An Error Detection and Tolerance Framework for Task Parallel Applications on High Performance Computing Systems

Yanfei Fang [1], Enming Dong[1], Yanbing Li[1], Qi Liu[1], Fengbin Qi[1] and Peibing Du[2]
[1]National Research Center of Parallel Computer Engineering and Technology, Beijing 100190, China
[2]Northwest Institute of Nuclear Technology, Xian 710024, China

Keywords: Error Detection, Fault Tolerance, High Performance Computing, Task Parallel Applications.

Abstract: With the high performance computing capability entering the E-level era, the computing scale of the system reaches more than 10 million cores. The mean time between failures of the system is short, which brings great challenges to the reliability of the system. Single processor failure is a common system failure. If the failure can be detected by the system, system-level fault tolerance can be implemented, and fault tolerant processing can be performed through technologies such as checkpoint rollback. However, there are single processor faults that cannot be detected by the system. These faults are manifested as wrong operation results, which cannot be detected by the system. To solve the above problems, an error detection and fault tolerance framework for task parallel applications is proposed. The framework consists of three functions: dynamic task scheduling, error detection, and fault tolerance. During the running process of task parallel applications, error detection is actively initiate. When a node failure is detected, the failed node is discarded. And tasks assigned to the node since the last checkpoint are reassigned to other healthy nodes. The experimental results show that the framework can effectively detect node failures. The fault tolerance can be performed without interrupting the operation of the project, effectively avoiding the time cost caused by the checkpoint rollback technology.

1 INTRODUCTION

With the increasing demand for high performance computing systems in important fields such as national defense, scientific research, and finance, high performance computing technology has developed rapidly, and the scale of parallel systems has become increasingly large. However, with the expansion of the system scale and the complexity of the system design, the mean time between failure (MTBF) of the system decreases gradually, which brings serious challenges to the system reliability.

Error detection and fault tolerance are important foundations for maintaining the reliability of high performance computing systems. Error detection plays a key role in fault management. Some common system faults can be detected by hardware and maintenance software, but some node faults cannot be detected. If the system cannot detect the node operation faults, once the node operation error occurs during the application running process, it is likely to affect the results of the entire project, especially for large-scale parallel applications that need to run for days or even weeks. Therefore, low-overhead error detection and fault tolerance mechanisms can reduce the impact of failures on parallel applications, and are of great significance to improving the robustness, performance, and system availability of parallel applications.

The paper proposes a fault detection and fault tolerance framework for task parallel applications. The framework can regularly detect the health of nodes, find fault nodes, and recycle and reassign tasks on the nodes without interrupting the project, which can greatly reduce fault tolerance cost.

2 RELATED WORKS

2.1 Fault Tolerance Technology

Fault tolerance technology mainly includes three aspects: error detection, fault diagnosis and fault recovery. Error detection and fault recovery are related to the work of the paper.

Error detection can be divided into transient error and permanent error detection. Transient error detection and fault tolerance mainly relies on repeated execution and comparison. According to the replicated objects, transient error detection techniques during processing can be divided into instruction-level, thread-level and application-level fault tolerance.

EDDI (Oh et al., 2002) and SWIFT (Reis et al., 2005) are typical representatives of instruction-level fault tolerance, which copy the instructions in the original program at compile time, and insert comparison instructions at appropriate locations to detect errors. Thread-level fault tolerance method, like AR-SMT (Rotenberg E, 1999), SRT (Reinhardt et al., 2000) and CRT (Mukherjee et al., 2002), etc., use more than two hardware threads or cores to execute the same task. A specific cache is added to the processor to store the execution results of the two threads, to detect errors by comparing the execution results. Application-level fault tolerance method, like PLR (Shye A, et al., 2009), performs replication and comparison at a higher software level, such as copying a process into multiple redundant processes for concurrent execution, and then comparing the program output.

Permanent error detection techniques can be divided into two categories. One is the hardware module fault detection technology at the micro-architecture layer, which is often used in the design of reconfigurable processors. The other is the detection of node faults in high performance systems. Since the MTBF decreases sharply, the node faults are common in the system.

However, repeated execution may cost too much time, which is not adopted by high performance computing applications. High performance computing systems mainly screen node operation errors based on the screening point program, and screen out the wrong points in advance, but the screening program cannot cover all subject situations, and errors during applications execution cannot be found.

Error recovery techniques can be divided into two categories: forward error recovery and backward error recovery.

Forward error recovery tries to correct the error after the error is detected and continue to execute forward without roll back to the state before the error moment. Redundancy is the basic way to realize forward error recovery. Three Modular Redundancy (TMR) is a widely used FER technology, which uses 3 modules to perform the same operation, and then selects the data through a majority voter at the output to achieve fault tolerance, but this method requires 3 times the computing resources and the overhead is large, so this method is generally not used in high performance system.

Backward error recovery returns to the state before the error occurred after an error is detected. The widely used backward error recovery method is checkpoint. According to the content of the storage checkpoint, checkpoint technology can be divided into system-level checkpoint and application-level checkpoint technology (Bronevetsky et al., 2004; Faisal et al., 2018). According to the medium of storage checkpoint, it can be divided into disk-based and diskless checkpoint technology (Chen, 2010; Alshboul et al., 2019).

Usually, error detection and recovery techniques are combined together to ensure the correctness of the applications. A task-based parallel programming model is proposed in (Wang et al., 2016), in which work-stealing scheduling scheme supporting fault tolerance is adopted to achieve dynamic load balancing support fault tolerance.

2.2 Parallel Application Model and Task Scheduling

Most parallel applications can be divided into two categories: data parallelism and task parallelism. Task parallel applications usually decompose the task into many sub-tasks, divide the data set, and execute the tasks and corresponding data in parallel on different computing resources. Task parallel applications are widely used in drug screening, genetic research, cryptanalysis, nuclear simulation and other fields. There is no correlation between subtasks, but the calculation number of subtasks may vary significantly. In large-scale environments, an efficient load balancing mechanism is the key to ensure application performance, and the results of each subtask have an important impact on the overall results of the project.

Corresponding to task parallel applications, task division is divided into static division and dynamic division (Mohit et al., 2019). In static division, each computing node is statically divided into the same number of tasks and executed separately. Dynamic partitioningis to dynamically adjust the tasks of computing resources according to the load of each computing resource, including dynamic scheduling with management nodes and task stealing (Dinan et al., 2009), etc. In high performance computing, dynamic task partitioning is generally used to enable applications to more fully utilize computing resources(He et al., 2016).

3 THE ERROR DETECTION AND FAULT TOLERANCE FRAMEWORK

At present, high performance computing systems mainly screen node operation errors based on the screening point program, and screen out the wrong points in advance, but this method has the following shortcomings:1) The screening program cannot cover all subject situations. 2) The screening program often needs to run for a long time. The nodes cannot be screened at all times, so some errors cannot be found through screening.

In addition, the nodes of high performance computer systems are interconnected through specific hardware and high-speed networks. Most of the nodes are homogeneous, and the node states have similarities when performing computing tasks. In task parallel applications, a typical mode is that each node runs tasks independently, when all tasks are completed, it communicates with other nodes to complete the entire task cooperatively.

Based on the above characteristics of high performance computing applications, an error detection and fault tolerance framework for task parallel applications is proposed for high performance computing systems. The basic idea of the framework is to use the dynamic task scheduling feature to combine node screening with applications. After a certain time interval, let the nodes in a fixed area perform the same task, realize redundant operations in the runtime phase, and carry out through the reduction results. Correctness judgment is used to detect whether there is a node failure. When a node failure occurs, fault tolerance is performed by recycling the tasks on the node and abandoning the failed node.

3.1 Frame Structure

As shown in Figure 1, the framework first implements the dynamic task scheduling module for task parallel applications. On this basis, it implements the functions of error detection and fault tolerance. Error detection is mainly based on MPI messages. Fault tolerance is supported by system components such as the operating system and job management.

To achieve error detection and fault tolerance, dynamic task scheduling must be implemented first because:

1) Load balancing of large-scale high performance computing systems is very important and requires dynamic task scheduling;

2) Error detection needs to implement redundant operations in the running phase, and for task parallel applications, this redundancy can be achieved by letting the computing nodes within a certain range perform the same task, which can be done by assign the same task to different nodes by the dynamic task scheduling.

3) To achieve fault tolerance without interrupting the project, a checkpoint recording mechanism, as well as a task recovery and redistribution mechanism are needed, which can be well supported by dynamic task scheduling.

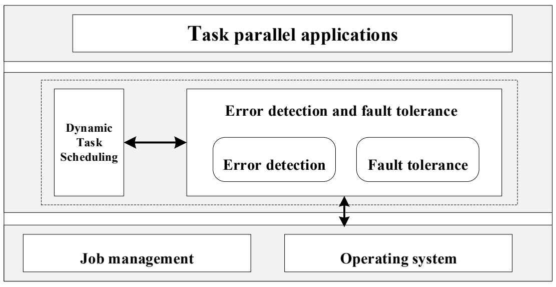

Figure 1: System Software Architecture of the Framework.

For applications running on high performance computing systems, the framework implements task scheduling, error detection, and fault tolerance. The main processes are as follows:

Step1: Initialize. Divide all computing resources into control nodes and computing nodes. Control nodes initiate task scheduling, error detection and fault tolerance.

Step2: Dynamic task scheduling. Control nodes assign tasks, while computing nodes execute tasks and report the completion of tasks.

Step3: Error detection. When the completed tasks reach a certain number, the control node initiates error detection. The computing nodes in a certain area calculate the same task, and compare the task results. If the results are correct, go to Step5. Else if a node is found to be wrong, go to Step4;

Step4: Fault tolerance. Report fault node to job management components and mark the faulty node. Recycle the tasks executed after the last checkpoint in the faulty node, and assign them to other healthy nodes for execution.

Step5: Record the checkpoint. When it is detected that all computing nodes are healthy and the completed tasks reach a certain threshold, record the checkpoint.

3.2 Dynamic Task Scheduling

The dynamic task scheduling module provides task scheduling support for the framework. It adopts the design idea of a partitioned hierarchical tree, as shown in Figure 2. During initialization, all nodes participating in the calculation are divided into control nodes (root nodes) and operation nodes (leaf nodes). In order to prevent a single control node from intensively affecting performance, the hierarchical idea is adopted, and the control nodes are divided into a global master node and some regional master nodes. The number of the regional master nodes is the same to the number of the regions. The control node performs task scheduling, and the leaf nodes perform task operations. Node types are shown in table 1.

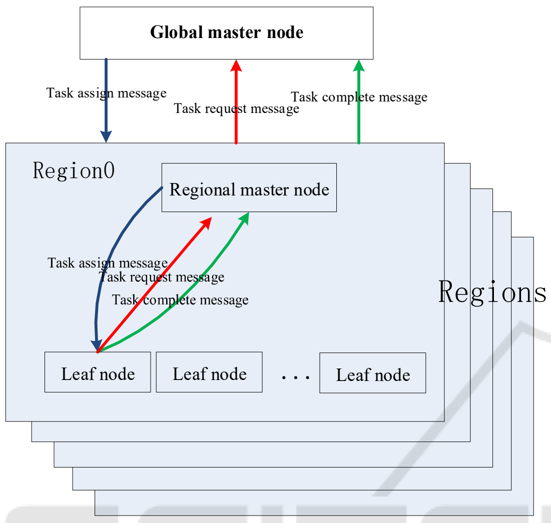

Figure 2: Diagram of dynamic task scheduling structure.

Table 1: Node types

Function type	Tree type	Node number
Control node	Global master node	The first node of all the computing nodes
	Regional master nodes	The first node of each region
Operation nodes	Leaf node	Nodes beside the control node

Task scheduling is implemented based on MPI messages. First, register the dynamic task message type, which supports the following types of messages:

Task request message (TRM). When the non-global master node finds that there are no tasks to be completed locally, it sends a task request message to the upper node, including the message request from the leaf node to the regional master node, and the message from the regional master node to the global master node.

Task completion message (TCM). When the non-global master node completes the task or task package, it sends the task completion message to the upper node, including the task completion message from the leaf node to the regional master node and the task completion message from the regional master node to the global master node.

Task assignment message (TAM). When the non-leaf node receives the task request message, it will detect whether there are still tasks in the local task pool, and if so, it will send the task assignment message to the requester, including task assignment message from the regional master node to the leaf node and from the global master node to the regional master node;

Task end message (TEM). When all tasks have been completed, the global master node informs the regional master nodes that the task is completed, and the regional master nodes inform the leaf nodes that the task is completed.

The main process of dynamic task scheduling is as follows:

Step1. Initialize hierarchical environment. Divide all computing resources into global master nodes, regional master nodes and leaf nodes. The global master node initializes the global task pool for all tasks, the regional master node initial task pool is empty, and the leaf nodes initially have no tasks.

Step2. Start the task application. The leaf node detects that the task pool is empty and applies for the tasks from the regional master node. After the regional master node receives the task application request, it checks that the regional task pool is empty, and applies for the task from the global master node.

Step3. Start task allocation. After the global master node receives the task request from the regional master node, it allocates the task block to the regional master node and records it. After the regional master node receives the task block from the global master node, it records it to the regional task pool and records it, then assign tasks to leaf nodes.

Step4. Execute the tasks. After the leaf node receives the task assignment from the regional master node, it starts to execute the tasks.

Step5. Complete the execution and report. After the leaf node completes the task, it reports to the regional master node. After the regional master node completes the task block, it reports to the global master node, and the global master node updates the task pool.

In the scheduling process, the task prefetching strategy is adopted, so that the computing overhead with the leaf nodes covers the message overhead of the task request, which greatly improves the performance of task scheduling.

3.3 Error Detection

The goal of error detection and fault tolerance is to find faulty nodes in time, and to ensure that the project continues to run correctly at the least cost.

Error detection and fault tolerance are combined with dynamic task scheduling. First, a checkpoint mechanism is introduced into the dynamic task scheduling module. The checkpointing operation is performed by the global master node, and a certain threshold is set for the frequency of the checkpointing operation. When the completed tasks reach the threshold, the completed tasks are recorded. When an inevitable error occurs in the program, fault tolerance can be performed through checkpoints. Additional message types for error detection need to be added, including:

Initiate error detection (IED). IED is initiated by the global master node. When the global master node needs to update the breakpoint file, it sends an error detection message to each regional master node;

Error detection completed (EDC). When the regional master node receives the error detection message sent by the global master node, it sends the same task to all nodes in the region, and after reducing the result, sends a detection completed message to the global master node.

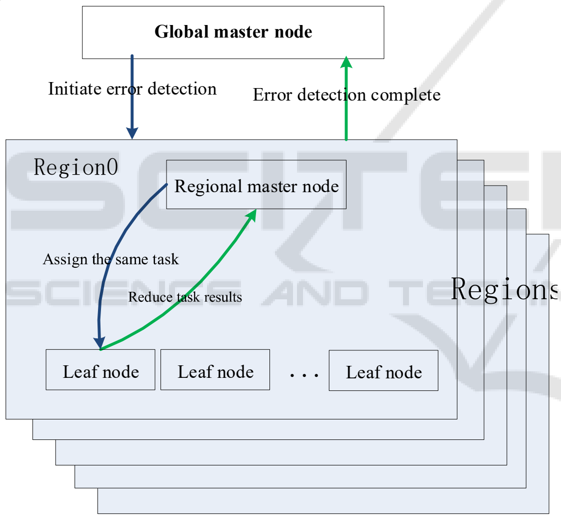

Figure 3. Diagram of error detection process

The mechanism of error detection function is shown in Figure 3, which is divided into the following steps.

Step1. Start error detection. Each time before the checkpoint file needs to be updated, the global master node sends an error detection message to the regional master node.

Step2. Assign the same task to all the leaf nodes in the same region. After receiving the error detection message, the regional master node sends the same task to the leaf nodes in the region.

Step3. Task execution. The leaf node executes the task and records the result.

Step4. Reduce the execution results and check if there exists error node in the region. The regional master node reduces the task results of each leaf node, checks all the results, and identifies whether there is any error node. If an error node is found, it enters the fault tolerance mechanism, otherwise it informs the global master node that the nodes in the region are healthy.

Step5. Finish error detection. After receiving the health information of all regions, the global master node starts to record the checkpoint file.

The reason why the detection is started before the checkpoint is recorded is to detect whether there is any node error between the last checkpoint record and the current checkpoint record.

The cost of error detection is analysed below. The average time of a single task is set to t, the total task number is N, and the number of tasks to be completed by each computing node when a checkpoint is recorded is M. If the task execution of nodes in the area is not synchronized, Because the detection needs to perform a reduction operation, the detection overhead is at most the time overhead of two tasks, that is, the time overhead is at most $2Nt/M$, where M can be adjusted according to system conditions.

3.4 Fault Tolerance

In order to achieve software fault tolerance, each node will maintain a node status queue. Initially, all node information is 0, indicating that the node is healthy. When a node error is found, the information in the status queue will be set to 1, indicating that the node is faulty. When the faulty node is detected by the error detection function, local fault tolerance and global fault tolerance can be used for fault tolerance according to the type of subject.

Local fault tolerance. If the faulty node will not participate in global operations since the time the error occurs, such as global protocol, global synchronization, etc., local fault tolerance can be used. The regional master node that detects the error will notify the leaf nodes in the area.

Global fault tolerance. If the faulty node may participate in global operations, such as global protocol, global synchronization, etc., global fault tolerance needs to be adopted. The regional master node that detects the error will send a signal to the operating system. The operating system will set the node status to fault, and all other nodes are signalled to be notified. After other nodes receive the signal, they will set the status of the node in the node status queue to fault, and all subsequent communication and synchronization operations related to the faulty node will be neglected.

The following describes the main steps of global fault tolerance:

Step1. Call operating system. After the regional master node detects a node error, it calls the operating system function interface to notify the operating system of the faulty node information.

Step 2. Kick out the faulty nodes. After receiving the faulty node information, the operating system marks the faulty node, kicks it out of the job, and sends a fault-tolerant signal to all the other healthy nodes.

Step3. Mark the faulty node. After the healthy node receives the signal from the operating system, it updates the node status queue and marks the faulty node.

Step4. Reassign tasks on the faulty node. The regional master node reclaims all the tasks assigned to the error node since the last update of the breakpoint file and reassigns them. Then it sends the detected exception message and the bitmap of the reclaimed tasks to the global master node.

Step5. Update the bitmap of tasks. The global master node starts the checkpoint file record after updating the completed task information according to the bitmap of the reclaimed tasks.

4 EXPERIMENTAL RESULTS

This section verifies the function and performance of the proposed framework through experiments. The main functions of the framework include task scheduling, error detection and fault tolerance. Error detection is mainly to detect operation errors that cannot be detected by the system, so this experiment mainly verifies the detection of such errors.

4.1 Experimental Verification

The experimental platform is the Sunway TaihuLight supercomputer system (Dongarara, 2016). 64 CPUs are used for testing. Each CPU contains 4 processes, and each CPU has 64GB memory. The CPUs are interconnected by a domestic network and can communicate through MPI.

The experiment uses a task parallel application. The total number of tasks N in the application can be set from 2^13 to 2^15. The evaluation execution time of each task is about 3 seconds. After each task is completed, the task number and execution result will be the output, and the execution result will be reordered according to the task number. The rules for reordering and deduplication are that, for all repeated task results, the last output result is taken as the correct result, and all other results are ignored.

The experimental method is as follows. First use the correct node to execute the task using the static task scheduling to generate the correct results. Then sort the results by task number, and get the correct result. Then execute the dynamic task scheduling framework and get the result, sort the dynamic task scheduling results. Compare the results, check the correctness of the results, record the running time, and check the performance. Then simulate node errors, use the random node and random error mode, enable error detection and fault tolerance, and reorder all output results according to the task number. Finally, the correct results are compared, the correctness is checked, and the running time is recorded.

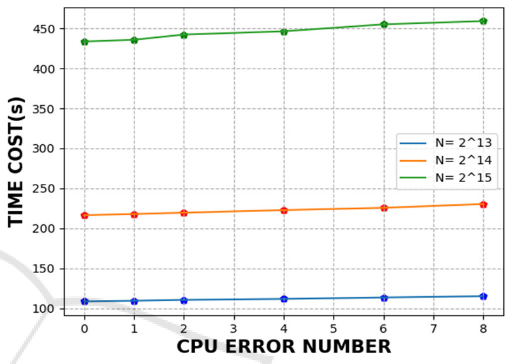

Figure 4. Performance of the example application with different CPU error numbers

4.2 Experimental Results and Analysis

First, the correctness of error detection and fault tolerance is verified. When the error detection and fault tolerance mode is turned on, 1, 2, 4, 6, and 8 CPUs are randomly selected to make random errors. The scheduling results and correct results are consistent, indicating that the error detection and fault tolerance mechanism designed in this paper is effective. After 1, 2, 4, 6, and 8 CPU errors occur, the running time of the entire program for different total task number N is shown in figure 4. With the increase of faulty CPUs, the entire time cost of the program increases gradually, and the increased time is basically linear with the increase of the faulty CPU, which indicates that the time overhead of fault tolerance is very small, and the main overhead is the reduction of computational resources.

Table 2 shows the time cost of dynamic task scheduling using static task allocation, dynamic task scheduling without error detection, and dynamic task scheduling with different checkpoint frequencies(W) with error detection enabled. The total task number is 2^13. Basically, after error detection is enabled, because error detection starts after the checkpoint update is initiated, the error detection frequencies increase gradually as W increases. Correspondingly the time cost increases as W increases, and the time of each error detection is about the time of 2 tasks.

Table 2. Performance of the example application with static task scheduling(a), dynamic task scheduling without error detection(b), dynamic task scheduling without error detection(c) on different checkpoint frequencies (W).

	a	b	c (W=1/2^12)	c (W=1/2^11)	c (W=1/2^10)
Time cost(s)	98.7	99.4	108.6	121.9	142.8
Gap(s)	0	0.7	9.2	13.3	20.9
Detection times	0	0	2	4	8

5 CONCLUSIONS

As the core function of fault management, error detection and fault tolerance are of great significance to improve the reliability of the system. In this paper, an error detection and fault tolerance framework is proposed for task parallel applications on high performance computing systems. The framework realizes the error detection function by periodically leading the computing nodes to perform the same tasks and reducing the results. The error detection time interval can be dynamically controlled according to the environmental conditions. Fault tolerance is realized without stopping the running applications, which effectively reduces the time caused by checkpoint. The experimental results verify the effectiveness of the framework proposed in this paper.

ACKNOWLEDGEMENTS

The work was supported by the National Science Foundation of China (No. 61907034), National Key R&D Program of China(2021YFB0301100).

REFERENCES

Alshboul, M , Elnawawy, H, Elkhouly, R, Kimura, K, & Yan, S. (2019). Efficient checkpointing with recompute scheme for non-volatile main memory. *ACM Transactions on Architecture and Code Optimization, 16(2), 1-27.*

Bronevetsky G, Marques D, Pingali K, Szwed PK, Schulz M. (2004). Application-Level checkpointing for shared memory programs. *In: Proc. of the 11th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. New York: ACM Press, 235-247.*

Chen Z. (2010). Adaptive checkpointing. *Journal of Communications, 5(1):81-87.*

Dinan J., Larkins, D B , Sadayappan, P, Krishnamoorthy, S, & Nieplocha, J. (2009). Scalable work stealing. *Conference on High Performance Computing Networking (pp.1).*

Dongarara J. (2016). Report on the Sunway TaihuLight sytem:UT-EECs-16-742. *Knoxville, USA: University of Tennessee.*

Faisal, Shahzad, Jonas, Thies, Moritz, & Kreutzer, et al. (2018). Craft: a library for easier application-level checkpoint/restart and automatic fault tolerance. *IEEE Transactions on Parallel and Distributed Systems.*

He W, Wei D, Quan J, Wu Wei, Qi F(2016). Dynamic Task Scheduling Model and Fault-Tolerant via Queuing Theory. *Journal of Computer Research and Development 53(6): 1271-1280(in Chinese).*

Mohit K, Sharma S.C., Anubhav G, Singh S.P.(2019). A comprehensive survey for scheduling techniques in cloud computing. *Journal of Network and Computer Applications, 143, 1-33.*

Mukherjee SS, Kontz M, Reinhardt SK. (2002). Detailed design and evaluation of redundant multithreading alternatives. *In: Skadron K, ed. Proc. of the 29th Annual Int'l Symp. on Computer Architecture. Washington: IEEE Computer Society, 99 - 110.*

Oh N, Shirvani PP, McCluskey EJ. (2002). Error detection by duplicated instructions in super-scalar processors. *IEEE Trans. on Reliability, 51(1):63 - 75.*

Reinhardt SK, Mukherjee SS. (2000). Transient fault detection via simultaneous multithreading. *In: Berenbaum A, ed. Proc. of the 27th Annual Int'l Symp. on Computer Architecture. New York: ACM Press, 25-36.*

Reis GA, Chang J, Vachharajani N, Rangan R, August DI. (2005). SWIFT: Software implemented fault tolerance. *In: Conte T, ed. Proc. of the Int'l Symp. on Code Generation and Optimization. Washington: IEEE Computer Society, 243- 254.*

Rotenberg E. (1999). AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. *In: Proc. of the 29th Annual Int'l Symp. on Fault-Tolerant Computing. Madison: IEEE, 84- 91.*

Shye A, Blomstedt J, Moseley T, Reddi VJ, Connors DA. (2009). PLR: A software approach to transient fault tolerance for multicore architectures. *IEEE Trans. on Dependable and Secure Computing,6(2):135-148.*

Wang YZ, Chen X, Ji WX, Su Y, Wang XJ, Shi F. (2016). Task-Based parallel programming model supporting fault tolerance. *Ruan Jian Xue Bao/Journal of Software, 27(7):1789- 1804 (in Chinese).*