Parallelizing Deadlock Resolution in Symbolic Synthesis of Distributed Programs*

Fuad Abujarad
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
abujarad@cse.msu.edu

Borzoo Bonakdarpour
VERIMAG
Centre Équation, 2 ave de Vignate
38610 Gières, France
borzoo@imag.fr

Sandeep S. Kulkarni
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
sandeep@cse.msu.edu

Previous work has shown that there are two major complexity barriers in the synthesis of fault-tolerant distributed programs: (1) generation of fault-span, the set of states reachable in the presence of faults, and (2) resolving deadlock states, from where the program has no outgoing transitions. Of these, the former closely resembles with model checking and, hence, techniques for efficient verification are directly applicable to it. Hence, we focus on expediting the latter with the use of multi-core technology.

We present two approaches for parallelization by considering different design choices. The first approach is based on the computation of equivalence classes of program transitions (called *group* computation) that are needed due to the issue of distribution (i.e., inability of processes to atomically read and write all program variables). We show that in most cases the speedup of this approach is close to the ideal speedup and in some cases it is superlinear. The second approach uses traditional technique of partitioning deadlock states among multiple threads. However, our experiments show that the speedup for this approach is small. Consequently, our analysis demonstrates that a simple approach of parallelizing the group computation is likely to be the effective method for using multi-core computing in the context of deadlock resolution.

Keywords: Program transformation, Symbolic synthesis, Multi-core algorithm, Distributed programs.

1 Introduction

Given the current trend in processor design where the number of transistors keeps growing as directed by Moore's law, but where clock speed remains relatively flat, it is expected that multi-core computing will be the key for utilizing such computers most effectively. As argued in [12], it is expected that programs and protocols from distributed computing will be especially beneficial in exploiting such multi-core computers.

One of the crucial issues in distributed computing is fault-tolerance. Moreover, as part of maintenance, it may be necessary to modify a program to add fault-tolerance to faults that were not considered in the original design. In such maintenance, it would be required that the existing functional properties of

*This work was partially sponsored by the COMBEST European project, NSF CNS 0914913 , and ONR Grant N00014-01-1-0744.

PDMC 2009
EPTCS ??, 20??, pp. 1–15, doi:10.4204/EPTCS.??.??

© F. Abujarad, B. Bonakdarpour & S. S. Kulkarni
This work is licensed under the
Creative Commons Attribution License.

the program continue to be preserved during the addition of fault-tolerance, i.e., no bugs should be introduced in such addition. For this reason, it would be highly beneficial if one could add such fault-tolerance properties using automated techniques.

One difficulty in adding fault-tolerance using automated techniques, however, is its complexity. In our previous work [4], we developed a symbolic (BDD-based) algorithm for adding fault-tolerance to distributed programs specified in terms of transition system with state space larger than 10^{30}. We also identified a set of bottlenecks that compromise the effectiveness of our algorithm. Based on the analysis of the experimental results from [4], we observed that depending upon the structure of the given distributed intolerant program, performance of synthesis suffers from two major complexity obstacles, namely *generation of fault-span* (i.e., the set of reachable states in the presence of faults) and *resolution of deadlock states*.

Our focus in this paper is to evaluate effectiveness of different approaches that utilize multi-core computing to reduce the time complexity of adding fault-tolerance to distributed programs. In particular, we focus on the second problem, i.e., resolution of deadlock states. Deadlock resolution is especially crucial in the context of dependable systems, as it guarantees that the synthesized fault-tolerant program meets its liveness requirements even in the presence of faults. A program may reach a deadlock state due to the fact that faults perturb program to a new state that was not considered in the fault-intolerant program. Or, it may reach a deadlock state, as some program actions are removed (e.g., because they violate safety in the presence of faults). To resolve a deadlock state, we either need to provide *recovery actions* that allow program to continue its execution or *eliminate* the deadlock state by preventing the program execution from reaching it.

To evaluate the effectiveness of multi-core computing, we first need to identify bottleneck(s) where multi-core features can provide the maximum impact. To this end, we present two approaches for parallelization. The first approach is based on the distributed nature of the program being synthesized. In particular, when a new transition is added (respectively, removed), since the process executing it has only a partial view of the program variables, we need to add (respectively, remove) a *group* of transitions based on the variables that cannot be read by the process. The second approach is based on partitioning deadlock states among multiple threads. We show that while in most cases the speedup of the first approach is close to the ideal speedup and in some cases it is superlinear the second approach provides a small performance benefit. Based on the analysis of these results, we argue that the simple approach that parallelizes the group computation is likely to provide maximum benefit in the context of deadlock resolution for synthesis of distributed programs.

Contributions of the paper. Our contributions in this paper is as follows:

- We present two approaches for expediting resolution of deadlock states in automated synthesis of fault-tolerance.
- We analyze these approaches in terms of three classic examples from distributed computing: Byzantine agreement [15], agreement in the presence of both failstop and Byzantine faults, and token ring [3].
- We discuss different design choices considered in these two approaches.

Organization of the paper. The rest of the paper is organized as follows. In Section 2, we define distributed programs and specifications. We illustrate the issues involved in the synthesis problem in the context of Byzantine agreement in Section 3. We present our two approaches, the corresponding experimental results and analysis in Sections 4 and 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 Programs, Specifications and Problem Statement

In this section, we define the problem statement for adding fault-tolerance. We begin with a fault-intolerant program, say p, that is correct in the absence of faults. We let p be specified in terms of its state space, S_p, and a set of transitions, $\delta_p \subseteq S_p \times S_p$. Whenever it is clear from the context, we use p and its transitions δ_p interchangeably. A sequence of states, $\langle s_0, s_1, ... \rangle$ (denoted by σ) is a computation of p iff (1) $(\forall j : 0 < j < \mathit{length}(\sigma) : (s_{j-1}, s_j) \in p)$, i.e., in each *step* of this sequence, a transition of p is executed, and (2) if the sequence is finite and terminates in s_j then $\forall s' :: (s_j, s') \notin p$ (a finite computation reaches a state from where there is no outgoing transition). A special subset of S_p, say S, identifies an *invariant* of p. By this we mean that if a computation of p begins in a state where S is true, then (1) S is true at all states of that computation and (2) the computation is *correct*. Since the algorithm for addition of fault-tolerance begins with a program that is correct in the *absence* of faults, we do not explicitly need the program specification in the absence of faults. Instead, the predicate S is used to determine states where the fault-tolerant program could recover in the presence of faults.

The goal of an algorithm that adds fault-tolerance is to begin with a program p and its invariant S to derive a fault-tolerant program, say p', and its invariant, say S'. Clearly, one additional input to such an algorithm is f, the class of faults to which tolerance is to be added. Faults are also specified as a subset of $S_p \times S_p$. Note that this allows modeling of different types of faults, such as transients, Byzantine (see Section 3.1), crash faults, etc. Yet another input to the algorithm for adding fault-tolerance is a safety specification, say $SPEC_{bt}$, that should not be violated in the presence of faults. We let $SPEC_{bt}$ also be specified by a set of *bad* transitions, i.e., $SPEC_{bt}$ is a subset of $S_p \times S_p$[1]. Thus, it is required that in the presence of faults, the program should not execute a transition from $SPEC_{bt}$.

Now we define the problem of adding fault-tolerance. Let the input program be p, invariant S, faults f, and safety specification $SPEC_{bt}$. Since our goal is to add fault-tolerance only, we require that no new computations are added in the *absence* of faults. Thus, if the output after adding fault-tolerance is program p' and invariant S', then S' should not include any states that are not in S; without this restriction, p' can begin in a state from where the correctness of p is unknown. Likewise, if (s_0, s_1) is a transition of p' and $s_0 \in S'$ then (s_0, s_1) must also be a transition of p; without this restriction, p' will have new computations in the absence of faults. Also, if p' has no outgoing transition from state $s_0 \in S'$, then it must be the case that p also has no outgoing transitions from s_0; without this restriction, p' may deadlock in a state that had no correspondence with p.

Additionally, p' should be fault-tolerant. Thus, during the computation of p', if faults from f occur then the program may be perturbed to a state outside S'. Just like the invariant captured the boundary up to which the program can reach in the *absence* of faults, we can identify a boundary upto which the program can reach in the *presence* of faults. Let this boundary (denoted by fault-span) be T. Thus, if any transition of p or f begins in a state where T is true, then it must terminate in a state where T is true. Moreover, if p' is permitted to execute for a long enough time without perturbation of a fault, then p' should reach a state where its invariant S' is true. Based on this discussion, we define the problem of adding fault-tolerance as follows:

Problem statement 2.1 Given p, S, f and $SPEC_{bt}$, identify p' and S' such that:

- $(C1)$: Constraints on the invariant
  - $S' \neq \phi$,
  - $S' \Rightarrow S$,

[1] As shown in [14], permitting more general specifications can significantly increase the complexity of synthesis. We also showed that representing safety specification using a set of transitions is expressive enough for most practical programs.

- $(C2)$: Constraints on transitions within invariant
  - $(s_0, s_1) \in p' \wedge s_0 \in S' \Rightarrow ((s_1 \in S') \wedge (s_0, s_1) \in p)$,
  - $s_0 \in S' \wedge (\forall s_1 :: (s_0, s_1) \notin p') \Rightarrow (\forall s_1 :: (s_0, s_1) \notin p)$, and
- $(C3)$ There exists T such that
  - $S' \Rightarrow T$,
  - $s_0 \in T \wedge (s_0, s_1) \in (p' \cup f) \Rightarrow s_1 \in T \wedge (s_0, s_1) \notin SPEC_{bt}$
  - $s_0 \in T \wedge \langle s_0, s_1, ... \rangle$ is a computation of $p' \Rightarrow (\exists j : 0 < j < length(\langle s_0, s_1, ... \rangle) : s_j \in S')$

3 Issues in Automated Synthesis of Fault-Tolerant Programs

In this section, we use the example of *Byzantine agreement* [15] (denoted *BA*) to describe the issues in automated synthesis of fault-tolerant programs. Towards this end, in Section 3.1, we describe the inputs used for synthesizing the Byzantine agreement problem. Subsequently, in Section 3.2, we identify the need for explicit modeling of read-write restrictions imposed by the nature of the distributed program. Finally, in Section 3.3, we describe how deadlock states get created while revising the program for adding fault-tolerance and illustrate our approach for managing them.

3.1 Input for Byzantine Agreement Problem

The Byzantine agreement problem (*BA*) consists of a *general*, say g, and three (or more) *non-general* processes, say j, k, and l. The agreement problem requires a process to copy the decision chosen by the general (0 or 1) and finalize (output) the decision (subject to some constraints). Thus, each process of *BA* maintains a decision d; for the general, the decision can be either 0 or 1, and for the non-general processes, the decision can be 0, 1, or $\bot$, where the value $\bot$ denotes that the corresponding process has not yet received the decision from the general. Each non-general process also maintains a Boolean variable f that denotes whether that process has finalized its decision. For each process, a Boolean variable b shows whether or not the process is Byzantine; the read/write restrictions (described in Section 3.2), ensure that a process cannot determine if other processes are Byzantine. A Byzantine process can output different decision to different processes. Thus, a state of the program is obtained by assigning each variable, listed below, a value from its domain. And, the state space of the program is the set of all possible states.

$V = \{d.g\} \cup$	(the general decision variables):$\{0, 1\}$
$\{d.j, d.k, d.l\} \cup$	(the processes decision variables):$\{0, 1, \bot\}$
$\{f.j, f.k, f.l\} \cup$	(finalized?):$\{false, true\}$
$\{b.g, b.j, b.k, b.l\}$.	(Byzantine?):$\{false, true\}$

Fault-intolerant program. To concisely describe the transitions of the (fault-intolerant) version of *BA*, we use guarded commands of the form $g \longrightarrow st$, where g is a predicate involving the above program variables and st updates the above program variables. The command $g \longrightarrow st$ corresponds to the set of transitions $\{(s_0, s_1) : g$ is true in s_0 and s_1 is obtained by *executing* st in state $s_0\}$. Thus, the transitions of a non-general process, say j, is specified by the following two actions:

$$BA_{intol_j} :: BA1_j :: (d.j = \bot) \wedge (f.j = false) \wedge (b.j = false) \longrightarrow d.j := d.g$$
$$BA2_j :: (d.j \neq \bot) \wedge (f.j = false) \wedge (b.j = false) \longrightarrow f.j := true$$

We include similar transitions for k and l as well. Note that the general does not need explicit actions; the action by which the general sends the decision to j is modeled by $BA1_j$.

Specification. The safety specification of *BA* requires *validity* and *agreement*. *Validity* requires that if the general is non-Byzantine, then the final decision of a non-Byzantine, non-general must be the same as that of the general. Additionally, *agreement* requires that the final decision of any two non-Byzantine, non-generals must be equal. Finally, once a non-Byzantine process finalizes (outputs) its decision, it cannot change it.

Faults. A fault transition can cause a process to become Byzantine, if no other process is initially Byzantine. Also, a fault can arbitrarily change the d and f values of a Byzantine process. The fault transitions that affect a process, say j, of *BA* are as follows: (We include similar actions for k, l, and g)

$$\begin{array}{lll} F1 :: \neg b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l & \longrightarrow & b.j := true \\ F2 :: b.j & \longrightarrow & d.j, f.j := 0|1, false|true \end{array}$$

where $d.j := 0|1$ means that $d.j$ could be assigned either 0 or 1. In case of the general process, the second action does not change the value of any f-variable.

Goal of automated Addition of fault-tolerance. Given the set of faults ($F1\&F2$), the goal of a synthesis algorithm is to start from the intolerant program (BA_{intol_j}) and generate the fault-tolerant program ($BA_{tolerant_j}$):

$$\begin{array}{lllll} BA_{tolerant_j} :: & BA1_j :: (d.j = \bot) \wedge (f.j = false) \wedge (b.j = false) & \longrightarrow & d.j := d.g \\ & BA2_j :: (d.j \neq \bot) \wedge (f.j = false) \wedge (d.j = d.l \vee d.j = d.k) & \longrightarrow & f.j := true \\ & BA3_j :: (d.l = 0) \wedge (d.k = 0) \wedge (d.j = 1) \wedge (f.j = 0) & \longrightarrow & d.j, f.j := 0, 0|1 \\ & BA4_j :: (d.l = 1) \wedge (d.k = 1) \wedge (d.j = 0) \wedge (f.j = 0) & \longrightarrow & d.j, f.j := 1, 0|1 \end{array}$$

In the above program, the first action is identical to that of the intolerant program. The second action is restricted to execute only in the states where another process has the same d value. Actions (3&4) are for fixing the process decision through appropriate recovery.

3.2 Group Computation: The Need for Modeling Read/Write Restrictions

A process in a distributed program has a partial view of the program variables. For example, in the context of the Byzantine agreement example from Section 3.1, process j is allowed to read $R_j = \{b.j, d.j, f.j, d.k, d.l, d.g\}$ and it is allowed to write $W_j = \{d.j, f.j\}$. Observe that this modeling prevents j from knowing whether other processes are Byzantine.

With such read/write restriction, if process j were to include an action of the form 'if $b.k$ is true then change $d.j$ to 0' then it must also include a transition of the form 'if $b.k$ is false then change $d.j$ to 0'. In general, if transition (s_0, s_1) is to be included as a transition of process j then we must also include a corresponding equivalence class of transitions (called *group* of transitions) that differ only in terms of variables that j cannot read. The same mechanism has to be applied for removing transitions as well.

More generally, let j be a process, let R_j (respectively, W_j) be the set of variables that j can read (respectively write), where $W_j \subseteq R_j$, and let $v_a(s_0)$ denote the value of variable v_a in the state s_0. Then if (s_0, s_1) is a transition that is included as a transition of j then we must also include the corresponding equivalence class of transitions of the form (s_2, s_3) where s_0 and s_2 (respectively s_1 and s_3) are indistinguishable for j, i.e., they differ only in terms of the variables that j cannot read. This equivalence class of transitions for (s_0, s_1) is given by the following formula:

$$\begin{array}{l} group_j((s_0, s_1)) = \bigvee_{(s_2, s_3)} \\ \qquad (\ \bigwedge_{v \notin R_j} (v(s_0) = v(s_1)\ \wedge\ v(s_2) = v(s_3)) \wedge \\ \qquad\ \bigwedge_{v \in R_j} (v(s_0) = v(s_2)\ \wedge\ v(s_1) = v(s_3))\ ). \end{array}$$

3.3 Need for Deadlock Resolution

During synthesis, we analyze the effect of faults on the given fault-intolerant program and identify a fault-tolerant program that meets the constraints of Problem Statement 2.1. This involves addition of new transitions as well as removal of existing transitions. In this section, we utilize the Byzantine agreement problem to illustrate how deadlocks states get created during the execution of the synthesis algorithm and identify two general approaches for resolving them (be them sequential or parallel).

- **Deadlock scenario 1 and use of recovery actions.** One legitimate state, say s, for the Byzantine agreement program is a state where all processes are non-Byzantine, $d.g$ is 0 and the decision of all non-generals is $\perp$. In this state, the general has chosen the value 0 and no non-general has received any value. From this state, the general can become Byzantine and change its value from 0 to 1 arbitrarily. Hence, a non-general can receive either 0 or 1 from the general. Clearly, starting from s, in the presence of faults ($F1$ & $F2$), the program (BA_{intol}) can reach a state, say s_1, where $d.g = d.j = d.k = 0, b.g = true, d.l = 1, f.l = 0$. From such a state, transitions of the fault-intolerant program violate agreement, if they allow j (or k) and l to finalize their decision. If we remove these safety violating transitions then there are no other transitions from state s_1. In other words, during synthesis, we encounter that state s_1 is a deadlock state. One can resolve this deadlock state by simply adding a *recovery* transition that changes $d.l$ to 0.
- **Deadlock scenario 2 and need for elimination.** Again, consider the execution of the program (BA_{intol}) in the presence of faults ($F1$ & $F2$) starting from state s in the previous scenario. From s, the program can also reach a state, say s_2, where $d.g = d.j = d.k = 0, b.g = true, d.l = 1, f.l = 1$; state s_2 differs from s_1 in the previous scenario in terms of the value of $f.l$. Unlike s_1 in the previous scenario, since l has finalized its decision, we cannot resolve s_2 by adding safe recovery. Since safe recovery from s_2 cannot be added, the only choice for designing a fault-tolerant program is to ensure that state s_2 is never reached in the fault-tolerant program by removing transitions that reach s_2 using backward reachability analysis. However, removal of such transitions can potentially create more deadlock states that have to be eliminated.

To maximize the success of synthesis algorithm, our approach to handle deadlock states is as follows: Whenever possible, we add recovery transition(s) from the deadlock states to a legitimate state. However, if no recovery transition(s) can be added from a deadlock state, we try to eliminate it by preventing the program from reaching the state. In this paper, we utilize parallelism to expedite these two aspects of deadlock resolution: adding recovery and eliminating deadlock states.

4 Approach 1: Parallelizing Group Computation

In this section, we present our approach for parallelizing group computation to expedite synthesis of fault-tolerant programs. First, in Section 4.1, we identify different design choices in devising our parallel algorithm. Then, in Section 4.2, we describe our approach for parallelizing the group computation. In Section 4.3, we provide experimental results. Finally, in Section 4.4, we analyze the experimental results to evaluate the effectiveness of parallelization for group computation.

4.1 Design Choices

The structure of the group computation permits an efficient way to parallelize it. In particular, whenever some recovery transitions are added for dealing with a deadlock state or some states are removed for

ensuring that a deadlock state is not reached, we can utilize multiple threads in a master-slave fashion to expedite the group computation.The context of our approach targets multi-processor/core shared memory infrastructure. Although we did not specifically analyze the influence of local memory sharing on the performance, we expect our solution to give similar results when it uses multi-core or multi-processor architecture. During the analysis for utilizing multiple cores effectively, we make the following observations/design choices.

- **Multiple BDD managers versus reentrant BDD package.** We chose to utilize different instances of BDD packages for each thread. Thus, at the time of group computation, each thread obtains a copy of the BDD corresponding to the recovery transitions being added. In part, this is motivated by the fact that existing parallel implementations have shown limited speedup (cf. Section 6). Also, we argue that the increased space complexity of this approach is acceptable in the context of synthesis, since the time complexity of the synthesis algorithm is high (as opposed to model checking) and we often run out of time before we run out of space.
- **Synchronization overhead.** The group computation is rather fine-grained, i.e., the time to compute a group of recovery transitions that are to be added to an input program is small (100-500ms on a normal machine). Hence, the overhead of creating multiple threads needs to be small. With this motivation, our algorithm creates the required set of threads up front and utilizes mutexes to synchronize them. This synchrnozation provides a significant benefit over creating and destroying threads for each group operation.
- **Load balancing.** Load balancing among several threads is desirable so that all threads take approximately the same amount of time in performing their task. To perform a group computation for recovery transitions being added, we need to evaluate the effect of read/write restrictions imposed by each process. A static way to parallelize this is to let each thread compute the set of transitions caused by read/write restrictions of a (given) subset of processes. A dynamic way is to consider the set of processes for which a group computation is to be performed as a *shared pool of tasks* and allow each thread to pick one task after it finishes the previous one. We find that given the small duration of each group computation, static partitioning of the group computation works better than dynamic partitioning since the overhead of dynamic partitioning is high.

4.2 Algorithm Description

Based on these design choices, the algorithm consists of three parts: initialization, assignment of tasks to worker threads and computation of group with worker threads.

Initialization. In the initialization phase, the master thread creates all required worker threads by calling the algorithm InitiateThreads (cf. Algorithm 1). These threads stay idle until a group computation is required and terminate when the synthesis algorithm ends. Due to the design choice for load balancing, the algorithm distributes the work load among the available threads statically (Lines 4-8). Then, it creates all the required worker threads (Line 10).

Tasks for worker thread. Initially, the algorithm WorkerThread (cf. Algorithm 2) locks the mutexes *mutexStart* and *mutexStop* (Lines 1-2). Then, it waits until the master thread unlocks the *mutexStart* mutex (Line 5). At this point, the worker thread starts computing the part of the group associated with this thread. This section of WorkerThread (Lines 7-15) is similar to the computing groups in the sequential setting except rather than finding the group for all the processes, the WorkerThread algorithm finds the group for a subset of processes (Line 8). The function *AllowWrite* relaxes a predicate with respect to the

Algorithm 1 InitiateThreads

Input: *noOfProcesses*, *noOfThreads*.

```
1: if noOfProcesses < noOfThreads then
2:         return ERROR;
3: end if
4: for i := 0 to noOfThreads − 1 do
5:         BDDMgr[i] = Clone(masterBDDManager) ;
6:         startP[i] := ⌊ i × noOfProcesses / noOfThreads ⌋;
7:         endP[i] := ⌊ (i+1) × noOfProcesses / noOfThreads ⌋ − 1;
8: end for
9: for thID := 0 to noOfThreads − 1 do
10:        SpawnThread ⇝ WorkerThread(thID);
11: end for
```

variables that the corresponding process is allowed to modify. The function *Transfer* transfers a BDD from one manager to another manager. And, the function *FindGroup* adds read restrictions to a group predicate. When the computation is completed, the worker thread notifies the master thread by unlocking the mutex *mutexStop* (Line 17).

Algorithm 2 WorkerThread

Input: *thID*.

```
    // Initial locking of the mutexes
1:  mutex_lock(thData[thID].mutexStart);
2:  mutex_lock(thData[thID].mutexStop);
3:  while true do
4:          // Waiting for signal from the master thread
5:          mutex_lock(thData[thID].mutexStart);
6:          gtr[id] := false;
7:          tPred := endP[thID] − startP[thID] + 1 ;
8:          for i := 0 to (endP[thID] − startP[thID]) + 1 do
9:                  tPred[i] := thData[thID].trans ∧ allowWrite[i + startP[thID]].Transfer(BDDMgr[thID]);
10:                 tPred[i] := FindGroup(tPred[i], i, thID);
11:         end for
12:         thData[thID].result := false;
13:         for i := 0 to (endP[thID] − startP[thID]) + 1 do
14:                 thData[thID].result := thData[thID].result ∨ tPred[i];
15:         end for
16:         // Triggering the master thread that this thread is done
17:         mutex_unlock(thData[thID].mutexStop);
18: end while
```

Tasks for master thread. Given transition set *tr*, the master thread copies *tr* to each instance of the BDD package used by the worker threads (cf. Algorithm 3, Lines 3-5). Then it assigns a subset of group computation to the worker threads (Lines 6-8) and unlocks them. After the worker threads complete, the master thread collects the results and returns the group BDD associated with the input *tr*.

4.3 Experimental Results

In this section, we describe the respective experimental results in the context of the Byzantine agreement (described in Section 3.1). Throughout this section, all experiments are run on a Sun Fire V40z with 4 dual-core Opteron processors and 16 GB RAM. The BDD representation of the Boolean formulae has been done using the C++ interface to the CUDD package developed at University of Colorado [17]. Throughout this section, we refer to the original implementation of the synthesis algorithm (without

Algorithm 3 MasterThread

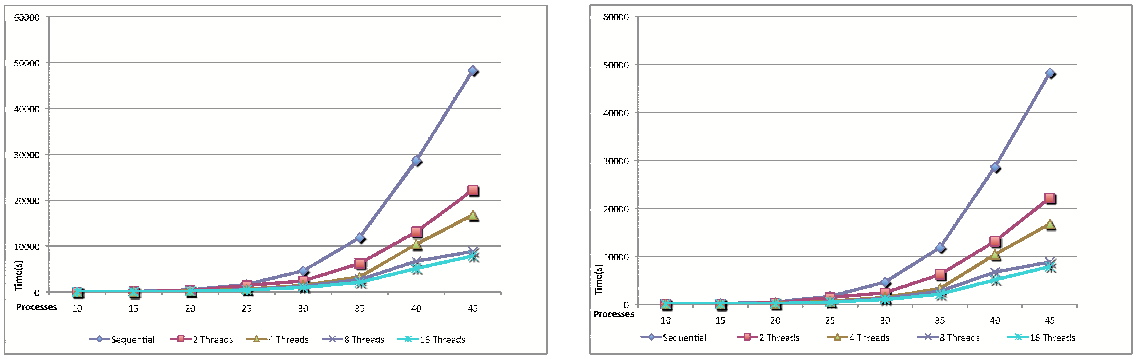
Input: transitions set *thisTr*.
Output: transition group *gAll*.

```
1:  tr := thisTr;
2:  gAll := false;
3:  for i :=  0 to NoOfThreads  1 do
4:          threadData[i].trans := trans.Transfer(BDDMgr[thID]);
5:  end for
    // all idle threads to start computing the group
6:  for i := 0 to NoOfThreads − 1 do
7:          mutex_unlock(thData[i].mutexStart);
8:  end for
    // Waiting for all threads to finish computing the group
9:  for i := 0 to NoOfThreads  1 do
10:         mutex_lock(thData[i].mutexStop);
11: end for
    // Merging the results from all threads
12: for i := 0 to NoOfThreads − 1 do
13:         gAll := gAll + thData[i].results;
14: end for
15: return gAll;
```

parallelism) as *sequential* implementation. We use *X threads* to refer to the parallel algorithm that utilizes X threads.

We would like to note that the synthesis time duration differs between the sequential implementation in this paper and the one in [4] due to other unrelated improvements on the sequential implementation itself. However, the sequential, and the parallel implementations differ only in terms of the modification described in Section 4.2.

We note that our algorithm is deterministic and the testbed is dedicated. Hence, the only non-deterministic factor in time for synthesis is synchronization among threads. Based on our observations and experience, this factor has a negligible impact and, hence, multiple runs on the same data essentially reproduce the same results.



(a) Deadlock Resolution Time (b) Total Synthesis Time

Figure 1: The time required to (a) resolve deadlock states and (b) to synthesize a fault-tolerant program for several numbers of non-general processes of *BA* using sequential and parallel algorithms. The BA has a state space $\approx 4 * 10^{1.08x}$ and reachable state space $\geq 2 * 10^{0.78x}$ where x is the number of process.

In Figure 1, we show the results of using the sequential approach versus the parallel approach (with

multiple threads) to perform the synthesis. All the tests have shown that we gain a significant speedup. For example, in the case of 45 non-general processes and 8 threads we gain a speedup of 6.1 . We can clearly see that the parallel 16-thread version is faster than the corresponding 8-threads version. This was surprising given that there are only 8 cores available. However, upon closer observation, we find that the group computation that is parallelized using threads is fine-grained. Hence, when the master thread uses multiple slave threads for performing the group computation, the slave threads complete quickly and therefore cannot utilize the available resources to the full extent. Hence, creating more threads (than available processors) can improve the performance further.

4.4 Group Time Analysis

In this section, we focus on the effectiveness of the parallelization of group computation by considering the time taken for it in sequential and parallel implementation. Towards this end, we analyze the group computation time for sequential and parallel implementations in the context of three examples: Byzantine agreement, agreement in the presence of failstop and Byzantine faults, and token ring [3]. The results for these examples are included in Tables 1-3. The number of cores used is equal to the number of threads.

To understand the speedup gain provided by our algorithm in Section 4.3, we evaluated the experimental results closely. As an example, consider the case of 32 *BA* processes. For sequential implementation, the total synthesis time is 59.7 minutes of which 55 are used for group computation. Hence, the ideal completion time with 4 cores is 18.45 minutes ($55/4+4.7$). By comparison, the actual time taken in our experiment was 19.1 minutes. Thus, the speedup gained using this approach is close to the ideal speedup.

In some cases, the speedup ratio is less than the number of threads. This is caused by the fact that each group computation takes a very small time and incurs an overhead for thread synchronization. Moreover, as mentioned in Section 3.3, due to the overhead of load balancing, we allocate tasks of each thread statically. Thus, the load of different threads can be slightly uneven. We also observe that the speedup ratio increases with the number of processes in the program being synthesized. This implies that the parallel algorithm will scale to larger problem instances.

		Sequential	*2-threads*		*4-threads*		*8-threads*	
No. of Processes	**Reachable States**	**Group Time**	**Group Time**	**Speedup Ratio**	**Group Time**	**Speedup Ratio**	**Group Time**	**Speedup Ratio**
15	10^{11}	50	29	1.72	17	2.94	11	4.55
24	10^{17}	652	346	1.88	185	3.52	122	5.34
32	10^{22}	3347	1532	2.18	848	3.95	490	6.83
48	10^{33}	33454	14421	2.32	7271	4.60	3837	8.72

Table 1: Group computation time for Byzantine Agreement.

An interesting as well as surprising observation is that when the state space is large enough then the speedup ratio is more than the number of threads. This behavior is caused by the fact that with parallelization, each thread is working on smaller BDDs during the group computation. To understand this behavior, we conducted experiments where we created the threads to perform the group computation and forced them to execute sequentially by adding extra synchronization. We found that such pseudo-sequential run took less time than that used by a purely sequential run.

No. of Processes	Reachable States	*Sequential*	*2-threads*		*4-threads*		*8-threads*	
		Group Time	Group Time	Speedup Ratio	Group Time	Speedup Ratio	Group Time	Speedup Ratio
10	10^{10}	53	24	2.21	23	2.30	30	1.77
15	10^{15}	624	319	1.96	175	3.57	174	3.59
20	10^{20}	4473	2644	1.69	1275	3.51	1128	3.97
25	10^{25}	26154	11739	2.23	6527	4.01	5692	4.59

Table 2: Group computation time for the Agreement problem in the presence of failstop and Byzantine faults.

No. of Processes	Reachable States	*Sequential*	*2-threads*		*4-threads*		*8-threads*	
		Group Time	Group Time	Speedup Ratio	Group Time	Speedup Ratio	Group Time	Speedup Ratio
30	10^{14}	0.32	0.15	2.12	0.10	3.34	0.12	2.75
40	10^{19}	0.84	0.36	2.34	0.22	3.84	0.23	3.59
50	10^{23}	1.82	0.68	2.68	0.39	4.66	0.42	4.37
60	10^{28}	3.22	1.22	2.63	0.67	4.80	0.64	5.01
70	10^{33}	5.36	1.91	2.80	1.06	5.05	0.86	6.23
80	10^{38}	7.77	2.94	2.64	1.53	5.09	1.23	6.30

Table 3: Group computation time for token ring.

5 Approach 2: Alternative (Conventional) Approach

A traditional approach for parallelization in the context of resolving deadlock states, say *ds*, would be to partition the deadlock states into multiple threads and allow each thread to handle the partition assigned to it. For example, we can partition *ds* using the partition predicates, $prt_i, 1 \leq i \leq n$, such that $\bigvee_{i=1}^{n}(prt_i \wedge ds) = ds$. Thus, if two threads are available during synthesis of the Byzantine agreement program then we can let $prt_1 = (d.j = 0)$ and $prt_2 = (d.j \neq 0)$.

Next, in Section 5.1, we discuss some of the design choices we considered for this approach. Subsequently, we describe experimental results in Section 5.2. We argue that for such an approach to work in synthesizing distributed programs, group computation must itself be parallelized.

5.1 Design Choices

To efficiently partition deadlock states among threads, one needs to design a method such that (1) deadlock states are evenly distributed among worker threads, and (2) states considered by different threads for elimination have a small overlap during backtracking. Regarding the first constraint, we can partition deadlock states based on values of some variable and evaluate the size of corresponding BDDs by the number of minterms that satisfy the corresponding formula. Regarding the second constraint, we expect that the overhead for such a split is as high as it requires detailed analysis of program transitions. Hence, instead of satisfying this constraint, we choose to add limited synchronization among threads so that the overlap in the explored states by different threads is small.

After partitioning, one thread would work independently as long as it does not affect states visited by other threads. As discussed in Section 3.3, to resolve a deadlock state, each thread explores a part of the state space using backward reachability. Clearly, when states visited by two threads overlap, we have two options: (1) perform synchronization so that only one thread explores any state or (2) allow two threads to explore the states concurrently and resolve any inconsistencies that may be created.

We find that the first option by itself is very expensive/impossible due to the fact that with the use of BDDs, each thread explores a set of states specified by the BDD. And, since each thread begins with a set of deadlock states and performs backward reachability, there is a significant overlap among states explored by different threads. Hence, the first option is likely to essentially reduce the parallel run to a sequential run. For this reason, we focus on the second approach where each thread explored the states concurrently. (We also use some heuristic based synchronization where we maintained a set of *visited* states that each thread checked before performing backward state exploration. This technique provided only a small performance benefit.)

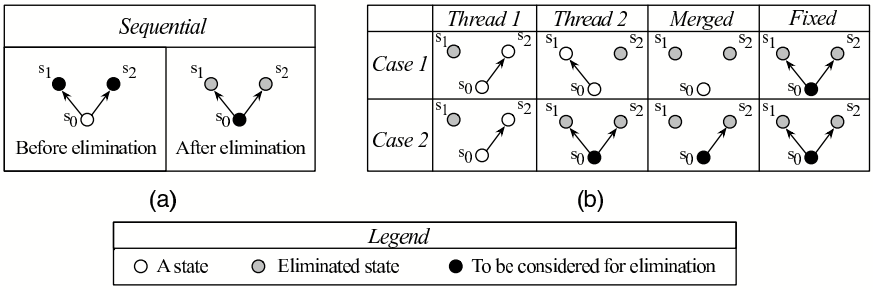


Figure 2: Inconsistencies raised by concurrency.

Inconsistency Resolution. When threads explore states concurrently, some inconsistencies may be created. Next, we give a brief overview of the inconsistencies that may occur due to concurrent state exploration and manipulation by different threads and identify how we can resolve them. Towards this end, let s_1 and s_2 be two states that are considered for deadlock elimination and (s_0, s_1) and (s_0, s_2) be two program transitions for some s_0. A sequential elimination algorithm, removes transitions (s_0, s_1) and (s_0, s_2) which causes s_0 to be a new deadlock state (cf. Figure 2.a). This in turn requires that state s_0 itself must be made unreachable. If s_0 is unreachable then including the transition (s_0, s_1) in the synthesized program is harmless. In fact, it is desirable since including this transition also causes other transitions in the corresponding group to be included as well. And, these grouped transitions might be useful in providing recovery from other states. Hence, it puts (s_0, s_1) and (s_0, s_2) (and corresponding group) back into the program being synthesized and it continues to eliminate the state s_0. However, when multiple worker threads, say th_1 and th_2, run concurrently, some inconsistencies may be created. We describe some of these inconsistencies and our approach to resolve them next.

Case 1. States s_1 and s_2 are in different partitions. Hence, th_1 eliminates s_1 which in turn removes the transition (s_0, s_1), and, th_2 eliminates s_2 which removes the transition (s_0, s_2) (cf. Figure 2.b). Since each thread works on its own copy, neither thread tries to eliminate s_0, as they do not identify s_0 as a deadlock state. Subsequently, when the master thread merges the results returned by th_1 and th_2, s_0 becomes a new deadlock state which has to be eliminated while the group predicates of transitions (s_0, s_1) and (s_0, s_2) have been removed unnecessarily. In order to resolve this case, we re-introduce all outgoing transitions that start from s_0 and mark s_0 as a state that has to be eliminated in subsequent iterations.

Case 2. Due to backtracking behavior of the elimination algorithm, it is possible that th_1 and th_2 consider common states for elimination. In particular, if th_1 considers s_1 and th_2 considers both s_1 and s_2 for elimination (cf. Figure 2.b), after merging the results, no new deadlock states are introduced. However, (s_0, s_1) would be removed unnecessarily. In order to resolve this case, we collect all the states that worker threads failed to eliminate and replace all incoming transitions into those states.

5.2 Experimental Results

We also implemented this approach for parallelization. The results for the problem of Byzantine agreement are as shown in Table 4. From these results, we notice that the improvement in the performance was small.

		Sequential		*Parallel Elimination with 2-threads*	
No. of Processes	**Reachable States**	**Deadlock Resolution Time**	**Total Synthesis Time**	**Deadlock Resolution Time**	**Total Synthesis Time**
10	10^7	7	9	8	9
15	10^{12}	78	85	78	87
20	10^{14}	406	442	374	417
25	10^{18}	1,503	1,632	1,394	1,503
30	10^{21}	4,302	4,606	3,274	3,518
35	10^{25}	11,088	11,821	10,995	11,608
40	10^{28}	27,115	28,628	21,997	23,101
45	10^{32}	45,850	48,283	39,645	41,548

Table 4: The time required to synthesis tolerant program for several numbers of non-general processes of *BA* in sequential and by partitioning deadlock states using parallelism.

6 Related Work

Automated program synthesis and revision has been studied from various perspectives. Inspired by the seminal work by Emerson and Clarke [6], Arora, Attie, and Emerson [2] propose an algorithm for synthesizing fault-tolerant programs from CTL specifications. Their method, however, does not address the issue of the addition of fault-tolerance to existing programs. Kulkarni and Arora [13] introduce enumerative synthesis algorithms for automated addition of fault-tolerance to centralized and distributed programs. In particular, they show that the problem of adding fault-tolerance to distributed programs is NP-complete. In order to remedy the NP-hardness of the synthesis of fault-tolerant distributed programs and overcome the state explosion problem, we proposed a set of symbolic heuristics [4], which allowed us to synthesize programs with a state space size of 10^{30} and beyond.

Ebnenasir [5] presents a divide-and-conquer method for synthesizing *failsafe* fault-tolerant distributed programs. A failsafe program is one that does not need to satisfy its liveness specification in the presence of faults. Thus, a respective synthesis algorithm does not need to resolve deadlock states outside the invariant predicate. Moreover, Ebnenasir's synthesis method resolves deadlock states inside the invariant predicate in a sequential manner.

We have also presented an approach [1] for utilizing multi-core technology in the design of self-stabilizing programs, i.e., a program that ensures that starting from an arbitrary state, it recovers to a legitimate state. This work utilizes parallelization of group computation as well as another approach for expediting the design of stabilizing programs. However, due to the nature of the problem involved, parallelization of group computation is more effective in deadlock resolution than in design of stabilizing programs [1].

Parallelization of symbolic reachability analysis has been studied in the model checking community from different perspectives. In [7, 8, 9], the authors propose solutions and analyze different approaches to parallelization of the *saturation*-based generation of state space in model checking. In particular, in [8], the authors show that in order to gain speedups in saturation-based parallel symbolic verification, one

has to pay a penalty for memory usage of up to 10 times, that of the sequential algorithm. Other efforts range from simple approaches that essentially implement BDDs as two-tiered hash tables [16, 18], to sophisticated approaches relying on *slicing* BDDs [11] and techniques for *workstealing* [10]. However, the resulting implementations show only limited speedups.

7 Conclusion

Summary. In this paper, we focused on improving the synthesis of fault-tolerant programs from their fault-intolerant version. We focused on two approaches for expediting the performance of the synthesis algorithm by using multi-core computing. We showed that the approach of partitioning deadlock states provides a small improvement. And, the approach based on parallelizing the group computation – that is caused by distribution constraints of the program being synthesized– provides a significant benefit that is close to the ideal, i.e., equal to the number of threads used. Moreover, the performance analysis shows that this approach is scalable in that if more cores were available, our approach can utilize them effectively.

Lessons Learnt. As shown in [4], there are two main bottlenecks in synthesizing fault-tolerant programs: *generation of fault-span* which is essentially a reachability problem that has been studied extensively in the context of model checking and *deadlock resolution* that corresponds to adding recovery paths from states reached in the presence of faults. The results in this paper show that a traditional approach (Section 5) of partitioning deadlock states provides a small improvement. However, it helped identify an alternative approach for parallelization that is based on the distribution constraints imposed on the program being synthesized.

The performance improvement with the use of the distribution constraints is significant. In fact, for most cases, the performance was close to the ideal speedup. What this suggests is that for the task of deadlock resolution, a simple approach based on parallelizing the group computation (as opposed to a reentrant BDD package that permits multiple concurrent threads or partition of deadlock states etc.) that is caused due to distribution constraints will provide the biggest benefit in performance. Moreover, the group computation itself occurs in every aspect of synthesis where new transitions have to be added for recovery or existing transitions have to be removed for preventing safety violation or breaking cycles that prevent recovery to the invariant. Hence, the approach of parallelizing the group computation will be effective in the synthesis of distributed programs.

Impact. Automated synthesis has been widely believed to be significantly more complex than automated verification. When we evaluate the complexity of automated synthesis of fault-tolerance, we find that it fundamentally include two parts: (1) analyzing the existing program and (2) transforming it to ensure that it meets the fault-tolerance properties. The first part closely resembles with program verification and techniques for efficient verification are directly applicable to it. What this paper shows is that the complexity of the second part can be significantly remedied by the use of parallelization in a *simple* and *scalable* fashion. Moreover, if we evaluate the typical inexpensive technology that is currently being used or is likely to be available in *near future*, it is expected to be 2-16 core computers. And, the first approach used in this paper is expected to be the most suitable one for utilizing these multicore computers to the fullest extent. Also, since the group computation is caused by distribution constraints of the program being synthesized, as discussed in Section 5, it is guaranteed to be required even with other techniques for expediting automated synthesis. For example, it can be used in conjunction with the approach in Section 5 as well as the approach that utilizes symmetry among processes being synthesized.

References

[1] F. Abujarad & S. S. Kulkarni (2009): *Multicore Constraint-Based Automated Stabilization*. In: *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*.

[2] A. Arora, P. C. Attie & E. A. Emerson (1998): *Synthesis of Fault-Tolerant Concurrent Programs*. In: *Principles of Distributed Computing (PODC)*. pp. 173–182.

[3] A. Arora & S. S. Kulkarni (1998): *Component Based Design of Multitolerant Systems*. *IEEE Transactions on Software Engineering* 24(1), pp. 63–78.

[4] B. Bonakdarpour & S. S. Kulkarni (2007): *Exploiting Symbolic Techniques in Automated Synthesis of Distributed Programs with Large State Space*. In: *IEEE International Conference on Distributed Computing Systems (ICDCS)*. pp. 3–10.

[5] A. Ebnenasir (2007): *DiConic addition of failsafe fault-tolerance*. In: *Automated Software Engineering (ASE)*. pp. 44–53.

[6] E. A. Emerson & E. M. Clarke (1982): *Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons*. *Science of Computer Programming* 2(3), pp. 241–266.

[7] J. Ezekiel & G. Lüttgen (2007): *Measuring and evaluating parallel state-space exploration algorithms*. In: *International Workshop on Parallel and Distributed Methods in Verification (PDMC)*.

[8] J. Ezekiel, G. Lüttgen & G. Ciardo (2007): *Parallelising Symbolic State-Space Generators*. In: *Computer Aided Verification (CAV)*. pp. 268–280.

[9] J. Ezekiel, G. Lüttgen & R. Siminiceanu (2006): *Can Saturation be Parallelised? On the Parallelisation of a Symbolic State-Space Generator*. In: *International Workshop on Parallel and Distributed Methods of Verification (PDMC)*. pp. 331–346.

[10] O. Grumberg, T. Heyman, N. Ifergan & A. Schuster (2005): *Achieving speedups in distributed symbolic reachability analysis through asynchronous computation*. In: *Correct Hardware Design and Verification Methods (CHARME)*. pp. 129–145.

[11] O. Grumberg, T. Heyman & A. Schuster (2006): *A work-efficient distributed algorithm for reachability analysis*. *Formal Methods in System Design (FMSD)* 29(2), pp. 157–175.

[12] Maurice Herlihy (2008): *The Future of Distributed Computing: Renaissance or Reformation?* In: *Twenty-Seventh Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2008)*.

[13] S. S. Kulkarni & A. Arora (2000): *Automating the Addition of Fault-Tolerance*. In: *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*. pp. 82–93.

[14] S. S. Kulkarni & A. Ebnenasir (2005): *Complexity Issues in Automated Synthesis of Failsafe Fault-Tolerance*. *IEEE Transactions on Dependable and Secure Computing* .

[15] L. Lamport, R. Shostak & M. Pease (1982): *The Byzantine Generals Problem*. *ACM Transactions on Programming Languages and Systems* 4(3), pp. 382–401.

[16] K. Milvang-Jensen & A. J. Hu (1998): *BDDNOW: A parallel BDD package*. In: *Formal Methods in Computer Aided Design (FMCAD)*. pp. 501–507.

[17] F. Somenzi. *CUDD: Colorado University Decision Diagram Package*. `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`.

[18] T. Stornetta & F. Brewer (1996): *Implementation of an efficient parallel BDD package*. In: *Design automation (DAC)*. pp. 641–644.