Note that there can be multiple local variable indices when dealing with exceptions. A *store-to-load path* is a path from the basic block of the spawnable call to a load of one of the local variables in which the spawnable call stores.

In this stage all spawnable calls of the spawnable method have been analyzed and every spawnable call is associated with one or more paths from store to load. The implementation retrieves all these paths and removes duplicates. For all these paths it tries to find a basic block from the end that is not in a loop. The result is now one or more basic blocks in which a sync needs to be placed.

For each of these basic blocks, the on-first-load strategy finds the first load of a variable in which one of the spawnable calls stores and this instruction is proposed as an instruction in front of which a sync statement should be inserted.

In the case that a spawnable method signature is of type `void`, results of a spawnable call are not read or exceptions are not handled (there is no catch block), it is not possible to create a store-to-load path. In this case sync statements will be placed at the ends of all ending paths. This means that on every exit from the method, a sync will be placed.

The control-flow-aware strategy contains two additional optimizations:

a) Unnecessary sync statements: It is possible that a spawning method contains multiple exclusive store-to-load paths, but where the sync statement is placed in such a way that it syncs the other paths as well. In this case it is not necessary to insert later sync statements. An example is Fig. 6. There is a path *without* the first read on line 7 and then *with* the second read on line 12, which is exclusive from the path *with* the first read and *without* the second read. Without this optimization, the sync generator would also place a sync before the third for-loop on line 10. However, because there is already a sync before the second for-loop, this sync is left out.

b) Array stores and object putfield: If a spawnable call stores into an object, the load in a store-to-load path is based on the load of the object reference. In many cases, the load is used to read from the object, which needs a sync statement. However, there are situations in which the load of an object reference is used to store into the object, for example a store into an array or to store something in a field of an object. This does not need a sync statement.

```
1 for (int i = 0; i < MAX_SPAWNS; i++) {
2   result[i] = spawnableMethod();
3 }
4
5 sync(); // placed by sync generator
6 for (int i = 0; i < MAX_SPAWNS; i++) {
7   readFirstTime(result[i]);
8 }
9
10 // sync left out by sync generator
11 for (int i = 0; i < MAX_SPAWNS; i++) {
12   readSecondTime(result[i]);
13 }
```

Fig. 6. Multiple exclusive store-to-load paths.

```
1 for (int i = 1; i < MAX_SPAWNS; i++) {
2   result[i] = spawnableMethod();
3 }
4
5 // the reference to object result is loaded,
6 // but sync is not placed here, because it
7 // is a store into an array
8 result[0] = someValue;
9
10 sync(); // by sync generator
11 for (int i = 0; i < MAX_SPAWNS; i++) {
12   read(result[i]);
13 }
```

Fig. 7. Ignoring array stores.

Fig. 7 shows a store into an array on line 8. The object reference for `result` needs to be loaded. Because of this, without optimization, the sync generator would insert a sync in front of the load. However, this optimization recognizes that the load of the object references is used for a store into an array or object field and places it beyond the load of the object reference on line 10.

4) Alias-aware strategy: The previous strategy places sync statements optimally in many cases, but the analysis is incorrect when *aliases* to object references are loaded instead of the *original* object references. Fig. 8 illustrates this. On line 8 and 9, two result objects are created, and on line 10 and 11 new references are pointing to the same objects. The results are stored using references `r1` and `r2`, but the results are loaded using the aliases `a1` and `a2`. The `sync()` on line 17 is incorrectly placed.

Alias analysis is typically imprecise [5], but aliasing within a spawning method is not so common in the Satin programming model. Therefore, we take a pragmatic approach and try to detect situations where aliasing can occur.

The alias-aware strategy uses the control flow graph to check whether the object references in which spawnable calls store, are loaded or stored *before* the spawnable call is executed. A load or store of such an object may mean that an alias has been created. If the sync generator detects such a load or store, it will revert to the fallback strategy described above, and will issue a warning that indicates that the sync generator cannot place sync statements optimally.

```
1 class Result {
2   int v;
3 }
4
5 int spawningMethod() {
6   if (stopCondition) return 0;
7
8   Result r1 = new Result();
9   Result r2 = new Result();
10  Result a1 = r1;
11  Result a2 = r2;
12
13  r1.v = spawnableMethod();
14  r2.v = spawnableMethod();
15
16  int sum = a1.v + a2.v;
17  sync(); // incorrect, should be on line 15.
18  return sum;
19 }
```

Fig. 8. Incorrect sync placement due to aliasing within a spawning method.

Aliases introduced *after* the spawnable call is executed, can only be introduced by loading the original object reference. The implementation detects this as a load and will insert a sync before the load. Therefore, aliases that are introduced after the spawnable call is executed, do not need special treatment.

Spawnable calls that store into parameters of the spawning method also pose a problem. Aliases to these parameters could have been created before the spawning method was called, and since the implementation has no knowledge about these aliases, it reverts back to the fallback strategy and issues a warning.

In the Satin programming model, it often happens that results are stored in a result object, similar to `r1` and `r2` on lines 8 and 9 of Fig. 8. The class is defined on line 1. Another situation is storing the results of a spawnable call into an array, for instance similar to Fig. 7. Creating these result objects or arrays means that the object references need to be loaded before the spawnable calls are called. However, creating an array can never introduce aliases. Creating an object can, but only if a reference to the new object escapes the constructor.

The alias-aware strategy detects these two cases. Array creation (one- or multi-dimensional) is recognized and allowed. For new objects, we perform a simple escape analysis on the called constructor to confirm that no reference to the new object escapes the constructor. Fig. 8 shows on line 8 and 9 creation of two new object with the default constructor. These two statements do not introduce aliasing.

To summarize: the alias-aware strategy cannot optimally place sync statements when a spawnable call stores into object references, except for new arrays and new simple objects as described above. However, it can detect possible aliasing and warn for non-optimal sync placement. This gives the programmer the opportunity to either restructure the code to reduce possible aliasing, or insert sync statements manually and disable the sync generation compilation pass.

V. Evaluation

The sync generator is evaluated using 35 pre-existing real-world Satin applications, amongst others a SAT solver [6], N–body simulation [1], Grammar-based text analysis [6], Grammar induction [7], Gene sequence alignment [1], FFT, and Game-tree search [8].

To evaluate the performance of the sync generator, the applications are stripped from sync statements. The applications are recompiled with the Satin compiler that runs the sync generator pass. The resulting bytecode is carefully examined and compared to the original bytecode. The placement of sync statements is compared in relation to the control flow and variables on which the correctness and amount of parallelism depends.

The placement is said to be optimal, if the placement is the same as the placement of the programmer or later in the control flow (but still correct). The sync generator may place sync statements in such a way that sync is called multiple times. This has a negligible performance effect.

Table I shows the results. The first column shows the name of the application. Column two shows whether the generated syncs are inserted at an optimal place. In the case that sync statements are not placed optimally, the third column indicates whether the programmer receives a warning, due to aliasing. Column four shows some additional notes.

The table does not show whether the applications are correct, because this is true for all the applications. Correct means that the applications with automatically inserted sync statements give the same result as the original applications.

The overall result is that the sync generator is able to compute an optimal sync placement for 31 out of the 35 applications. In the application FFT, the sync generator inserted correct sync statements that were missing in the original incorrect application. We also tested the sync generator on 12 additional test applications which resulted in optimal placement in all cases.

A. Evaluation per application

To get more insight in the results, we discuss the applications separately. The applications that have optimal sync placement and no further problems are omitted.

a) Adaptive integration, Binomial coefficients, Fifteen puzzle: In these applications the programmer placed syncs where the sync generator leaves them out. The sync generator is correct and these syncs can be safely left out. This shows that unnecessary sync insertion is no problem.

b) Checkers (negamax version): This application is the only application that does not have optimal placement and has no warning. Fig. 9 shows the cause. The method `srch()` has a variable `beta_cutoff` that has value 0 initially. Depending on this value the control flow breaks out of the for loop. The programmer knows that this variable is not important for the result, only for performance. The sync generator must regard the variable `beta_cutoff` as a variable in which some value is stored inside the catch block. This means that before this variable is loaded there should be a sync statement.

```
1 void srch() throws Result {
2   int beta_cutoff = 0;
3
4   for(x = 0; x < count; x++) {
5     // sequential work
6     try {
7       spawn_srch();
8     }
9     catch (Result res) {
10      killer[p.ply] = move_list[res.choice_ix];
11      // other stuff
12      if (-res.score >= p.beta) {
13        beta_cutoff = 1;
14        abort();
15      }
16      return;
17    }
18    // sync generator's sync
19    if (beta_cutoff != 0) break;
20  }
21  // programmers sync
22 }
```

Fig. 9. Skeleton of method `srch()` in Checkers (negamax version).

Unfortunately, this leads to sync placement right behind the try and catch blocks of the spawnable call `spawn_srch()` within the for loop on line 18, resulting in sequential execution of this application.

c) FFT: The sync generator inserts more statements than were inserted by the programmer. Surprisingly, the original application was incorrect. The sync generator corrected it.

d) Matrix multiplication (both versions), N-Queens (contest version): These three applications are affected by possible aliasing. The two matrix multiplication applications write into a parameter of the spawning method. Therefore, only the first four of eight calls of the application are executed in parallel. The N-Queens (contest version) application stores into an object that is used by a different spawn and that is also a parameter. Because the sync generator cannot assure that there are no aliases, syncs are placed conservatively taking away much of the parallelism. However, the sync generator warns for non-optimal sync placement in all three cases. This allows the programmer to restructure the code or disable the sync generation pass and insert sync statements manually.

VI. Discussion

The analysis can be made more precise by extending analysis to methods that are called from the spawning method. Fig. 10 illustrates this. Due to possible aliasing and loading of `result1` on line 6, the sync generator will not place the sync on line 10 where we want it. We could extend the analysis to verify that no aliases are created in `createResult()` on lines 2 and 3. We could also analyze `doSomethingHarmless()` on line 6 to verify that it does not load variable `x`.

However, this more precise analysis will never solve the problem in Fig. 9. The `beta_cutoff` variable depends amongst others on what happens in the catch block. An

TABLE I
AUTOMATIC SYNC STATEMENT GENERATION IN REAL-WORLD SATIN APPLICATIONS.

application	optimal	alias warning	notes
Adaptive integration	yes	-	programmer placed unnecessary syncs
Awari: game tree search with the mtdf algorithm [8]			
Reference version with transposition tables	yes	-	
Pre-allocated transposition tables	yes	-	
Transposition tables as structure of arrays	yes	-	
Replicated transposition tables with Java RMI	yes	-	
Replicated transposition tables with sockets	yes	-	
No speculative parallelism	yes	-	
Shared objects version	yes	-	
Binomial coefficients	yes	-	programmer placed unnecessary syncs
Checkers			
Reference version	yes	-	
Negamax search with alpha-beta pruning	**no**	**no**	based on false dependency
Fast Fourier Transform (FFT)	yes	-	original program incorrect, corrected by sync generator
Fifteen puzzle, iterative deepening A* algorithm	yes	-	programmer placed unnecessary syncs
Gene sequence alignment [1]	yes	-	
Grammar induction [7]	yes	-	
Grammar-based text analysis [6]	yes	-	
Knapsack	yes	-	
Matrix multiplication			
Standard version	**no**	**yes**	writing into parameter
shared objects version	**no**	**yes**	writing into parameter
N-body simulation (Barnes-Hut) [1]	yes	-	
N-Queens problem			
Reference version	yes	-	
Using exceptions	yes	-	
Using speculative parallelism and aborts	yes	-	
Using speculative parallelism, aborts and thresholing	yes	-	
Non-speculative version, counting total number of solutions	yes	-	
Version that won the Grids@work 2005 contest [9]	**no**	**yes**	writing into object reference, writing into parameter
Othello: game tree search with the mtdf algorithm [8]	yes	-	
Prime factorization	yes	-	
Raytracer	yes	-	
SAT solver [6]	yes	-	
Text indexing	yes	-	
Traveling Salesman Problem			
Reference version	yes	-	
Shared objects version	yes	-	
Young-brothers-wait version	yes	-	
VLSI cell router (LocusRoute) [10], [11]	yes	-	

```
1 int spawningMethod() {
2   Result result1 = createResult();
3   Result result2 = createResult();
4
5   result1.x = spawningMethod();
6   result1.doSomethingHarmless();
7
8   result2.x = spawningMethod();
9
10  //sync(); preferred place
11  return result1.x + result2.x;
12 }
```

Fig. 10. Situation not optimally analyzed.

analyzer will never find out whether this is relevant for a correct result or not.

The evaluation of the applications revealed that some execution time measurements became incorrect, because the sync generator postponed sync statements beyond the time measurements of the programmer. To place time measurements correctly, programmers have to either first load one of the results of the spawnable calls, or insert sync statements themselves, which is still possible.

In principle, the analysis we describe in this paper can be used to analyze Cilk [3] programs. However, due to the use of C as a base language, the implementation needs to take into account unrestricted aliasing (aliasing to basic types, casting) and global variables which are more common on shared memory. In addition, our analysis operates on bytecode and does not need the source code. A similar binary rewriting approach would be difficult with Cilk code.

VII. RELATED WORK

Satin [1] is closely related to Cilk by Blumofe et al. [3]. Both systems provide a divide-and-conquer programming model with asynchronous spawnable calls and a local barrier synchronization primitive called sync. Cilk differs from Satin in the base language. Where Cilk extends C and C++ with keywords such as `spawn` and `sync`, Satin uses the Java type and class systems to make similar annotations. As a result, a Satin program is still a valid Java program, whereas a Cilk program is not a valid C or C++ program. Another difference is

that Cilk targets multi-core architectures, whereas Satin targets execution on clusters and grids using the Ibis system [2] that provides communication primitives to nodes on a grid.

We are unaware of any previous work on trying to make sync statements implicit in a divide-and-conquer model. Implicit sync statements show resemblance with implicit futures, introduced by Baker and Hewitt [12]. A future evaluates an expression in a separate thread or process. The original process can perform other work and blocks when it tries to use the expression of the future until the result is computed. Flow Java [13] is a variant of Java that implements futures through the use of single assignment variables. The difference between implicit syncs and implicit futures is that a future can block on a specific spawn, whereas a sync blocks for all spawns in the method.

The sync generator uses several well known techniques to decide where to place sync statements. We are interested in which variables a spawnable call stores and where these are loaded again. The analysis can be incorrect due to aliasing of variables before a spawnable call is called. We perform simple alias detection, but the analysis could be made more precise through more advanced alias analysis, for example with help of the Spark framework [14]. We also perform a simple escape analysis that can also be more precise, for example by implementing work of Whaley and Rinard [15] or Blanchet [16].

VIII. Conclusion

Satin provides a divide-and-conquer programming model to execute applications efficiently on clouds and grids. Programmers annotate recursive calls to be spawnable to make sure that these calls are executed in parallel on the grid. Sync statements indicate where the system has to block to wait for the result. In this paper we discussed a sync generator that automatically inserts these sync statements for the programmer.

We conclude that it is possible to automatically insert sync statements in Satin code in such a way that every resulting program is correct. Correctness in this sense is that the resulting program that is run on a grid will deliver the same result as the sequential version. The fact that we provide correctness is well illustrated by a previously incorrect application (FFT) that was corrected by the sync generator.

In many cases the sync generator will be able to place sync statements optimally. An optimal place in this sense is that as many independent asynchronous spawnable calls as possible are called within a spawning method. To perform the analysis, the sync generator needs to be control flow and alias aware.

We evaluated the sync generator using 35 pre-existing real-world applications, and an additional 12 test applications. The sync generator finds an optimal place in all tests, and in 31 out of 35 real-world applications. For three applications, the sync generator does not place syncs optimally due to possible aliasing. However, these applications still have some parallelism and the sync generator also warns for non-optimal placement. One application is completely sequentialized by the sync generator due to a false dependency, deliberately introduced by the programmer, that does not affect the result of the computation, but only the performance. However, no analyzer can dismiss this false dependency and place sync statements in such a way that more parallelism is enabled.

The current analysis could be made more precise by extending the analysis to methods that are called from the spawning method. However, this analysis also cannot disregard false dependencies as above.

References

[1] R. V. van Nieuwpoort, G. Wrzesińska, C. J. H. Jacobs, and H. E. Bal, "Satin: A High-Level and Efficient Grid Programming Model," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 3, pp. 1–39, 2010.

[2] H. E. Bal, J. Maassen, R. V. van Nieuwpoort, N. Drost, R. Kemp, N. Palmer, G. Wrzesińska, T. Kielmann, F. Seinstra, and C. J. H. Jacobs, "Real-World Distributed Computing with Ibis," *Computer*, vol. 43, pp. 54–62, August 2010.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, 1995.

[4] *Byte Code Engineering Library*, http://jakarta.apache.org/bcel.

[5] M. Hind, "Pointer Analysis: Haven't We Solved This Problem Yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ser. PASTE '01. New York, NY, USA: ACM, 2001, pp. 54–61.

[6] K. van Reeuwijk, R. V. van Niewpoort, and H. E. Bal, "Developing Java grid applications with Ibis," in *Proc. of the 11th International Euro-Par Conference*, Lisbon, Portugal, September 2005, pp. 411–420.

[7] P. Adriaans and C. Jacobs, "Using MDL for Grammar Induction," in *8th International Colloquium on Grammatical Inference (ICGI '06)*, Tokyo, Japan, September 2006.

[8] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin, "Best-First Fixed-Depth Minimax Algorithms," *Artificial Intelligence*, vol. 87, no. 1–2, pp. 255–293, November 1996.

[9] R. V. van Nieuwpoort, J. Maassen, A. Agapi, A. Oprescu, and T. Kielmann, "Experiences Deploying Parallel Applications on a Large-scale Grid," in *EXPGRID - Experimental Grid testbeds for the assessment of large-scale distributed applications and tools, workshop in conjunction with HPDC-15*, june 2006.

[10] J. Rose, "LocusRoute: A Parallel Global Router for Standard Cells," *Design Automation Conference*, pp. 189–195, 1988.

[11] G. Wrzesińska, J. Maassen, K. Verstoep, and H. E. Bal, "Satin++: Divide-and-share on the grid," in *2nd IEEE International Conference on e-Science and Grid Computing*, Amsterdam, The Netherlands, 2007.

[12] H. C. Baker, Jr. and C. Hewitt, "The Incremental Garbage Collection of Processes," in *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*. New York, NY, USA: ACM, 1977, pp. 55–59.

[13] F. Drejhammar, C. Schulte, P. Brand, and S. Haridi, "Flow Java: Declarative Concurrency for Java," in *Logic Programming*, ser. Lecture Notes in Computer Science, C. Palamidessi, Ed. Springer Berlin / Heidelberg, 2003, vol. 2916, pp. 346–360.

[14] O. Lhoták and L. Hendren, "Scaling Java Points-to Analysis Using Spark," in *Compiler Construction*, ser. Lecture Notes in Computer Science, G. Hedin, Ed. Springer Berlin / Heidelberg, 2003, vol. 2622, pp. 153–169.

[15] J. Whaley and M. Rinard, "Compositional Pointer and Escape Analysis for Java Programs," in *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '99. New York, NY, USA: ACM, 1999, pp. 187–206.

[16] B. Blanchet, "Escape Analysis for Object-Oriented Languages: Application to Java," in *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '99. New York, NY, USA: ACM, 1999, pp. 20–34.