Fault-Tolerant Dynamic Task Graph Scheduling

Mehmet Can Kurt*, Sriram Krishnamoorthy†, Kunal Agrawal‡, and Gagan Agrawal*
* The Ohio State University,
Email: {kurt,agrawal}@cse.ohio-state.edu
†Pacific Northwest National Laboratory,
Email: sriram@pnnl.gov
‡Washington University in St. Louis,
Email: kunal@cse.wustl.edu

***Abstract*—In this paper, we present an approach to fault-tolerant execution of dynamic task graphs scheduled using work stealing. In particular, we focus on selective and localized recovery of tasks in the presence of soft faults. From users, we elicit the basic task graph structure in terms of successor and predecessor relationships. The work-stealing-based algorithm to schedule such a task graph is augmented to enable recovery when the data and metadata associated with a task get corrupted. We use this redundancy, and knowledge of the task graph structure, to selectively recover from faults with low space and time overheads. We show that the fault tolerant design retains the essential properties of the underlying work stealing-based task scheduling algorithm, and that the fault tolerant execution is asymptotically optimal when task re-execution is taken into account. Experimental evaluation demonstrates the low cost of recovery under various fault scenarios.**

***Keywords*—dag, task graphs, cilk, work stealing, fault tolerance**

I. Introduction

Most of the work on fault tolerance in parallel systems has focused on *fail-stop* failures, where a node completely halts. In recent years, there has been growing concern about another class of failures, namely, the *soft errors* or *silent data corruption*. These errors involve bit flips in either the processing cores, the memory, or the disk, and there are several causes for such bit flips in an operational system. Traditionally, although radiation has been considered the cause of such random bit flips [1], the use of smaller and smaller transistors as well as efforts to improve power efficiency in hardware now are attributed as causes of these faults occurring more frequently [2]. Other factors can include packaging materials and voltage fluctuations. Many recent publications have summarized the observed frequency of these faults [3]. For example, double bit flips, which cannot be corrected by Error Correcting Codes (ECC), occur daily at a national lab's Cray XT5, and, similarly, such errors were frequent in BG/L's unprotected L1 cache. While hardware designs have traditionally tolerated many of the soft errors, there is an increasing need for software solutions to this problem.

In this paper, we focus on dealing with *detectable* soft errors in task graphs. With the increasing popularity of large-scale multi-core and many-core machines and a need for parallelizing dynamic or irregular applications, task graph scheduling has emerged as an important problem. A task graph represents tasks as basic units of work and the dependences between the tasks. Task graphs expose greater concurrency than parallelism typically available in the hardware and enable automated scheduling of tasks onto processor cores. Such task scheduling can be performed to satisfy a variety of requirements, although the most common consideration is ensuring load balance among the processing cores and, thus, reduced execution.

We consider the design, implementation, and evaluation of scheduling algorithms that can continue execution of an application specified through a task graph to completion despite faults. The goal of the work is to minimize the slowdown of the application in the presence of soft errors. To the best of our knowledge, this problem has not been addressed in the past. There is a considerable amount of work on fault-tolerant task graph scheduling in the real-time systems community [4], [5], [6]. These efforts require replicated task execution and/or duplicated state to support efficient failover. In comparison, our goal is to minimize the impact of faults without significantly impacting the performance or resource utilization during normal execution. In addition, we also consider the scheduling problem in the context of *dynamic* task graphs [7] with two characteristics. First, the task graphs cannot be fully expanded until they actually are being executed. Second, the task execution times are not known ahead of time. These two characteristics hold true for scientific applications that are likely to be expressed with a task model. At the same time, because of these characteristics, low-overhead runtime approaches are needed not only to schedule tasks, but to reschedule tasks when failures occur. In comparison, *static* task graphs, which are graphs that can be fully processed at the compile time, allow offline scheduling for performance and possibly even for fault tolerance. However, applications that can be expressed using static tasks graphs likely may also be expressed with more structured frameworks, such as OpenMP or its variants.

While both our problem formulation as well as the main ideas in our solution are general and applicable to task graph execution on shared and distributed-memory platforms, the detailed algorithms we have developed and implemented are specific to the NABBIT system. NABBIT [8] is a framework for scheduling task graphs in a provably time-efficient manner using work stealing. In this paper, we adapt the NABBIT dynamic task graph scheduling algorithm to support scalable recovery from soft errors that impact individual tasks. The recovery is performed in a *non-collective* fashion without interfering with threads not impacted by the fault. Simultaneously, the threads requiring a waiting task efficiently perform the

978-1-4799-5500-8/14 $31.00 © 2014 IEEE
DOI 10.1109/SC.2014.64

IEEE computer society

recovery without incurring significant blocked or idle time. The presented approach can recover from an arbitrary number of task failures while incurring very low overheads in the absence of faults.

Our work is significant from both theoretical and practical viewpoints. We prove the fault-tolerant algorithm retains the time-efficiency properties of the NABBIT framework. In particular, in the absence of faults, we get the same bounds as the original NABBIT bounds. In addition, for graphs with small degree, the overhead due to contention is negligible, and the bounds are asymptotically optimal. From the practical side, extensive experimental evaluation involving five real scientific applications on a 48-core machine demonstrates that the overheads of the fault tolerance scheme are low (within system noise) in the absence of faults, and the cost of recovery from failures is proportional to the work lost. For the benchmarks considered, recovery from loss of 5% of work introduces an overhead between 5.1%-8.2%, and failure of a small constant number of tasks (up to 64) can be recovered with no statistically significant overhead.

II. Background and Problem Statement

In our work, a task graph is represented by a directed acyclic graph with vertices representing tasks and edges representing dependences between them, pointing from a source of a dependence to its destination. A task/vertex cannot begin execution until all tasks it depends on, referred to as its *predecessors*, complete. Vertices with no incoming dependences are referred to as *source* vertices, and those with no outgoing dependences are referred to as the *sink* vertices.

The task graph representation in this work encompasses models that represent the dependence between tasks rather than between tasks and data blocks [9], [10]. This representation can be derived from such a bipartitite dependence graph by transitively constructing the task-task dependences. We do not require that each data block be constructed exactly once. Instead, we allow updates to data blocks, as long as the dependences specified ensure that all uses of a data block causally precede a subsequent definition (considered the *next version*) of the same block. Each task is considered synonymous with the definitions of data blocks it effects. A single task can produce multiple data blocks, and two tasks have a dependence between them if any data block defined by one task is used by the other. Figure 1 shows an example task graph with A as the source and E as the sink task. Note that task A's data could be overwritten by task C because all uses of A's output, except by C itself, are complete when task C begins execution.

Our goal is to handle task scheduling in the presence of *soft faults* or errors. A soft fault can result in a bit flip in combinational or sequential logic. This can propagate into an incorrect arithmetic result or memory state. When undetected, this can lead to what is referred to as *silent data corruption*. A soft error affecting a task affects the computation only if the description of the task or any of its outputs is affected. Therefore, we focus on recovery from corruption of data blocks or task descriptors and, more specifically, on recovery from such corruption *once it is detected*. There is significant ongoing research in error detection, which can stem detect errors from a variety of sources: hardware or software error detection

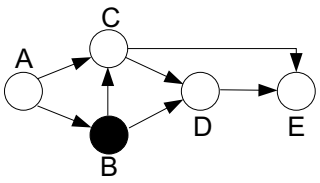


Fig. 1: A task graph example with task A as the source and task E as the sink. Dependence edges are drawn from producer to consumer. Black-shaded task B fails during execution.

codes, such as ECC; symptom-based error detectors [11]; application-level assertions; etc. Sequential execution elements (memory allocator, call stack, etc.), information on the task graph structure embedded in the application, the work stealing runtime, and the application data structures beyond the data blocks operated on by tasks are assumed to be made resilient through other means. We also assume that once an error is detected, all subsequent accesses to that object will observe the error.

In this paper, we are interested in runtime scheduling algorithms for dynamic task graphs that can recover from corruption of task descriptors and data blocks. We consider algorithms that do not require or complement checkpoint-restart and, particularly, do not require using stable storage during task graph processing. Collective recovery approaches, such as those with checkpointing and restart, would synchronize all threads, possibly rolling them back to a prior execution. These approaches will require the overhead of synchronization even when there are no failures, and, with frequent errors, the application's progress may be extremely slow. Our overall objective is to minimize overheads in the absence of faults with recovery costs proportional to the amount of work lost, while ensuring high resource utilization (i.e., not involving unnecessary repeated execution or replication).

For an illustration of the challenges involved, consider a specific snapshot of the execution of the task graph in Figure 1. In the snapshot, task B fails right after its computation, and the failure is detected by the thread operating on task B. Now, even before such a failure is detected, the threads processing tasks C and D could have observed the computation of task B and started their respective computations. Thus, the first challenge is to ensure that the threads executing tasks C and D are aware of the fault in the computation of task B. Then, task B's output has to be recomputed (or *recovered*) before computations of C and D are restarted. For efficiency, it is important that task B recovers only once—not twice—which could occur possibly at the initiation of two different threads executing C and D.

Yet another complication arises because of the *reuse* or *overwriting* of memory that takes place. Recall from our earlier example that task C reuses the space allocated by task A for its output (as the only other use of A's output is by B, which needs to finish before C's execution). Thus, even before C is aware of B's failure, it could be overwriting A's output. However, A's output is required for recomputing B, as well as for restarting the computation of C once B has been recomputed. This implies that A will have to be recovered as well. Finally, similar to the preceding discussion, it is important that A also

```
TRYINITCOMPUTE(A, key, life, pkey)
 1 inserted = INSERTTASKIFABSENT(pkey)
 2 # get task descriptor of A's predecessor (B)
 3 (B, blife) = GETTASK(pkey)
 4 if (inserted)
 5   # B has just been inserted. explore B
 6   spawn INITANDCOMPUTE(B, pkey, blife)
 7 try
 8   if(B.overwritten) throw;
 9   finished = true
10   lock(B)
11   if (B.status <Computed)
12     # B should notify A once computed
13     add A.key{key} to B.notifyArray
14     finished = false
15   unlock(B)
16 catch
17   finished = false
18   RECOVERTASKONCE(pkey, blife)
19 if (finished)
20   # B has already been computed
21   NOTIFYONCE(A, key, pkey, life)
```

```
NOTIFYONCE(A, key, pkey, life)
 1 try
 2   # get index of pkey in the ordered list of preds
 3   ind = CONVERTPREDKEYTOINDEX(key, pkey)
 4   success = ATOMICBITUNSET(A.bitVector, ind)
 5   # notify A only if the vector bit was set
 6   if(success)
 7     val = ATOMICDECANDFETCH(A.join)
 8     # execute A if join counter is zero
 9     if (val==0)COMPUTEANDNOTIFY(A, key, life)
10 catch
11   RECOVERTASKONCE(key, life)

INITANDCOMPUTE(A, key, life)
 1 INIT(A)
 2 # traverse immediate predecessors of A
 3 for pkey ∈ predecessors(A.key{key})
 4   spawn TRYINITCOMPUTE(A, key, life, pkey)
 5 NOTIFYONCE(A, key, key, life)

NOTIFYSUCCESSOR(key, skey)
 1 (S, slife) = GETTASK(skey)
 2 NOTIFYONCE(S, skey, key, slife)
```

```
COMPUTEANDNOTIFY(A, key, life)
 1 try
 2   COMPUTE(A)
 3   A.status = Computed
 4   n = SIZEOF(A.notifyArray)
 5   notified = 0
 6   # notify all successors enqueued in notify array
 7   while (notified <n)
 8     for i ∈ [notified, n)
 9       skey = A.notifyArray[i]
10       spawn NOTIFYSUCCESSOR(key, skey)
11     notified = n
12     lock(A)
13     n = SIZEOF(A.notifyArray)
14     if (notified == n) A.status = Completed
15     unlock(A)
16 catch
17   if(error in A) RECOVERTASKONCE(key, life)
18   else RESETNODE(A, key, life)
```

Fig. 2: Routines used by the fault-tolerant task graph scheduler. Non-shaded portions correspond to the actions of the non-fault-tolerant NABBIT scheduler. Shaded portions indicate additions to the algorithm that make it fault tolerant. "A.key{key}" means that key replaces A.key in the fault-tolerant version. Auxiliary routines used for recovery (in *catch* blocks) are discussed in Section IV.

recovers only once.

Thus, handling such recovery correctly and efficiently for a general dynamic task graph is a significant challenge. This problem is addressed in the rest of the paper, in the context of an existing scheduler used in the NABBIT framework.

III. TASK GRAPH SCHEDULING USING NABBIT

In this section, as a prelude to presenting the scheduling scheme in the presence of failures, we describe the elements of the task graph to be specified by the user and outline the scheduling algorithm in the absence of faults. The fault-tolerant scheduling algorithm relies on the following information from the user about the task graph:

Task key: A unique identifier for each task used to relate different references to the same task without the need for a pre-allocated task object.

Sink task: The task that transitively depends on all other tasks in the task graph.

Predecessors and successors: Functions that return an ordered list of predecessors and successors of a task, given its key. This information is used to reveal the dependences among tasks.

Compute: A function that defines the main operations to be performed by each task.

Once this information is available through library-provided classes, the task graph scheduling algorithm captures the structure of the task graph. The task scheduling algorithm is built on the NABBIT task graph scheduler [8], which is a provably efficient scheduler based on work stealing. Figure 2 presents the routines used by the task scheduler. For this section, we only refer to the non-shaded portions of the routines and ignore the gray shaded parts, which show how the runtime is modified to support failure recovery.

The tasks in the task graph are referred by keys (type int64_t), and their execution is controlled by the runtime through a concurrent hash map. A created task is atomically inserted into the hash map using the INSERTTASKIFABSENT routine and obtained later with a call to GETTASK. Note that the hash map stores the pointers to the tasks and not the tasks themselves.

For each task, the runtime holds the following fields:

(int) join: A counter, referred to as the join counter, that tracks the number of outstanding predecessors for a task. This counter is the basic unit of completion notification. It is initialized with the number of predecessors and decremented whenever a predecessor completes execution. Given that a task can be notified in parallel by any of its predecessors, while it is potentially being operated upon, the join counter is incremented and decremented atomically. A task is ready to be executed when its join counter is zero.

(int64_t*) notifyArray: An array (initially empty) that stores the successors that are enqueued to be notified once the task finishes its execution. Similar to the join counter, mutual exclusion is ensured among concurrent operations on a task's notify array by protecting them with a lock associated with each task.

(int) status: The execution status of a task at the moment. The possible values are Visited, Computed, and Completed. Once a task has been inserted into the hash map, its status is set as Visited.

The execution of a task graph begins with the creation and insertion of the sink task into the hash map followed

by an invocation of the INITANDCOMPUTE function. INITANDCOMPUTE initializes the task and traverses its immediate predecessors through calls to TRYINITCOMPUTE. Note that in general, if the predecessors of a task are all computed, the task can be executed right away using the COMPUTEANDNOTIFY routine. Otherwise, the task registers itself to the notify array of each predecessor that is not ready. Invoking INITANDCOMPUTE and TRYINITCOMPUTE in a recursive fashion, the execution expands the task graph and eventually reaches one of the source tasks with no incoming dependences. COMPUTEANDNOTIFY executes such a task's COMPUTE function (provided by the user), updates its status as Computed, and begins notifying the successors (via NOTIFYSUCCESSOR) registered in its notify array. After there are no more successors left in the notify array, the task changes its status to COMPLETED. Any successor traversing this task after the status change can see the completion of the task's execution and can directly decrement its join counter. If a successor's join counter is observed to be zero, it is scheduled for execution using the same COMPUTEANDNOTIFY routine.

The actions of the task graph scheduler are parallelized using Cilk-style support for recursive parallel (strict) computations. In particular, the creation and computation of the predecessors of a given task are concurrent and can be executed by different threads. The work-stealing scheduler randomly finds work, in terms of segments of task graph traversals to be performed, until the root of the computation—the sink task—is completed.

The actions of individual threads coupled with the scheduling properties of work stealing were used to show that this task graph scheduling algorithm is provably efficient. In particular, given a task graph with work T_1 (the time it takes to execute the task graph on a single processor), critical path length T_∞ (the time it takes to execute on an infinite number of processors), and maximum degree d (maximum number of predecessors and successors), the running time is $O(T_1/P+T_\infty \min\{P, d\})$, where P is the number of available processors. The running time is asymptotically optimal for task graphs where the degree of each node can be bounded by a constant. Even for task graphs with large degree nodes, the running time is close to optimal, *modulo* a small amount of synchronization overhead.

IV. FAULT-TOLERANT SCHEDULING

Figure 2 presents the pseudocode for the fault-tolerant dynamic task graph scheduler. The statements and function parameters shaded in gray are introduced to the NABBIT scheduler to support fault tolerance. These routines are supported by the recovery routines shown in Figure 3. The changes to the NABBIT algorithm are two-fold. First, different phases of the algorithm that access a task object or a data block are enclosed in try-catch statements. Any detected errors are assumed to throw an exception, which is caught to trigger the recovery procedure. Second, additional function parameters and data structures are introduced to support the recovery.

We describe our recovery approach in terms of how the algorithm supports fault recovery by providing key guarantees. These guarantees are then combined to show that the proposed approach can execute a task graph to completion with the correct result irrespective of the number of faults. This descriptive outline of the correctness proof is provided in place of a formal proof due to space constraints. We shall use the example task graph in Figure 1 and execution snapshot described in Section II for illustration.

Guarantee 1. *Each failure is recovered at most once.*

For example, we want task B in Figure 1 to be recovered once and not by every observer of the fault. We associate each creation and insertion of a task object into the hash map with a *life number*. This life number is tracked through the call stack whenever that task is processed. Thus, a failure detected is associated with a particular life number of that task, referred to as its *incarnation*. When multiple threads detect the failure, they try to simultaneously recover that task using the RECOVERTASKONCE routine. As the first step, the ISRECOVERING routine checks whether recovery of the current incarnation of the task (the task with the current life number) already has been initiated. The first thread to observe this condition performs the actual recovery. To facilitate this, we maintain a separate concurrent hash map (denoted as *R* in Figure 3) that associates a given key with the most recent life number for which a recovery process has been initiated. The recovery table is empty initially, and a record for a given key is inserted only if a failure occurs on the corresponding task. When a task fails the first time, the thread that inserts the record into *R* performs the recovery. For subsequent failures on the same task, the thread that succeeds in updating the existing record performs the recovery. Note that a failed task whose successors already have been computed is not recovered, because no other task attempts to access such a task.

Guarantee 2. *A task's status is correctly recovered.*

A task's status dictates the action a thread takes when it is encountered. As indicated before, a task's status could be Visited (created but not computed), Computed (COMPUTE function has been executed), or Completed (all enqueued successors have been notified). Rather than attempt to restore a recovered task's status from a backup or snapshot, we treat a task being recovered as a newly created task and begin its processing by checking the status of its predecessors. This is shown in the RECOVERTASK routine, where the recovering thread calls INITANDCOMPUTE to redo the task's execution. Any successor requesting this task from the hash map (using its key) would get this new incarnation and handle it as they would a non-failed predecessor.

Guarantee 3. *The join counter of a task object is decremented exactly once per predecessor.*

Consider the scenario where task B decremented D's join counter before failure and attempts to do so again after recovery. D's join counter would now be zero, allowing it to be executed, even though its predecessor task C might not have been computed. We avoid these scenarios by associating each join counter with additional information about which predecessors have notified it. In particular, we retain a *bit vector* that tracks if the join counter has been decremented for a particular predecessor in the ordered list of predecessors. This bit vector is initialized to 1 for all bits. Each bit is unset when the corresponding predecessor is observed to have been

```
RECOVERTASKONCE(key, life)
 1 if (!ISRECOVERING(key, life))
 2   RECOVERTASK(key)

ISRECOVERING(key, life)
 1 inserted = INSERTRECORD(R, key, life)
 2 if (inserted) return false
 3 stored = GETRECORD(R, key)
 4 success = ATOMICCOMPANDSWAP(stored, life-
   1, life)
 5 return !success

RESETNODE(A, key, life)
 1 try
 2   A.join = 1 + SIZEOF(A.preds)
 3   SETALLBITS(A.bitVector)
 4   INITANDCOMPUTE(A, key, life)
 5 catch
 6   RECOVERTASKONCE(key, life)
```

```
REINITNOTIFYENTRY(T, key, S, skey, slife)
 1 try
 2   # ignore Computed and Completed tasks
 3   if (S.status ≠ Visited) return
 4   ind = CONVERTPREDKEYTOINDEX(skey,key)
 5   if (S.bitVector[ind] == 1)
 6     lock(T)
 7     # skey's waiting notification from T
 8     add skey to T.notifyArray
 9     unlock(T)
10 catch
11   if (error in S)
12     RECOVERTASKONCE(skey, slife)
13   else
14     throw
```

```
RECOVERTASK(key)
 1 repeat
 2   try
 3     success = true
 4     # insert the new incarnation of the task
 5     (T, life) = REPLACETASK(key)
 6     T.recovery = true
 7     # traverse successors to recreate notify arr.
 8     T.succs = GETSUCCS(key)
 9     for skey ∈ T.succs
10       (S, slife) = GETTASK(key)
11       REINITNOTIFYENTRY(T, key, S, skey,
   slife)
12     spawn INITANDCOMPUTE(T, key, life)
13   catch
14     if (!ISRECOVERING(key, life))
15       success = false
16 until (success)
```

Fig. 3: Additional routines to assist in the recovery from faults.

computed, or when that predecessor issues a notification to this task. The join counter is decremented only if that bit is set. This change to NABBIT is shown in NOTIFYONCE in Figure 2. The CONVERTPREDKEYTOINDEX routine, given a predecessor key, returns the predecessor's index in the ordered list of predecessors, so that the corresponding bit in the vector can be unset.

Guarantee 4. *Every task waiting on a predecessor is notified.*

Successors enqueued in the notify array of a failed task are expected to be notified by this task when it computes. Not notifying any such task can result in some tasks never being executed, leading to a hung execution state. In the example, task B needs to correctly handle the notification to tasks C and D. Recovering this information (mainly, the notify array) using some form of duplicated storage would require additional support to recover from multiple failures. We work around this issue by altering the base task graph scheduling algorithm's behavior in the presence of failures. This is shown in REINITNOTIFYENTRY. A task being recovered traverses all of its successors with Visited status to check their bit vector and observe if they have been notified. Any successor not yet notified is considered to have been enqueued for notification before the failure and is enqueued into the reconstructed notify array. When this task computes, all such enqueued tasks are notified. This ensures that tasks C and D are notified when task B has been recovered and recomputed. This *eager* notification deviates from the NABBIT algorithm and could potentially result in execution of tasks in a different, albeit still correct, order impacting its optimality guarantees. We prove in Section V that this change does not violate NABBIT's asymptotic parallel efficiency guarantee.

Guarantee 5. *Failures in data blocks observed during task computation are recovered.*

Note that faults in the data blocks could also be observed in application code, denoted by the COMPUTE routine in COMPUTEANDNOTIFY, when a thread is executing the COMPUTE function to perform the actual computation represented by the task. We assume that these detected errors are reported back to the runtime through exceptions. When an exception is caught in the COMPUTEANDNOTIFY routine, we first identify which task's fault resulted in the failure. Then, we check if the task being computed is the source of the error. If so, that task is recovered. If not, we begin processing the task anew, using the RESETNODE routine, by atomically resetting the notification bit vector and join counter and traversing its predecessors to verify if any of them have failed. As just described, the bit corresponding to each predecessor is unset upon notification, and the join counter is decremented exactly once per predecessor.

Guarantee 6. *Failures observed during recovery are recursively recovered.*

We would like to recover from failures that might affect tasks while they are being recovered. As shown in the RECOVERTASK routine, the recovery of a failed task involves updating its entry in the hash map with a new incarnation of the task using the REPLACETASK routine, initializing the notify array by inspecting its successors, and re-executing the task as if it were a normal task. If an error occurs when such a re-execution is being performed, the next incarnation of the task is inserted into the hash map, and the re-execution begins anew. Such errors can happen an arbitrary number of times and still be recovered. In the algorithm, this can be observed by the fact that operations in the recovery routines in Figure 3 are themselves enclosed in try-catch statements.

Lemma 1. *Every task is executed only after all of its predecessors have been computed, and the final output of every task is computed from the same inputs with and without failures.*

Proof Sketch: Each predecessor decrements a task's join counter exactly once. When the join counter becomes zero and the task is ready to be computed, all of its predecessors have been computed. If a task observes a failure in the task descriptors or data associated with one of its predecessors, it attempts to identify and recover from that failure. This continues until all predecessors execute without faults, and the resulting data blocks are fault-free. Fault-free execution of predecessors produces the same inputs to a task even if the

predecessors had experienced prior failures. ■

Lemma 2. *A task whose predecessors fail is eventually executed.*

Proof Sketch: A task might observe the failure of one of its predecessors during its computation or when it is checking its predecessors. Any such failed predecessor ensures that it notifies this task exactly once. Note that a task may have been notified by an earlier incarnation of a predecessor before it failed. If the task begins its user computation while the failed predecessor is being recovered, it would reset its bit vector, traverse the predecessors one more time, and observe this failure. This task either observes the recovered state of its predecessor and is computed, or is eventually notified by any outstanding predecessors, including the predecessor being recovered. The last predecessor to notify executes this task. ■

Lemma 3. *The sink task is executed to completion irrespective of the number of failures.*

Proof Sketch: The predecessors of a sink task might observe a failure in themselves, or in one of their predecessors, during their computation. All of these failures can be recovered from, as explained in Lemma 2. Thus, all predecessors of the sink task execute to completion. If the sink task itself fails, it is observed by the thread processing the sink task or by one of its predecessors. As such, failures in the sink task or its predecessors are recovered. When the sink task and all of its predecessors are in an error-free state, the sink task is executed. ■

Theorem 1. *The task graph execution produces the same result with and without faults.*

Proof Sketch: Lemma 3 showed that the sink task is executed irrespective of the number of failures encountered. All tasks are assumed to be stateless, meaning every execution of a task produces the same output for the same inputs. Lemma 1 shows that the sink task is executed with the correct non-faulty inputs, producing the same final result with and without faults. ■

When data blocks are reused by multiple tasks in a graph, additional re-execution may be required. For example, task A's output has been partially overwritten by task C. However, re-execution of task B (and later task C) requires the output from A. During normal execution, the dependences specified ensure that all uses of a particular *version* of a data block are complete before the task that produces the next version of the data block is allowed to execute. However, a fault might result in the need to use such a data block version after it has been overwritten. Our algorithm tracks such overwrites and re-executes these tasks by treating them as if they failed.

The fault-tolerant algorithm does not affect the performance of the base task graph scheduler in the absence of faults, except through the addition of atomic operations to maintain the bit vector. The recovery focuses on tasks that failed and is performed by threads that need to operate on them without affecting other threads. While single-assignment task graphs only incur these costs, reuse of data buffers could result in additional re-execution in trying to reproduce the inputs to the failed tasks. In the experimental evaluation, we show that this overhead is not significant in practice, and could be ameliorated by retaining the intermediate versions in memory. While errors can still affect these retained versions, the impact of such re-execution also can be minimized.

V. Performance Analysis

In this section, we provide a theoretical analysis of the runtime of the program on P processors in the presence of faults. As there is failure recovery built into the system, the running time depends on the number of times each node fails and is recovered. In particular, we prove an *a posteriori* bound—our bound depends on the number of times each node failed and was recovered. We first calculate the upper bounds on the total work and span of any execution. Then, we translate these bounds to completion time bounds using known theoretical bounds on completion times of series-parallel programs using randomized work stealing [12], [13].

A. Definitions

Consider a task graph $\mathcal{D} = (V, E)$. Each node $A \in V$ has a list $\texttt{in}(A)$ of immediate predecessors and a list $\texttt{out}(A)$ of immediate successors. Therefore, $|\texttt{out}(A)|$ and $|\texttt{in}(A)|$ denote the out- and in-degrees of A, respectively. For simplicity in stating the results, we assume that every node is a successor of a unique node $\texttt{root}(\mathcal{D})$ with no incoming edges and a predecessor of a unique node $\texttt{final}(\mathcal{D})$ with no outgoing edges. Let $paths(A, B)$ be the set of all paths in $\mathcal{D}$ from node A to node B.

Note that actual execution of the computation is nondeterministic for several reasons. A node may be executed more than once due to failures or because the memory containing the result has been freed since the node executed and another successor needs this result. In addition, the actual execution of a dynamic task graph may depend on the input and the schedule. Hence, our completion time bounds will be for a particular execution. Each possible execution can be represented as an execution graph (more specifically, DAG) $\mathcal{E}$. Each execution of $\mathcal{D}$ leads to a potentially different $\mathcal{E}$.

We define several notations for subgraphs of an execution graph $\mathcal{E}$. For a particular execution graph $\mathcal{E}$, we define a function N such that $N(\mathcal{E}, A)$ is the number of times task A is executed in $\mathcal{E}$. In addition, let $\texttt{com}_i(\mathcal{E}, A)$ be the subgraph corresponding to ith execution of the compute function and $\texttt{comNot}_i(\mathcal{E}, A)$ be the subgraph corresponding to the ith execution of the compute and notify function. For any subgraph $\mathcal{E}'$ of an execution DAG, we denote the work of the subgraph as $W(\mathcal{E}')$ and the span as $S(\mathcal{E}')$.

To analyze the running time, we must examine executions of the worst-case parameters of $\mathcal{E}$. We define that the total work done by an execution $\mathcal{E}$ of $\mathcal{D}$ is $W(\mathcal{E})$, and the span is $S(\mathcal{E})$. We will overload the notation, indicating that $W(\mathcal{D}^N)$ is the maximum work among all execution graphs that have the same function N, and $S(\mathcal{D}^N)$ is the maximum span among all of these execution graphs. Similarly, for each node A, we define $W(\texttt{com}(A))$ as the maximum time a compute function can take over all execution graphs $\mathcal{E}$ and all executions of that compute function. Similarly, $S(\texttt{com}(A))$ is the maximum span of the compute function of A over all execution possibilities of A.

B. Work Analysis

To calculate the work of a task graph execution, we first construct (pessimistic) bounds on the time NABBIT spends waiting at synchronization operations. Let $L_W(A)$ be the maximum over all executions of the time spent on atomic decrements.

Lemma 4. *Any execution of $\mathcal{D}$ has work at most* $W(\mathcal{D}^N) = O\Big(\sum_{A\in V} N(A)\big(W(\texttt{com}(A)) + \sum_{B\in \texttt{out}(A)} N(B) + L_N(A)\big) + L_J(A)\Big)$, *where*

$$L_J(A) = O\left(\sum_{B\in \texttt{out}(A)} \min\{|\texttt{in}(B)|, P\}\right) \text{ and}$$
$$L_N(A) = O\left(\sum_{C\in \texttt{in}(A)} \min\{|\texttt{in}(C)|, P\}\right).$$

Proof: The first term arises from the work of the compute functions because each node is executed $N(A)$ times. The second term arises from the fact that every time A is executed, we must look through the notify array to see if any of them should be notified. Note that each successor B of A can appear many times in the notify array, in particular, $N(B)$ times. The term L_N stems from the fact that a node may be added many times to its predecessor's notify array, and there is potential contention on this array. Finally, the term L_J bounds the amount of time we can spend waiting to decrement the join counter. Every time a node B is notified and its join counter is decremented, the decrement may have to wait for other updates. We do not multiply this quantity by $N(A)$ as the failure model ensures that each node is notified at most once. ∎

C. Span Analysis

The nondeterministic nature of the computation complicates a direct calculation of $S(\mathcal{D}^N)$. Instead, we construct a new, deterministic execution DAG $\mathcal{E}^N$, which is parametrized by the function N. The span of this DAG is an upper bound on the span of the computation where the number of times each node is executed is dictated by the function N.

In this DAG, for a node A, there are $N(A)$ executions of the compute function represented by DAGs $\texttt{com}_1(A), \texttt{com}_2(A), ..., \texttt{com}_{N(A)}(A)$. We omit $\mathcal{E}$ from this notation and assume that each of these DAGs is the worst-case DAG for the compute function over all executions. At the end of each compute function, we assume there is a notification to all of the successors, but none of these notifications actually succeed. Finally, in the graph $\mathcal{E}^N$, we add edge from the end of DAG $\texttt{com}_i(A)$ to $\texttt{com}_{i+1}(A)$ as these computations must occur sequentially one after the other.

More importantly, in $\mathcal{E}^N$, we make two assumptions. First, for each node, we define the method $\texttt{comNot}^*(A)$ to be the same as the original method, except that all possible recursive calls always occur. In other words, $\texttt{comNot}^*(A)$ always makes recursive calls for all of its successors. Second, we assume that only the last execution of each node A actually manages to inform all of the successors. In other words, the last call to $\texttt{com}(A)$ is the one that turns into $\texttt{comNot}^*(A)$. Therefore, the first instance of $\texttt{com}_1(B)$ of B has an edge from the last instance of $\texttt{com}_{N(A)}(A)$ of A if A is a parent of B.

Lemma 5. *With a dynamic execution graph $\mathcal{E}$ generated by $\mathcal{D}$, where a node A is executed $N(A)$ times, the span of $\mathcal{E}$ is at most $S(\mathcal{E}^N)$.*

Proof Sketch: Consider the longest path to any subDAG $\texttt{com}_1(A)$ in $\mathcal{E}$ and in $\mathcal{E}^N$. Both paths must contain at least one instance of $\texttt{comNot}(B)$ for every B that precedes A. However, it is easy to see that the path in $\mathcal{E}^N$ is not shorter because everyone is always successfully notified by the last execution of every node. ∎

Lemma 6. *The span of the computation is $S(\mathcal{E}^N)$ is at most*

$$S(\mathcal{E}^N) \le O\Big(\max_{p\in \textit{paths}(\texttt{root},\texttt{final})} \Big\{\sum_{X\in p} N(X)\big(S(\texttt{com}(X)) + \sum_{Y\in \texttt{out}(A)} N(Y) + L_N(X)\big) + \sum_{(X,Y)\in p} L_S(X,Y)\Big\}\Big),$$

where

$$L_S(X,Y) = O\left(\min\{|\texttt{in}(Y)|, P\}\right) \text{ and}$$
$$L_N(A) = O\left(\sum_{C\in \texttt{in}(A)} \min\{|\texttt{in}(C)|, P\}\right).$$

Proof sketch: The first term is due to the fact that node X is executed $N(X)$ times (one after the other), and each execution has the span $S(\texttt{com}(X))$. The second term comes from the fact that each of these compute functions tries to notify all of the nodes in the notify array but fails to do so. The term L_N is due to the contention on the notify array. Finally, the last term stems from the fact the last execution of X successfully notifies all of X's successors. The term $L_S(X)$ accounts for the contention cost of decrementing the join counter for Y, where Y is a descendant of X. In the worst case, this decrement might have to wait for $\min\{|\texttt{in}(Y)|, P\}$ other decrements. ∎

D. Completion Time Bounds

We have bounded the work and span of the execution graph using the characteristics of the task graph. Now, we relate these bounds back to the execution time using a work-stealing scheduler.

For any task graph $\mathcal{D}$, where a node A is executed $N(A)$ times according to the failure model, define T_1 as the time it takes to execute $\mathcal{D}$ on a single processor. Define T_∞ as the time it takes to execute $\mathcal{D}$ on an infinite number of processors, assuming no synchronization overhead. We have

$$T_1 = \sum_{A\in V} N(A)(W(\texttt{com}(A)) + |\texttt{out}(A)|)$$

and

$$T_\infty = \max_{p\in \textit{paths}(\texttt{root},\texttt{final})} \Big\{\sum_{X\in p} N(A)S(\texttt{com}(X))\Big\}.$$

Using Lemmas 4 and 6, and the analysis of a work-stealing scheduler [12], [13], we obtain the following upper bound for the completion time of the task graph on P processors:

Theorem 2. *Consider a task graph $\mathcal{D}$, where each node A is executed $N(A)$ times. The graph has the maximum degree d and maximum path length (number of nodes on the longest path in the task graph from* `root` *to* `final`*) M. Also, $\mathcal{N} = \max_{A\in\mathcal{D}} N(A)$. With probability at least $1-\epsilon$,* NABBIT *executes $\mathcal{D}$ on P in time*

$$O\left(T_1/P + T_\infty + \lg(P/\epsilon) + \mathcal{N}Md + \mathcal{N}L(\mathcal{D})\right),$$

where $L(\mathcal{D}) = O\left((|E|/P + M)\min\{d, P\}\right)$.

	LCS	LU	Cholesky	FW	SW
N	512Kx512K	10Kx10K	10Kx10K	5Kx5K	6Kx6K
B	2Kx2K	128x128	128x128	128x128	128x128
T	65536	173880	88560	64000	132650
E	195585	508760	255960	308880	262600
S	510	238	238	120	1475

TABLE I: Matrix size (N), block size (B), total number of tasks (T), total number of dependencies (E) and the critical path length (S) for each benchmark.

Proof sketch: From [12], [13], a Cilk-like work-stealing scheduler completes a computation with work W and span S in time $O(W/P + S + \lg(P/\epsilon))$ on P processors with probability at least $1-\epsilon$. To prove the completion time, we relate the work $W(\mathcal{E}^N)$ and span $S(\mathcal{E}^N)$ to T_1 and T_∞.

The proof follows from Lemmas 4 and 6. We bound the in- and out-degrees of nodes by d and bound expressions that compute maximum over paths p in terms of M. Bounding the work in Lemma 4 using the maximum degree, we know that

$$W(\mathcal{E}^N) = T_1 + \mathcal{N}|E| \min\{d_i, P\} \ .$$

Similarly, we can use Lemma 6 to show that $S(\mathcal{E}^N)$ is not more than

$$O\left(T_\infty + \mathcal{N}Md_o + \mathcal{N}M \min\{d_i, P\}\right) \ . \qquad \blacksquare$$

We note a few things about this bound. It reduces to the normal NABBIT bound when there are no failures and $N(A) = 1$ for all A. Second, it is asymptotically optimal for constant degree graphs, graphs whose degree can be bounded by a constant. Even when the degree is not bounded, we do not expect the terms that depend on the degree to have much impact on the runtime all nodes have sufficient work.

VI. EXPERIMENTS

We evaluated our scheduling algorithm on a 48-core Redhat 4.1.2-54 Linux system, consisting of four sockets, each with a 12-core AMD Opteron 2.3 GHz processor, and 256 GB of memory. Our implementation is built on top of Cilk++ 8503 x86_64 release, and all benchmarks are compiled with Cilk++ compiler (based on gcc 4.2.4) with "-O3" optimization. We attempted to mitigate the impact of system noise and variability by using only 44 of the 48 cores for our experiments and setting the affinity of each thread to a specific core. In all experiments reported in this section, we take 10 runs and report the average (arithmetic mean). Standard deviations are presented as error bars.

We evaluated our implementation using five benchmarks: LCS (longest common subsequence); Smith-Waterman (local sequence alignment algorithm [14]); Floyd-Warshall (all-pairs shortest path algorithm in a weighted graph); and two dense linear algebra kernels, which are LU decomposition and Cholesky factorization. The configuration used for each benchmark is shown in Table I. LCS, Smith-Waterman (SW), and Floyd-Warshall (FW) are implemented using the recursive definitions of the corresponding dynamic programming solutions. We evaluated single-assignment and memory reuse strategies for implementing these benchmarks. In the memory reuse version, we allocate a set of data blocks and reused them to store the outputs of subsequent tasks whenever possible. In all cases, except LCS, the memory reuse implementation resulted in improved performance. Thus, we used this version for subsequent evaluation. Note that memory reuse increases the potential cost of our algorithm by requiring recomputation of tasks because their outputs have been overwritten. We expect the overheads of our fault tolerance scheme for the single-assignment implementations to be lower. Memory reuse is not applicable to LCS because each task's output is part of the computation's final output and cannot be reused.

We observed that the cost of fault recovery for Floyd-Warshall significantly depended on the exact location of the fault. This is due to the impact of varying costs incurred by the recursive recomputation of inputs to a failed task. We adapted the implementation to retain two versions per data block, doubling the memory requirement to minimize the impact of such cascading recomputation. Note that neither version of a data block is considered checkpointed as either, or potentially both, could fail. Our resilience approach would be equally applicable to the single-assignment approach for implementing Floyd-Warshall and might be preferable when optimized memory management policies make that implementation comparable or better performing than the two-version implementation. In general, such choices point to the trade-offs between different task graph representations for a given application rather than the scheduling algorithm.

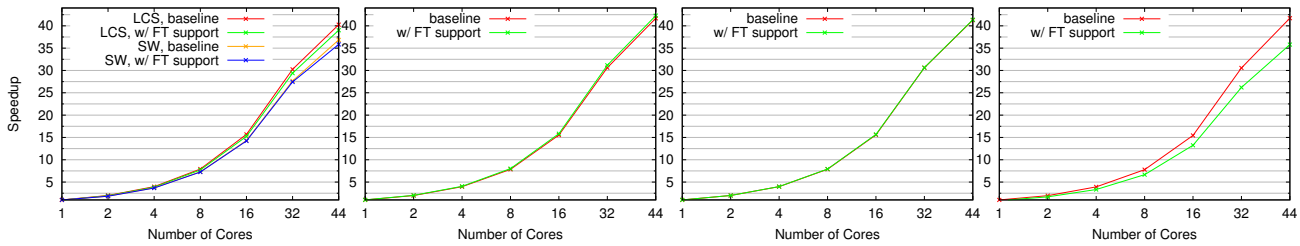
A. Overheads Without Failures

We evaluated the impact of fault tolerance support on performance in the absence of failures. Figure 4 shows speedups achieved by the implementation with no fault tolerance support (`baseline`) and by our fault-tolerant implementation for each benchmark. Unlike the fault-tolerant version, the baseline version includes no additional data structures or statements introduced for fault tolerance. Results indicate that these additional structures do not incur substantial overheads. One exception is Floyd-Warshall, with nearly 10% overhead on the execution with 44 cores. This is due to the memory management and additional cache misses introduced by the two versions maintained for each data block.

B. Overheads With Failures

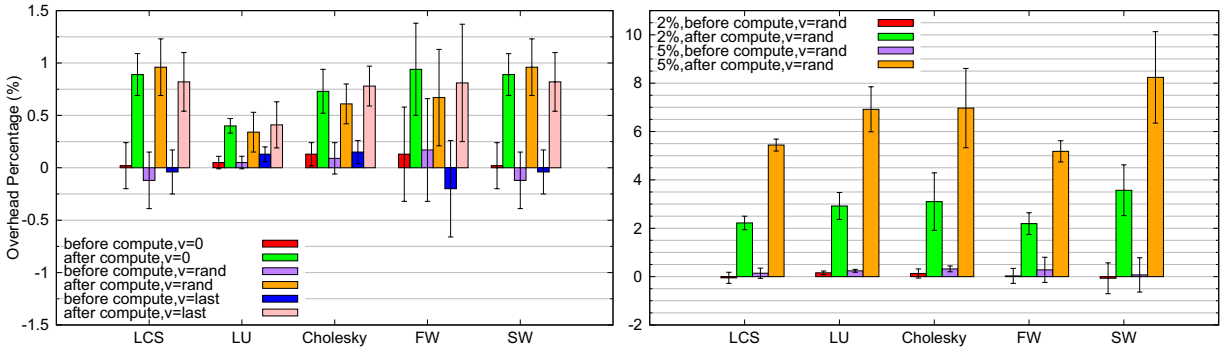
We evaluate overheads in the presence of failures by injecting faults into execution. For each figure in this section, *recovery overhead* is defined as the increase in the execution time (over the fault-tolerant version in the absence of failures) and displayed as a percentage. To simulate faults, we *a priori* identify the tasks that would fail and the point in their lifetimes where they would fail. When a fault is injected, a flag is set to mark the fault, which is then observed by a thread accessing that task. We evaluate the following fault scenarios:

Amount of work lost: We randomly inject failures in the task graph to effect the loss of a constant amount of work or a certain percentage of the total work. A fault affects both a task and the data blocks it has computed. For every experiment, we verify the fault injection by ensuring that the number of tasks recovered matches the loss of work in terms of number of tasks, intended. This cannot be guaranteed in some scenarios (descriptions follow).



(a) LCS, SW (b) LU (c) Cholesky (d) FW

Fig. 4: Speedup for the baseline and fault-tolerant versions in the absence of faults. Sequential times (in secs) for each benchmark are as follows: LCS (baseline=650, w/FT-support=668), SW (baseline=578, w/FT-support=579), LU (baseline=624, w/FT-support=613), Cholesky (baseline=338, w/FT-support=337), and FW (baseline=315, w/FT-support=371).



(a) 512 (b) 2% and 5%

Fig. 5: Impact of the time of failure ("before compute" and "after compute") on the recovery overhead, when failures occur on different task types ("v=0," "v=rand," and "v=last"). **(a)** Failures cause the re-execution of 512 tasks; **(b)** failures cause 2% and 5% of the total number of tasks to be re-executed. In both **(a)** and **(b)**, overheads are calculated by taking the sequential execution time of FT-support version, in the absence of faults, as the baseline. FT-support (evaluated in Figure 4) introduces an additional 10% overhead for FW, and negligible overhead for the other benchmarks.

Time: The cost incurred by a fault depends on the point in a task's lifetime at which the fault affects it. A task lifetime consists of three phases: *before compute*, *after compute*, and *after notify*. A task failing in the *before compute* phase has traversed its predecessors and is waiting for one or more notifications to be scheduled for execution. A task that has completed its main operations and is about to notify its successor tasks, is considered to be in the *after compute* phase. Once a task finishes notifying its successors, it transitions into the *after notify* period. A task may spend a significantly larger fraction of its lifetime in *before compute* and *after notify* phases as these phases could potentially involve several recursive task computations.

The impact of a fault on a task is different in each of these scenarios. A task in the *before compute* phase has not performed its computation, incurring a computation cost. Recovering such a task, while incurring the cost of resetting its state and tracking its predecessors, does not result in task re-execution overhead. In contrast, failures occurring in *after compute* require the task to be re-executed, potentially incurring significant overhead. Statically analyzing the impact of faults in the *after notify* phase is difficult due to fact that some tasks might not recover if all successors of a failed task finish their computation before the fault can be injected.

Task type: We define three sets of tasks for the injection of failures: *v=0*, *v=last*, and *v=rand*. Tasks with the *v=0* label denote tasks that produce the first version of a data block. The *v=last* label denotes tasks that produces the last version. A failure on a *v=0* task causes either 0 or 1 task re-execution, depending on the time at which it is injected. Conversely, the failure of a *v=last* task can trigger a chain of re-executions, where all of the tasks that produce the previous versions of a particular data block get re-executed. Representing a case between these two extremes, we identify a *v=rand* task as a task that produces a random i^{th} version ($0 \leq i \leq n$) of a data block, where n is the last version number of the same data block.

We show the impact of the timing of failure (*before compute* and *after compute*) when we inject failures on three

	LCS			LU			Cholesky			FW			SW		
	0	last	rand	0	last	rand	0	last	rand	0	last	rand	0	last	rand
Avg	443	448	442	473	3606	470	389	5020	483	307	213	508	318	2955	209
Min	431	437	431	448	1976	185	350	3954	232	247	0	333	259	1890	76
Max	453	464	452	484	5385	1292	413	5483	966	419	551	880	361	4397	401
Std	8	9	8	10	1206	360	19	483	282	48	198	192	37	828	97

TABLE II: Average, minimum, maximum, and standard deviation of the number of re-executed tasks in *after notify* scenario when the original set of failures on different task types ("v=0," "v=last," and "v=rand") implies 512 task re-executions. The resulting average recovery overhead for each benchmark is presented in Figure 6.

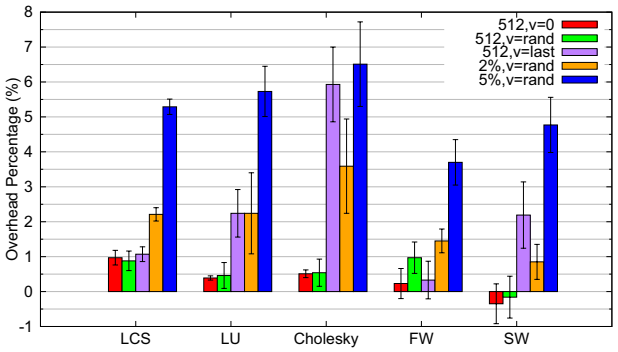


Fig. 6: Recovery overheads in the *after notify* scenario when the set of injected failures implies re-execution of 512 tasks, 2% and 5% of the total number of tasks in the graph.

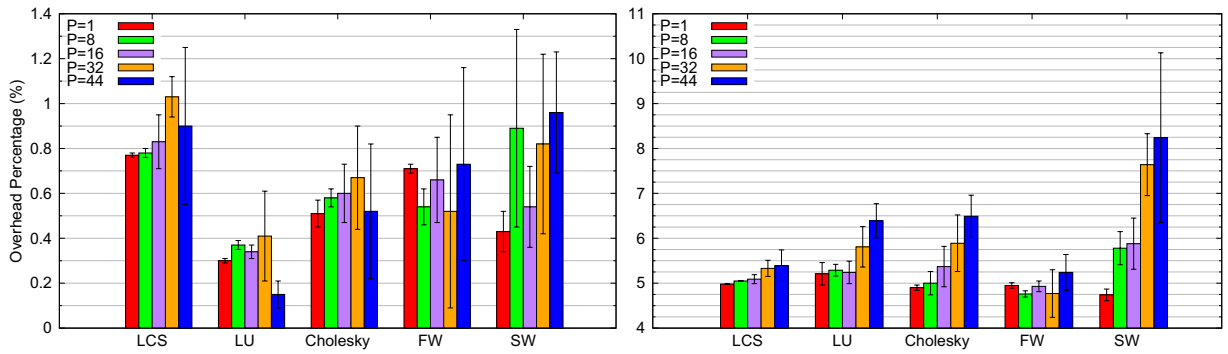
different task types ("v=0," "v=rand," and "v=last") in Figure 5(a). In each of the six injection scenarios, the injected failures cause the recovery to execute a total of 512 tasks corresponding to less than 1% of all tasks for each application, with the largest observed overhead being 0.96%. As expected, the *before compute* cases incur negligible overheads as recovering such failures does not lose any work done prior to the failure. In comparison, *after compute* scenarios lead to loss of computed work, bringing an observable but still small overhead on all benchmarks. Moreover, we observe there is no clear difference in the impact of failures between the *before compute* and *after compute* scenarios, for the various task types. We repeated the same experiment for scenarios with only 1, 8, and 64 task re-executions and did not observe any statistically significant overheads. The relevant figures are not displayed due to space constraints.

Figure 5(b) demonstrates results for the same setup, but with injected failures causing 2% and 5% of the total number of tasks to be re-executed. For this experiment, we report the results for "v=rand" tasks only because the available amount of "v=0" and "v=last" tasks in most of our benchmarks are below 5%. As in the previous figure, *before compute* failures hardly bring any overhead, whereas *after compute* failures cause, at most, 3.6% and 8.2% overheads for the "2%" and "5%" cases, respectively. Broadly, the amount of re-execution overhead is proportional to the amount of work lost.

Next, we evaluate the impact of faults injected in the *after notify* phase for different types of tasks. We present these results separately because the impact of *after notify* faults depends on the specific benchmark and types of faults introduced. In one extreme case, all of the successor tasks may have already used the failing predecessor's output by the time the failure is injected, implying no re-executed work. As another extreme, if the successor discovers the predecessor's failure after it has started to overwrite the previous versions of its output data block, separate recoveries must be initiated to restore both the failing predecessor's output and the previous versions of the successor's output. Table II reports the statistics for the actual number of re-executed tasks (average, min, max, and standard deviation) for each benchmark when the initial set of failures injected at the *after notify* period implies 512 task re-executions. Results show that *after notify* failures on different task types can have a distinct impact on each benchmark, especially in benchmarks where the number of uses on the last version of data blocks depends on the problem size (as in LU, Cholesky, and SW). Such failures can lead to a high number of re-executions with large standard deviation values. In contrast, applications such as LCS, where each data block has, at most, three uses, the re-execution amounts are low and similar for all task types. Figure 6 demonstrates the average overheads for the setup in Table II along with the scenarios where the injected failures imply re-execution of 2% and 5% of the total number of tasks (on "v=rand" tasks only). For most cases, the overheads do not exceed 2.5% and 6.5% for "2%" and "5%" scenarios, respectively.

C. Scalability Analysis

For this analysis, we focus on failures that occur at the *after compute* time period on "v=rand" tasks. Figure 7 plots the scalability of the fault tolerance mechanism when varying the number of cores are used. As Figure 7(a) shows, a constant number of re-executions hardly has any effect on the overall execution, whereas, in Figure 7(b), we observe an increasing trend in the recovery costs when more cores are employed. Generally, when a data reuse strategy is applied for subsequent versions of data blocks, each failure in the system leads to a chain of task re-executions, which cannot be run in parallel. The chain involves the producer task for the corrupted data block, as well as any tasks that produce a previous version for the same data block. These chains lack concurrency to keep all threads busy and comprise the biggest scalability challenge for any task graph execution scheme and related fault tolerance mechanisms. Unsurprisingly, this loss in available concurrency is more visible with a higher number of cores, as shown by Figure 7(b). Nevertheless, the overhead of our fault tolerance mechanism still does not exceed 6.5% for most cases, reaching a maximum of 8.2% in Smith-Waterman on 44 cores.



(a) 512 (b) 5%

Fig. 7: Recovery overheads when varying the number of cores (P) for a failure scenario, causing the re-execution of **(a)** 512 tasks and **(b)** 5% of all the tasks. In both **(a)** and **(b)**, failures are injected on "v=rand" tasks at the *after compute* failure point.

VII. Related Work

The challenges of dealing with an architecture that can expose *soft errors* to programmers has received some attention in recent years. The area of *approximate computation* has emerged [15], [16], [17], [18], [19], [20], [21], with the idea being that a programmer can explicitly declare what computations can be performed approximately. The target domains have been multimedia, data mining, and machine learning. In comparison, scientific programmers are typically unwilling to accept a lack of precision unless supported by rigorous mathematical analysis. Erez et al. have experimented with the notion of *containment domains* [22], which can confine errors in certain program segments. This approach also requires additional programmer effort. Another direction has been to use replication of processes [3], [23]. While this approach does not require additional programmer effort, it decreases resource utilization efficiency, increasing costs and power budget. Our solution does not require additional programmer effort, and has much lower costs compared to replication. However, it does assume that soft errors can be detected in a timely fashion. In other efforts, Li et al. [24] recently studied soft error vulnerabilities in scientific programs, but their work does not provide a specific solution for addressing the problem.

Algorithm-level solutions for fault tolerance: Algorithm-level fault tolerance solutions have been a topic of investigation for almost three decades. In this approach, the idea is that recovery can be performed by using already existing information in running processes, and, if the algorithm itself does not contain such redundant information, it can be added by modifying them [25], [26], [27], [28], [29]. While this approach can handle most types of soft errors, it is specific to a particular algorithm, and cannot be applied to all algorithms.

Task graph scheduling with knowledge of computation and communication times: Many task graph scheduling approaches assume the structure of the task graph and weights associated with the vertices and edges, corresponding to task computation and inter-task communication times, are known in advance [30]. Fault tolerance approaches under these assumptions include both entirely offline approaches [31], [32], [33], [4] and approaches that include some dynamic decision making [34], [5]. These approaches typically employ task duplication, by a fixed or variable amount, to tolerate failures of individual tasks [4], [6] while minimally impacting the task graph execution latency. Given our focus on minimizing fault tolerance overheads rather than minimizing execution latency, we do not assume such complete *a priori* knowledge and do not employ task duplication.

Fault tolerance for task-based computations: Maehle and Markus [35] presented fault-tolerant scheduling of data flow programs on distributed systems by preserving the inputs to tasks so a task can be recovered from a failure using its inputs. In general, recovery from multiple failures requires multiple checkpoints. Vrvilo et al. [36] consider fault tolerance for Concurrent Collections using checkpoints of the current execution frontier of the data flow graph. Using single-assignment data allows checkpointing to be performed in parallel with computation once the data has been created. Charm++ supports message-driven execution and employs message logging and checkpointing to recover from faults [23], [37]. Our approach complements these schemes and can increase the time between checkpoints in computations that can be structured as task graphs.

Fault-tolerant key-value store: Key-value stores have been made resilient against a variety of errors. Dynamo [38]; MapReduce [39], which operates on key-value pairs; and Linda [40], [41], which operates on tuples, provide fault tolerance through replication. These approaches typically store significant application state in the key-value store, requiring expensive techniques to ensure fault tolerance. In our scheme, the values in the hash map are individual scalars that track tasks and can be made resilient with low overhead. We observed that application data blocks and computation dominate the total time (>96% in most cases) and space consumed during execution. Thus, the hash map can be made resilient with minimal impact on overall application performance.

We are not aware of any prior work that combines provably efficient task scheduling and scalable error recovery.

VIII. Conclusions

We presented a fault-tolerant dynamic task graph scheduling algorithm that recovers from faults without global coordination and can efficiently interleave recovery from faults and normal execution to avoid scalability limitations. The algorithm was shown to be asymptotically optimal for graphs whose degree can be bound by a constant. In the absence of faults, the fault-tolerant version was shown to not incur significant overheads compared to the non-fault-tolerant version. Experimental evaluation through injection of faults at various stages of execution showed the presented algorithm can efficiently recover from an arbitrary number of faults with costs that are roughly proportional to the amount of work lost.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under award number 63823. It also is partially supported by NSF award CCF-1318420 to the Ohio State University.

References

[1] H. Quinn and P. Graham, "Terrestrial-based radiation upsets: a cautionary tale," in *FCCM*, 2005, pp. 193–202.

[2] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: a large-scale field study," *CACM*, vol. 54, no. 2, pp. 100–107, 2011.

[3] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. B. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *SC*, 2012, pp. 78:1–78:12.

[4] X. Qin and H. Jiang, "A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems," *Parallel Computing*, vol. 32, no. 5, pp. 331–356, 2006.

[5] G. Fohler, "Adaptive fault-tolerance with statically scheduled real-time systems," in *RTS*, 1997, pp. 161–167.

[6] O. González, H. Shrikumar, J. A. Stankovic, and K. Ramamritham, "Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling," in *RTSS*, 1997, pp. 79–89.

[7] T. Johnson, "A concurrent dynamic task graph," in *ICPP*, 1993, pp. 223–230.

[8] K. Agrawal, C. E. Leiserson, and J. Sukha, "Executing task graphs using work-stealing," in *IPDPS*, 2010, pp. 1–12.

[9] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for high performance computing," *Parallel Computing*, vol. 38, no. 1, pp. 37–51, 2012.

[10] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach *et al.*, "Concurrent collections," *Scientific Programming*, vol. 18, no. 3, pp. 203–217, 2010.

[11] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *DSN*, 2012, pp. 1–12.

[12] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *SPAA*, 1998, pp. 119–129.

[13] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[14] M. Y. H. Low, W. Liu, and B. Schmidt, "A parallel BSP algorithm for irregular dynamic programming," in *APPT*, 2007, pp. 151–160.

[15] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ASPLOS*, 2012, pp. 301–312.

[16] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: approximate data types for safe and general low-power computation," in *PLDI*, 2011, pp. 164–174.

[17] J. Bornholt, T. Mytkowicz, and K. S. McKinley, "Uncertain <T>: A first-order type for uncertain data," in *ASPLOS*, 2014, pp. 51–66.

[18] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *OOPSLA*, 2013, pp. 33–52.

[19] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality programmable vector processors for approximate computing," in *MICRO*, 2013, pp. 1–12.

[20] M. Samadi, D. A. Jamshidi, J. Lee, , and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *ASPLOS*, 2014, pp. 35–50.

[21] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *MICRO*, 2013, pp. 25–36.

[22] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez, "Containment domains: a scalable, efficient, and flexible resilience scheme for exascale systems," in *SC*, 2012, pp. 58:1–58:11.

[23] X. Ni, E. Meneses, N. Jain, and L. V. Kalé, "Acr: automatic checkpoint/restart for soft and hard error protection," in *SC*, 2013, pp. 7:1–7:12.

[24] D. Li, J. S. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *SC*, 2012, pp. 57:1–57:11.

[25] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Computers*, vol. 33, no. 6, pp. 518–528, 1984.

[26] Z. Chen, "Extending algorithm-based fault tolerance to tolerate fail-stop failures in high performance distributed environments," in *IPDPS*, 2008, pp. 1–8.

[27] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen, "High performance linpack benchmark: a fault tolerant implementation without checkpointing," in *ICS*, 2011, pp. 162–171.

[28] H. Liu, T. Davies, C. Ding, C. Karlsson, and Z. Chen, "Algorithm-based recovery for Newton's method without checkpointing," in *IPDPSW*, 2011, pp. 1541–1548.

[29] Z. Chen, "Algorithm-based recovery for iterative methods without checkpointing," in *HPDC*, 2011, pp. 73–84.

[30] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, pp. 406–471, 1999.

[31] K. Hashimoto, T. Tsuchiya, and T. Kikuno, "Effective scheduling of duplicated tasks for fault tolerance in multiprocessor systems," *IEICE Trans. on Info. and Systems*, vol. 85, no. 3, pp. 525–534, 2002.

[32] A. Girault, H. Kalla, M. Sighireanu, Y. Sorel *et al.*, "An algorithm for automatically obtaining distributed and fault-tolerant static schedules," in *DSN*, 2003, pp. 165–190.

[33] A. Benoit, M. Hakem, and Y. Robert, "Fault tolerant scheduling of precedence task graphs on heterogeneous platforms," in *IPDPS*, 2008, pp. 1–8.

[34] N. Tabbaa, R. Entezari-Maleki, and A. Movaghar, "A fault tolerant scheduling algorithm for dag applications in cluster environments," in *ICDIPC*, 2011, pp. 189–199.

[35] E. Maehle and F.-J. Markus, "Fault-tolerant dynamic task scheduling based on dataflow graphs," in *Fault-Tolerant Parallel and Distributed Systems*, 1998, pp. 357–371.

[36] N. Vrvilo, V. Sarkar, K. Knobe, and F. Schlimbach, "Implementing asynchronous checkpoint/restart for CnC," Concurrent Collections workshop, 2013.

[37] E. Meneses, "Scalable message-logging techniques for effective fault tolerance in HPC applications," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2013.

[38] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *SOSP*, 2007, pp. 205–220.

[39] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *CACM*, vol. 51, no. 1, pp. 107–113, 2008.

[40] D. E. Bakken and R. D. Schlichting, "Supporting fault-tolerant parallel programming in Linda," *TPDS*, vol. 6, no. 3, pp. 287–302, 1995.

[41] A. Xu and B. Liskov, "A design for a fault-tolerant, distributed implementation of Linda," in *FTCS*, 1989, pp. 199–206.