A Work-Stealing Scheduling Framework Supporting Fault Tolerance

Yizhuo Wang, Weixing Ji, Feng Shi, Qi Zuo
School of Computer Science and Technology
Beijing Institute of Technology
Beijing, China
{frankwyz, jwx,sfbit,zqll27}@bit.edu.cn

***Abstract*—Fault tolerance and load balancing are critical points for executing long-running parallel applications on multicore clusters. This paper addresses both fault tolerance and load balancing on multicore clusters by presenting a novel work-stealing task scheduling framework which supports hardware fault tolerance. In this framework, both transient and permanent faults are detected and recovered at task granularity. We incorporate task-based fault detection and recovery mechanisms into a hierarchical work-stealing scheme to establish the framework. This framework provides low-overhead fault-tolerance and optimal load balancing by fully exploiting task parallelism.**

Keywords—fault tolerance; work-stealing; multicore; cluster

I. INTRODUCTION

Today, most of high performance computing platforms such as clusters, grids and desktop grids are built from multicore machines connected via high speed interconnects. To compute long-running applications in this kind of systems, two key issues need to be resolved: *1) How to exploit parallelism for performance enhancement of the application? 2) How to make long-running computations resilient to hardware faults?*

To exploit parallelism of the multiple processing elements (PEs), efficient task partitioning of the application and scheduling of these tasks onto parallel PEs are of utmost importance. Work-stealing is a popular task scheduling approach which achieves an efficient dynamic load balancing. Under work stealing, each processor maintains its own work queue of tasks and idle processors attempt to steal work from victim processors selected randomly. On shared memory systems, work-stealing has been studied extensively. Most of the popular parallel programming models and languages offer work-stealing schedulers for task parallelism e.g., Cilk [1], Intel Threading Building Blocks (TBB)[2], Microsoft Task Parallel Library (TPL)[3], OpenMP 3.0, and Java Concurrency Utilities. On distributed memory systems, recent researches [4][5][6][7] also show good performance by using work-stealing schedulers, and there are some public available implementations of work-stealing on clusters of SMP machines, such as X10 [8] and Kaapi [9].

Work-stealing has native fault tolerance capability. For example, in Cilk-NOW [10], an implementation of the Cilk runtime system for networks of workstations, if a processor crashes, the other processors automatically steal and redo the work that was lost in the crash. Any work-stealing algorithm can be implemented to tolerate such processor failures. But for another type of hardware faults—transient faults [11], there is not a work-stealing technique scheme providing tolerance support. As the number of processors in a computing system increases, the probability of a transient fault increases, which leads us to design a fault tolerant work-stealing scheme.

Fault tolerance involves both fault detection and fault recovery. Most of the transient fault detection techniques are replication-based. In [11][12][13], operations are duplicated at different granularities (instruction, thread and process). The results of original operations and duplicated operations are compared to detect a fault. Fine-grained replication and comparison provides fast reaction to a fault, but needs tightly-coupled synchronization which harms to dynamic load balancing. Coarse-grained replication is vice versa. For task parallel applications, it is natural to use a task as a basic unit of replication and comparison. Task has a medium granularity and gives a tradeoff between fault response speed and load balancing. However, as far as we know, none of the existing fault detection methods work at task granularity.

In this paper, we introduce task-based fault detection and recovery. Each task is scheduled on two PEs to execute twice in parallel. Each worker thread runs on a different PE, and each thread maintains a private space for shared data. Here, the shared data is the data which may be write-accessed concurrently by the two copies of a same task. During the execution of a task, any write to the shared data is buffered in thread's private space. After the double executions, the buffered data in two spaces is compared. If the data does not match, a fault is detected and the task will be re-executed. Otherwise, the data is committed to their original memory addresses. We combine this task-based fault detection with the work-stealing scheduling, and propose a fault tolerant work-stealing framework which minimizes the performance degradation due to fault detection by fully exploiting task parallelism.

Once a fault (transient or permanent) is detected, the program needs to restart the computation from a previously established state in the computation before the occurrence of the fault. Checkpoint/restart is the most commonly used fault recovery approach. Most existing systems for checkpointing such as LAM/MPI and Condor take system-level checkpoints which consists of memory contents, register values and process

978-3-9815370-0-0/DATE13/©2013 EDAA

context. Checkpoint/restart in these systems is actually a mechanism for process migration between cluster nodes. For SPMD applications, a same program runs on all worker nodes. If an application is partitioned into tasks, task migration can be implemented to replace process migration. The data of a task needs to be stored in a checkpoint before running this task. This is application-level checkpointing technique [14] which reduces the size of the saved state and can make a task restarted on any platform.

In our technique, task queues of each cluster node and the relevant data are saved periodically in a checkpoint. If a node crashes, another node will restart the tasks in the checkpoint. According to the storage solution, checkpointing techniques can be classified as disk-based checkpointing and diskless checkpointing. Disk-based checkpointing stores checkpoints in stable storage. Diskless checkpointing techniques use the volatile memory of other computers within the system to store the data instead of using stable storage. Diskless checkpointing reduces the overhead of storing checkpoints, but cannot survive the failure of the whole system. In out framework, we assume that there is at least one processor working at any moment. Diskless checkpointing is applied in our framework for recovery from permanent faults.

The rest of the present paper is organized as follows. In Section 2, work-stealing and fault tolerance techniques are reviewed. A novel fault tolerant work-stealing framework is presented in Section 3. The implementation of the proposed framework is described in Section 4. Section 5 presents experimental results and Section 5 concludes the paper.

II. Background and Related Work

A. Work-stealing

Work-stealing is the most popular way to achieve dynamic load balancing in the execution of task parallel applications. The basic work-stealing paradigm is shown in Figure 1. Each processor (worker) maintains a local task queue which is double-ended. Tasks are dynamically generated during the execution and enqueued to or dequeued from a task queue at the queue's bottom end at runtime. Tasks in all queues are independent each other and can be executed in parallel. When a processor's task queue is empty, the processor will steal one task or a group of tasks from another processor, called a *victim* which is randomly selected normally. To minimize synchronization overhead for the queue's owner, stolen elements are always taken from the top end of the queue.

Work-stealing has been implemented for shared memory systems in TBB[2], Cilk++[15], TPL[3], etc., and for distributed memory systems in Satin[16], ATLAS[17], Kaapi[9], etc. However, as far as we know, there is not a work-stealing system like ours which supports both transient and permanent fault tolerance.

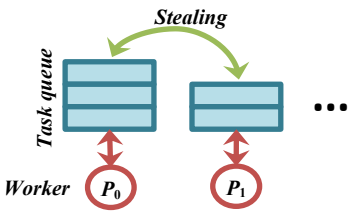

Figure 1. Classic work-stealing.

B. Fault Tolerance

Fault tolerance requires at least two basic mechanisms: fault detection and fault recovery.

1) Fault Detection

In clusters, heartbeat mechanism is widely used to detect permanent faults of the nodes. Permanent fault (node failure) detection schemes only differ on the implementations.

Transient fault detection necessitates redundancy in general. EDDI [11] duplicates instructions during compilation and inserts appropriate instructions to check the results. AR-SMT [18] uses SMT to execute two copies of the same program to detect a fault. AR-SMT was later improved by many researchers in different architectures (e.g. CRT[19], SRT[12]). These techniques run a "leading" thread and a duplicated "trailing" thread, and compare their outputs to detect error. Process-level Redundancy (PLR) [13] is a software technique for transient fault tolerance, which creates a set of redundant processes per application process and systematically compares the processes to guarantee correct execution. Above techniques introduce redundancy at different granularities. But none of them implements fault detection at task level like ours.

2) Fault Recovery

Prior work in fault recovery can be classified into two categories: backward error recovery (BER) and forward error recovery (FER). FER techniques detect and correct the errors without requiring to rollback to a previous state. N-version programming [20] is a typical FER mechanism which uses redundant code execution and software-implemented voting to achieve fault recovery.

BER techniques periodically save a checkpoint and rollback to the latest checkpoint when a fault is detected. Such checkpoint/restart mechanism has been widely used in HPC field by systems such as Condor, LAM/MPI, Open MPI, FT-MPI and MPICH-V. We use diskless application-level checkpoint/restart in our system.

III. Architecture

We first present a hierarchical work-stealing scheme for multicore clusters. Based on it, we set up our fault tolerant work-stealing framework.

A. Hierarchical Work-stealing Scheme

Figure 2 depicts the system model for dynamic task scheduling in a multicore cluster environment. In general, it is a master-worker model. An application is partitioned into tasks and the task dependences are represented as a directed acyclic graph (DAG). The application and its DAG are submitted to a master node where a global scheduler works. When the job is submitted to the master node, the global scheduler performs initial partitioning with a static scheduling algorithm. The initial partitioning is essential on distributed memory systems because the cost of task stealing between cluster nodes is much higher than it between threads on a shared memory system. The initial partitioning could balance the workload before the parallel execution of the tasks and thus reduce the frequency of dynamic task stealing across the nodes. Therefore, our work-stealing algorithm of the global scheduler starts with an initial partitioning phase. For tasks with different patterns of parallelism, the static partitioning algorithms are different.

- For *flat parallelism* provided by parallel loops, the simplest way is to assign even partitions of the loop iterations to the workers.
- For *recursive parallelism* provided by divide-and-conquer algorithms, the first few steps of the recursive calculation must be done at the master node to spawn enough subtasks based on the number of available worker nodes.
- For *irregular task parallelism*, a static DAG scheduling algorithm will be applied to partition the tasks into batches.

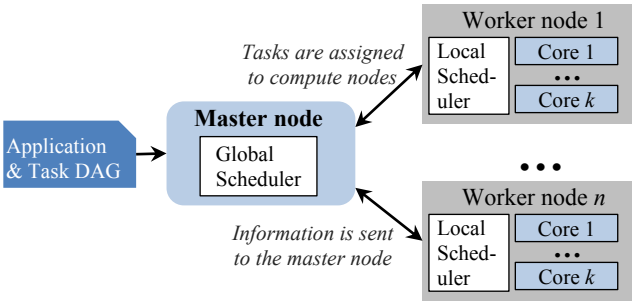

Figure 2. The system model for task scheduling in a multicore cluster.

After the initial partitioning, tasks are dispatched to the worker nodes. Each node runs a local scheduler instance, which balances the workload between the PEs on the node with a work-stealing algorithm. The local scheduler can be implemented by expanding any existing work-stealing scheduler for shared memory system. To support transient fault detection, every two PEs in a same chip share a local task queue and a faulty task queue in our system (see next section for details). When the task queues on a node are empty, a message is sent to the master node to require a task. The global scheduler determines whether to steal a task from another node or to assign a subsequent task to the requestor.

The inter-node work-stealing (implemented in the global scheduler) and the intra-node work-stealing (implemented in the local scheduler) establish a hierarchical work-stealing scheme. It is similar to the state-of-the-art work-stealing scheme for multicore HPC clusters in [6]. But we adopt the initial partitioning to reduce the inter-node steals. In the next two subsections, we describe the details of the fault tolerance mechanisms in our framework.

B. Transient Fault Tolerance

To detect a transient fault, each task is executed twice on two cores, and the results (output data of the task, i.e., the data which is written in this task, but not locally accessed only in this task) of the double executions are compared.

We make the following rule: each task must be executed twice in two different processing elements (PEs). Thus, two PEs share a task queue in our work-stealing framework, as shown in Figure 3. Task is stolen from the top end of the task queue. The two PEs obtain a same task from the bottom end of their task queue to execute. Besides the task queue, every pair of PEs has another task queue in which the faulty tasks coming from other task queues are recorded. Faulty tasks are those that the results of the double executions are not identical. A faulty task running on a pair of PEs will be transferred to a faulty task queue of another pair of PEs, as dot arrow lines in Figure 3. A task element has two tags (P' and P'') which refer to the two PEs which are running this task.

As shown in Figure 4(a), P_0 and P_1 share a task queue. There are two tasks C_0 and C_1 in the task queue. P_0 and P_1 get C_0 to execute at the beginning. The tags P' and P'' of C_0 equal -1 initially. P_0 set P' of C_0 to 0 (the ID of P_0) and P_1 set P'' to 1 (the ID of P_1) before they execute C_0. P_0 updates the tag P' and P_1 updates P'' of the tasks during the execution. Synchronization is required between P_0 and P_1 to determine whether the double executions complete.

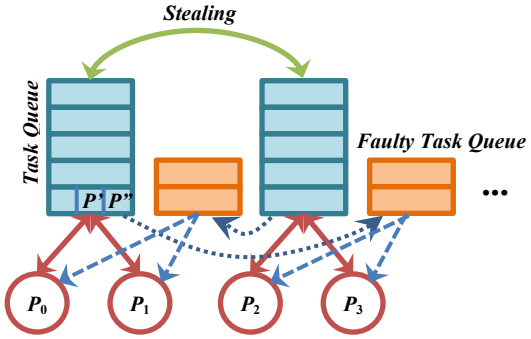

Figure 3. Fault tolerant work-stealing.

In practice, the double executions are not possible to start or end at the same time because of the real environment and the synchronization. Let's assume P_0 finishes the task C_0 before P_1 as shown in Figure 4(a). P_0 will continue to get the next task in the queue to execute (each PE has its own head pointer to the task queue). When P_1 finishes C_0, it will compare the results of the double executions. If they are equal, P_1 will commit the data, move task C_0 out of the task queue and enqueue the new tasks while there are new tasks spawned. If not equal, P_1 will transfer C_0 to a faulty task queue selected randomly from the faulty task queues of other PE pairs.

In Figure 4, P_0 completes C_0 early. There are following three possibilities of the next scheduling step of P_0:

1) P_0 checks its faulty task queue firstly in order to recover from the faults as soon as possible. If there is a task, as C' in Figure 4(b), P_0 gets C' to execute, and then compares the results, commits the data and updates the task queue that C' comes from, or transfers C' to another faulty task queue if the results are not equal.

2) If there is not task in the faulty task queue and there is task in the task queue, as C_1 in Figure 4(a). P_0 gets C_1 to execute. P_0 checks whether P_1 works or not before the execution of the next task. If P_1 crashes, P_0 transfers C_0 to a faulty task queue of other PE pairs directly.

3) If there is not task in the local task queue and faulty task queue, as shown in Figure 4(c), P_0 will steal a task from other task queues. Termination detection is done when P_0 attempts to steal a task. If all the task queues are empty, there are following two possibilities:

a) If all the faulty task queues are empty, the job is finished.

b) If there are tasks in the faulty task queues, P_0 will get one of them to execute and compare the results. If the results are not equal, P_0 will re-execute the task again and commit the data directly without comparison. It is

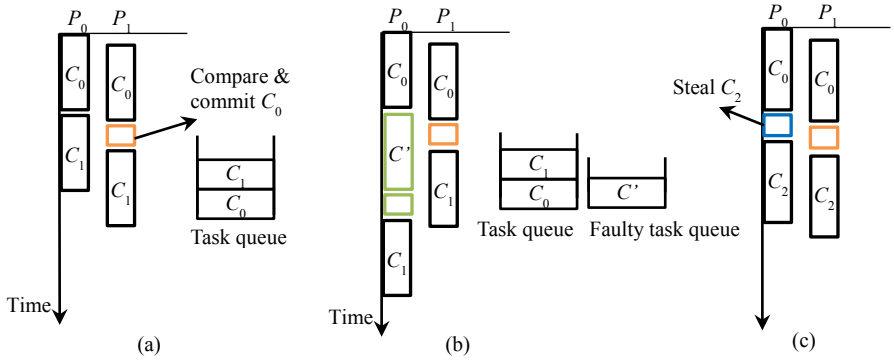

Figure 4. Task queues and scheduling for sample execution.

to avoid that a faulty task is transferred between the faulty task queues infinitely which results that the job is never completed.

In a multicore cluster, when a transient fault is detected, the faulty task is firstly re-executed by another core in the same node on which the task just runs. Only if a fault cannot be recovered within a node, the faulty task would be dispatched to another node, that is, be pushed onto a faulty task queue located on another node. In our current implementation, we limit the migration of faulty task inside a worker node because inter-node migration of faulty tasks would make the system complex and the fault frequency is normally not so high that a task cannot be correctly executed once on the node.

To ensure the correctness of the multiple times executions of a same task on a node, we use a buffer-and-commit computation model which is similar to the SpiceC parallel computation model in [21]. In this model, the double executions of a task load shared data from the same place, but store the data to different buffers. If the contents of the two buffers compare as equal after the executions, the data in one of the buffers is committed to their original addresses. Otherwise, a fault is detected and the task is pushed onto a faulty task queue to be re-executed.

Discussion. Our technique cannot detect a fault which occurs during the task scheduling or the data comparing. However, considering the overhead of these operations is relatively low compared with the execution time of tasks, our technique has reasonable fault coverage. In addition, if undetected faults cause a processor crash, the tasks assigned to it will be stolen by other processors and the faulty task will migrate to another processor in our framework. Therefore, the fault recovery coverage of our framework is high although the fault detection coverage is not high.

C. *Permanent Fault Tolerance*

The master node is responsible for permanent fault (node failure) detection in our system. A heartbeat is sent from each worker node to the master node periodically to check its status. When the master node exceeds a maximum waiting time without receiving any information from a particular worker node, it suspects that the worker node has failed.

In order to recover from a node failure, a diskless checkpointing scheme is used in our system. The information of the task queues on a node and the interim data needed to restart the tasks are saved in a checkpoint periodically. Note that we save checkpoints periodically not with a time interval but with a task interval, that is, after the execution of every few tasks. The checkpointing frequency is determined by the user with respect to the sizes of the tasks. During checkpointing, the checkpoint of a worker node is transferred to another node and stored in its local memory. The master node maintains a table which records the node where a checkpoint is stored. The initial state of the table and the changes after a node failure are represented in Figure 5. The circles represent the nodes and the arrows represent the checkpoint transmissions. The basic rule of checkpoint mapping is transferring a checkpoint to the nearest neighbor.

After a node failure is detected, the master node notifies the worker node which stores the checkpoint of the crashed node to re-execute the tasks in the checkpoint. Unlike traditional checkpointing method which saves the state of a process, we save minimum necessary information of the tasks in a checkpoint. The task information is related to the application. For example, in a standard matrix multiplication program, a few row and column numbers can be used to represent a task.

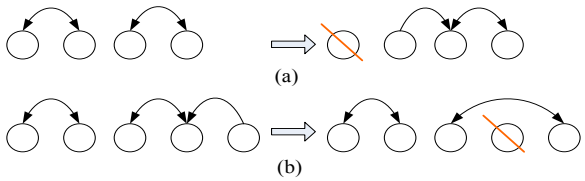

Figure 5. Checkpoint mapping between the nodes and the changes after a node failure. (a) An even number of nodes. (b) An odd number of nodes.

Even though we do not deal with the master node failure in the proposed framework, there are several ways in which the failure of the master node can be handled. For example, backups for the global scheduler can be scheduled on some other nodes in the system. When the master node fails, another node with a global scheduler becomes a new master.

IV. IMPLEMENTATION

This section describes the prototype implementation of our framework. We implement the global scheduler (*GS*) and the local scheduler (*LS*) in a runtime library. Following core classes are defined in the library: *LS*, *GS*, *Task* and *Worker*. *LS* and *GS* provide scheduling methods, communication routines and termination detection function. A sample task class implementation for standard matrix multiplication (*C=A*B*) is

shown in Figure 6(a). To detect transient fault, a task is executed twice by two workers calling *Execute1( )* and *Execute2( )* respectively, and the results are compared after the executions. The two workers write to different buffers *c1* and *c2* which are allocated in *Execute( )* and freed in *Commit( )*. We use memory pools to manage the buffers. To support permanent fault recovery, two functions, *Checkpoint* and *Restart* are defined in the class. During checkpointing, each task in task queues saves the necessary information to a specified structure, *TaskInfo*. The local scheduler is responsible for combining these *TaskInfo* objects into a checkpoint. Checkpoint transmissions and task migrations across the nodes are all represented as messages. We use MPICH2 to implement the message passing between the nodes.

Every *Worker* object starts a new thread and bind it to a processor. Pseudocode of the worker thread implementation is shown in Figure 6(b). A task is obtained by calling *LS.Schedule( )* method and executed. After the execution, tags are checked to determine whether the double executions of the task are finished. Then the thread compares the results to detect a fault or continues to ask for a task.

```
class Task : public TaskBase {  ...
   double *a, *b, *c;  // Point to the data of the matrices.
   double *c1, *c2;   // Point to the buffers for the double executions.
   int startrow,size; // Portions of the matrices computed in this task.
   void  Spawn (TaskQueue *tq);
   Task*  Execute1( ) {  Execute (c1);  }
   Task*  Execute2( ) {  Execute (c2);  }
   void  Execute (double *c);  //  c = a*b;
   bool  Compare ( );  // Compare the data in the buffers c1 and c2.
   void  Commit ( );   // Copy c1 to c.
   ...
   TaskInfo*  Checkpoint ( );
   Task*  Restart (TaskInfo *tinf);
};
                                        (a)
while (true) {
   Task* t = LS.Schedule ( );
   if (thread_id%2 == 0)  t->Execute1( );
   else  t->Execute2( );  ...
   if ( tag1 && tag2){   // The double executions are finished.
      if ( true == t->Compare( ))   // No fault occurs.
         t->Commit( );
      else
         LS.FaultyTQ_push( t );  // Push t to a faulty task queue.
   }...}
                                        (b)
```

Figure 6. Pseudocode for (a) task definition and (b) worker thread implementation.

V. EVALUATION

In this section, we evaluate our framework using the task-parallelized applications in Table 1. Experiments were conducted on a cluster which has 16 multi-core nodes. Each node is equipped with 12G memory and a 2.4GHz quad-core Intel Xeon E5620 processor which supports 8 hardware threads. Each node is connected to a switch on Gigabit Ethernet.

To measure the performance overhead of our framework, we compare our framework, denoted by FTWS (Fault Tolerant Work-Stealing), with a non-fault-tolerant framework, denoted by HWS (Hierarchical Work-Stealing). HWS is implemented by just removing fault tolerance elements in the prototype implementation of FTWS. Therefore, comparing FTWS with HWS, we can get the fault tolerance overhead. We run the benchmarks in a fault free environment. The execution times (normalized w.r.t. HWS) are reported in Figure 7. The performance overhead of FTWS is 39% (or 1.39x) on average. The overhead comes from the compare and commit operation, checkpointing and the double executions. The costs of compare-commit operations and checkpointing depend on the output data of tasks (see Section 3.2). For Fib(n) and Nq(n), each task has a small output. In contrast, tasks of Ms(4G), MM(12k) and St(12k) produce a lot of data. Therefore, we observe the overhead of Fib(52) and Nq(20) is less than that of Ms(4G), MM(12k) and St(12k). The double executions would make the overhead more than 100% in theory. However, the maximum overhead in our experiments is 57%, which can be attributed to efficient load balancing and fine-grained task parallelism in our framework. In addition, for MM(n) and St(n), small input sets reduce the overhead significantly. The minimum overhead of 21% is achieved on St(6k). It is due to the fact that not all PEs are really used when there are not enough tasks produced because of the small input sets. FTWS increases PEs usage because each task is executed twice on different PEs.

TABLE I. BENCHMARK APPLICATIONS

Benchmark	Description
Fib (n)	Recursively compute the nth Fibonacci number.
Nq (n)	The n-queens problem.
MM (n)	Standard matrix multiplication using a loop nest where the outermost loop is parallelized (n*n double matrix).
Ms (n)	Parallel merge sort on an integer array of size n.
St (n)	Dense matrix multiplication using Strassen's algorithm (n*n double matrix).

To measure the fault coverage of our framework, we run the benchmarks 1000 times with small input sets. Pin tool instrumentation [22] is used to inject a single bit flip fault in each run. The behaviors of program after injecting a fault are categorized as *Detected*, *Seg Fault*, *Timeout* and *Benign*. *Timeout* means the thread which is influenced by the fault has not exit when all the tasks are finished. This thread may fall into an infinite loop. Note that FTWS could still finish the work in this case. *Benign* means the fault does not affect the program's execution and the program finishes with correct output. Our experimental results in Figure 8 show that 62% faults are detected and recovered on average. Although 28% cause segmentation faults and 6.8% make a thread *Timeout*, all runs successfully complete the work due to the permanent fault recovery mechanism in FTWS. Therefore, FTWS provides 100% fault recovery coverage in practice. Note that, in theory, the fault recovery coverage of FTWS is not 100% because the program will finish with incorrect result in the following cases: 1) two transient faults occur at the same location of the double executions of a task; 2) a fault occurs when a faulty task is finally re-executed (see Section 3.2). However, the possibility of these cases is extremely low.

To measure the permanent fault recovery overhead, we kill the process running on a cluster node to simulate a node crash. Figure 9 shows the execution times of the benchmarks in fault-free case, faulty case with one node crash and faulty case with 2 nodes crashes. Recovery time from a checkpoint can be calculated by subtracting fault-free execution time from faulty case execution time. From the figure, we see that the recovery overhead is rather small. It is because we use diskless

checkpoint/restart and the tasks reloaded from a checkpoint are executed in parallel.

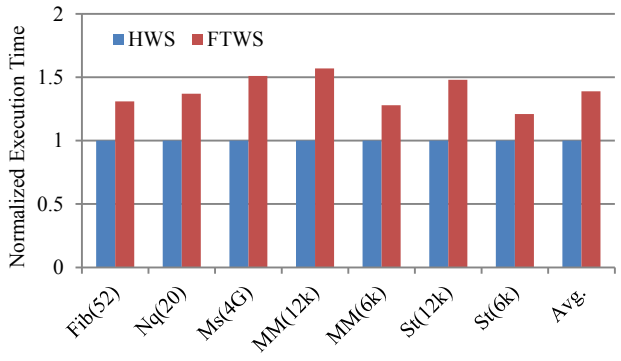

Figure 7. Performance comparison of FTWS with HWS.

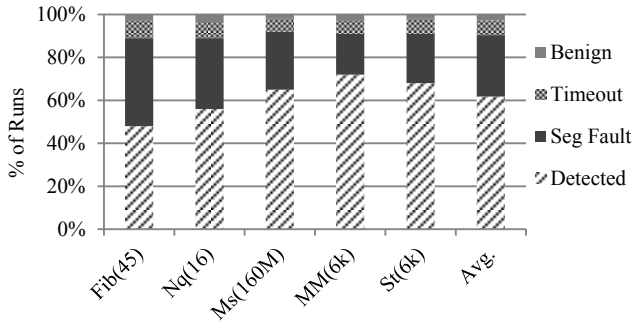

Figure 8. Fault detection distribution.

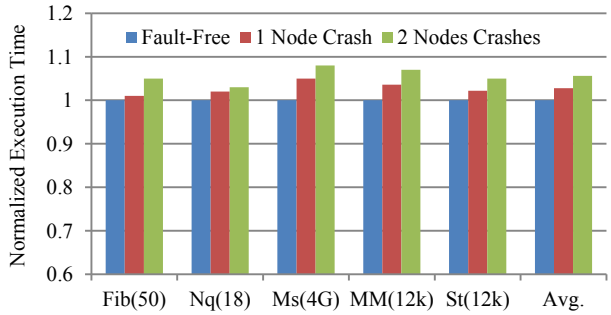

Figure 9. Performance of FTWS with permanent faults

VI. CONCLUSION

In this paper we propose a fault tolerant work-stealing framework (FTWS) for multicore clusters. This framework supports both transient and permanent fault tolerance at task granularity. FTWS is a pure software technique. Our experimental results show that FTWS provides high fault coverage at very low cost. As future work, we plan to support fault tolerance to the master node failure in our system and investigate the behavior of FTWS with some other scientific applications.

ACKNOWLEDGMENT

This work was supported in part by the National Natural Science Foundation of China under grant NSFC-60973010. We would like to thank the anonymous reviewers for their helpful comments on earlier versions of this manuscript.

REFERENCES

[1] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5): 212–223, 1998.

[2] Intel(R) Threading Building Blocks, Intel Corporation.

[3] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *OOPSLA*, pp. 227-242, 2009.

[4] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-based global load balancing. In *PPoPP*, pp. 201–212, San Antonio, USA, 2011.

[5] J.-N. Quintin and F. Wagner. Hierarchical work-stealing. In *EuroPar*, pp. 217–229, Berlin, Heidelberg, 2010.

[6] K. Ravichandran, S. Lee, and S. Pande. Work stealing for multi-core HPC clusters. In *EuroPar*, pp. 205-217, Berlin, Heidelberg, 2011.

[7] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *SC*, pp. 1–11, New York, NY, USA, 2009.

[8] P. Charles, C. Grothoff, et al. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pp. 519–538, 2005.

[9] MOAIS software: http://kaapi.gforge.inria.fr

[10] R. D. Blumofe. Executing Multithreaded Programs Efficiently. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.

[11] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Trans. Reliability*, 51(1):63-75, March 2002.

[12] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA*, pp. 25-36, 2000.

[13] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, D. A. Connors. PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures. *IEEE Trans. Dependable Sec. Comput.* 6(2): 135-148, 2009.

[14] G. Bronevetsky, D. Marques, et al. Application-level checkpointing for shared memory programs. In *ASPLOS*, pp. 235-247, 2004.

[15] Cilk Arts: http://www.cilk.com.

[16] R. V. van Nieuwpoort, G. Wrzesińska, C.J.H. Jacobs, and H. E. Bal. Satin: A high-level and efficient grid programming model. *ACM Trans. Program. Lang. Syst.* 32(3), 39 pages, 2010.

[17] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. Atlas: an infrastructure for global computing. In *Proceedings of the 7th workshop on ACMSIGOPS European workshop*, pp. 165–172, New York, USA, 1996.

[18] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, pp. 84-91, 1999.

[19] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. *SIGARCH Comput. Archit. News*, 30(2):99–110, 2002.

[20] A. Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Trans. On Software Engineering*, 11(12): 1491-1501, 1985.

[21] M. Feng, R. Gupta, and Y. Hu. SpiceC: scalable parallelism via implicit copying and explicit commit. In *PPoPP*, pp. 69-80, 2011.

[22] C.-K. Luk, et al. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pp. 190–200, New York, USA, 2005.