Fault Tolerant Scheduling for Parallel Loops on Shared Memory Systems

YIZHUO WANG\textsuperscript{1}, ROSARIO CAMMAROTA\textsuperscript{2} AND ALEXANDRU NICOLAU\textsuperscript{3}

\textsuperscript{1}School of Computer Science
Beijing Institute of Technology
Beijing, 100081 P.R. China
E-mail: frankwy@bit.edu.cn

\textsuperscript{2}Qualcomm Research
San Diego, CA 92121, USA
E-mail: rosario.c@ics.uci.edu

\textsuperscript{3}Department of Computer Science
University of California
Irvine, CA 92697, USA
E-mail: nicolau@ics.uci.edu

While multicore/multiprocessor systems achieve significant speedup for many applications by exploiting loop level parallelism, they also suffer from increased reliability problems as a result of ever scaling device size. This paper addresses the reliability of loop dominated applications, aiming to execute parallel loops efficiently in the presence of various types of hardware faults. In this paper, we present a fault tolerant work-stealing scheme which makes parallel loop execution resilient to hardware faults. A lightweight buffer-commit mechanism is applied in the proposed scheme to ensure the correctness of the re-execution of loop iterations. In addition, we split large failing chunks of loop iterations at runtime to improve load balancing, and a worker thread is discarded when faults occur frequently on it. We evaluated our techniques on a multi-socket multicore system, using a set of loop dominated benchmarks. The proposed scheme achieves the minimum overhead of supporting fault tolerance and optimal load balancing.

**Keywords:** fault tolerance, loop scheduling, work-stealing, multicore and multiprocessor, self-scheduling

1. INTRODUCTION

Along with the increasing of system integration and complexity, future multicore/multiprocessor systems will suffer more frequently from various types of hardware faults that may occur during execution. The external events (e.g. cosmic radiation, power supply noise, device coupling) may cause temporary bit-flipping in memory and logic circuits, that is called transient fault. Irregularities in production process and aging of the circuit may cause permanent faults. Process variation or in-progress wear-out combined with voltage and temperature fluctuations may cause intermittent faults [1]. For the increasing demand on system reliability, tolerance to various hardware faults is bound to become a major aspect in future system design. On the other hand, trends in processor architecture have shown an increasing use of program parallelism to improve performance. Parallel loops account for the greatest percentage of program parallelism, espe-
cially in scientific computing and high performance computing [2]. Therefore, extensive researches have been performed to exploit loop level parallelism (LLP) by scheduling a parallel loop efficiently on multiple processing elements (PEs).

As stated above, fault tolerance and loop scheduling are two critical issues in design of a reliable parallel system which mainly runs loop dominated applications. Many MPSoCs (Multiprocessor System-on-Chip) for aerospace are such systems, in which fault often occurs because of the harsh space environment. Fault tolerance and loop scheduling are normally studied separately and implemented on different system levels, fault tolerance on a hardware level and loop scheduling on a software level. However, as the software-based fault tolerant technique evolves, it is possible to study fault tolerance and loop scheduling in combination and implement them on the same software level.

This paper focuses on loop scheduling and aims to run a parallel loop as fast as possible in the presence of various types of faults. Fault tolerance itself has been studied extensively in two aspects, fault detection and fault recovery. The existing fault detection techniques are generally based on redundancy mechanism in different granularities. Error Detection by Duplicated Instructions (EDDI) [3] introduces redundancy at instruction level. Redundant Multithreading (RMT) techniques (AR-SMT [4], SRT [5], CRT [6], etc.) run a “leading” thread and a duplicated “trailing” thread on different processing elements in parallel, and compare their outputs to detect error. Process-level Redundancy (PLR) [7] 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. Fault recovery is supported with checkpoint-rollback mechanism in most existing fault recovery systems. Checkpoint-rollback can be implemented at system level or application level. System level checkpointing is transparent to users, but too much data needs to be stored in a checkpoint. Application level checkpointing (ALC [8]) only stores the necessary data to recover the program, which reduces the overhead of checkpointing. However, existing ALC techniques are designed for general programs. The characteristics of parallel loops, such as repeating predictable code and restart ability, are not considered. Assuming that checkpoint is set every \( N \) iterations when ALC is applied to a parallel loop, \( N \) iterations will be rolled back if fault occurs on the \( N \)th iteration. But in fact only the last iteration is required to rollback. It implies that checkpoint should be set before every iteration of the loop. However, the checkpointing overhead would be too high in such a way. Although there are some approaches [9, 10] for selecting an optimal checkpoint interval, they do not work at the application level and do not take the natural bounds of loop iterations into account. In addition, for multi-threading applications, checkpoint coordination becomes a thorny issue. But for parallel loops, the worker threads are independent of each other and no coordination is required. The above discussions show that checkpoint-rollback is expensive and inefficient for loop dominated applications. This inspired us to design a low cost fault tolerance mechanism considering characteristics of parallel loops.

A fault tolerant loop scheduling scheme is proposed in this paper. To tolerate transient fault, the most efficient way is to re-execute an iteration directly when fault occurs during the execution of the iteration. However, not every loop is directly re-executable. In this paper, we analyze the re-execute ability of loops and provide approaches to make a parallel loop re-executable. To tolerate permanent fault, a dynamic scheduling scheme is proposed, which supports fault tolerance while enforces locality and load balancing. In
this scheme, iterations of the loop are initially partitioned into $p$ parts (local work queues) assuming there are $p$ processors (worker threads). Each part is further partitioned into a few chunks with decreasing sizes. An idle processor gets a chunk to execute first from its local work queue. When the work queue is empty, the processor attempts to get a chunk from another processor’s work queue. Above process is similar to the work-stealing [11] and affinity scheduling [12]. We incorporate fault tolerance into above dynamic scheduling scheme. When a processor crashes, the remaining chunks in its work queue will be stolen and executed by the other processors. To deal with the chunk which is running on the processor when it crashes, a straight forward method is to re-schedule the chunk to another active processor. This method has the following problems: (a) large crashed chunks cause significant load imbalance; (b) in shared memory system, the results of the finished iterations in the crashed chunk are still available after the processor crashes, therefore it is not necessary to re-execute these iterations. To address above problems, the large crashed chunk is re-partitioned to achieve better load balancing in our scheme and a buffer-commit mechanism is applied to guarantee correctness of the concurrent execution. As such, we make the following contributions in this paper:

1. We propose a unified scheme for loop scheduling and fault tolerance. It is a purely software-based scheme and addresses load balancing and reliability at the application level.
2. We discuss how to make the iterations of a parallel loop re-executable. Loop transformations and a buffer-commit mechanism serve the re-execution purposes.
3. We present FT-WSS, a fault tolerant loop scheduling algorithm, which achieves optimal load balancing by locality-aware work-stealing and dynamically splitting large failing chunks. FT-WSS is more efficient than traditional loop scheduling algorithm combined with checkpointing in practice.
4. We introduce the problem of discarding a processing element in which faults occur frequently. A discarding algorithm is presented for loop dominated applications, and its feasibility is discussed.

The rest of the paper is organized as follows: Section 2 reviews loop scheduling schemes and discusses related work. Section 3 presents the details of our technique to tolerate various types of faults in loop scheduling. Section 4 describes the implementation of our fault tolerant work-stealing scheduling scheme. Experimental results are presented in section 5, and the conclusion is presented in Section 6.

2. BACKGROUND AND RELATED WORK

Loop scheduling is the problem of assigning the $N$ iterations among $p$ processors to execute the loop as fast as possible. The simplest loop scheduling approach is static scheduling which assigns even partitions of loop iterations to the processors. Static scheduling is the most suitable scheme under the assumption that the workload distribution of the loop is uniform, the system is homogeneous and no other jobs run on the system. However, significant load imbalance will arise if any of the above conditions is not met.
In order to achieve load balance in partitioning and scheduling of loops with 
non-uniform iteration spaces, characteristics of the loops are considered in [13-15]. All these 
approaches are profile-based, causing portability problems. In addition, the information 
obtained with profile will be invalid if the global workload of the system changes. 
Therefore, static partitioning is not suitable to complex and volatile runtime environment 
and dynamic scheduling schemes were proposed to address the problem.

Unlike static scheduling that is task-focused, in which iterations of the loop are 
“pushed” to the processors, dynamic scheduling is processor-focused, in which an idle 
processor “pulls” a number of iterations (called a chunk) to execute. The iterations are 
partitioned into chunks to be scheduled. The number of chunks $\beta$ should be larger than $p$ 
and less than $N$. That is the first self-scheduling scheme [16] as $\beta=N$. This scheme assigns 
one iteration to an idle processor each time, so to achieve best load balancing, but 
at the highest scheduling overhead.

Chunk sizes are adjusted in different dynamic scheduling algorithms to tradeoff 
between load balancing and scheduling overhead. The larger chunks raise the less syn-
chronization overhead and the more imbalance. A proper chunk size is obtained by em-
pirical study in fixed size chunking strategies [17]. Polychronopoulos and Kuck pro-
posed a practical method, called guided self-scheduling (GSS) [18], which use chunks of 
decreasing (rather than fixed) size. The initial larger chunks incur very little scheduling 
overhead, whereas the smaller chunks are scheduled at the end of the execution to 
achieve load balancing. GSS has been incorporated in the compilers supporting OpenMP. 
Besides GSS, many self-scheduling schemes were proposed, such as Factoring Self-
Scheduling (FSS) [19] and Trapezoid Self-Scheduling (TSS) [20]. They all generate de-
creasing sized chunks. The difference between them is the function to generate chunk 
sizes (see Table 1). More recently, some adaptive scheduling schemes [21-23] use feedback 
or profiling information to adjust chunk sizes to achieve better load balancing.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Chunk sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSS</td>
<td>250 188 141 106 79 59 45 33 25 19 14 11 8 6 4 3 2 1 1 1 1</td>
</tr>
<tr>
<td>FSS</td>
<td>125 125 125 125 125 62 62 62 62 32 32 32 32 32</td>
</tr>
<tr>
<td>TSS</td>
<td>125 117 109 101 93 85 77 69 61 53 45 37 28</td>
</tr>
</tbody>
</table>

Above dynamic scheduling schemes have not exploit processor affinity to improve 
the data locality. Markatos et al. [12] propose affinity scheduling (AFS) which is locality 
aware. AFS divides loop iteration space into $p$ sub-spaces within $\lceil N/p \rceil$ iterations each. 
The $i^{th}$ sub-space is always placed on the local work queue of processor $i$. An idle pro-
cessor removes a chunk of iterations from its local work queue and executes them, which 
enforces the locality. When a processor’s work queue becomes empty, it steals a chunk 
from another processor’s work queue, which achieves load balancing. There are several 
schemes [23, 24] which are similar to AFS. We call them work-stealing loop scheduling 
(WSS) collectively. Work-stealing is widely used in parallel programming models, such 
as Cilk [11], Intel TBB [25], X10 [26], Kaapi [27] and Microsoft TPL [28] for task 
scheduling. For parallel loops, a chunk of iterations is scheduled as a task in work-
stealing algorithms. Compared with previous work-stealing technique, our technique incorporates fault tolerance into the scheduling scheme and adopts runtime chunk splitting to improve load balancing.

In the context of loop scheduling, a few schemes [29-31] support fault tolerance in practice. In [29], fault tolerant loop scheduling scheme is proposed based on self-scheduling algorithms in Grid environment. In [30], regular loops are transformed and carefully scheduled to detect the faults, but no recovery support. Cilk-NOW [31], an implementation of the Cilk runtime system for networks of workstations, supports fault tolerance in work-stealing scheduling scheme like ours. In Cilk-NOW, if a processor crashes, the other processors automatically redo any work that was lost in the crash. Checkpointing mechanism is applied to support fault recovery in Cilk-NOW, unlike our work in which a buffer-commit mechanism is applied and the re-execute ability of loops is analyzed so that only the data with anti-dependence is buffered. Feng and Gupta et al. also use copying and committing in a parallel programming model, SpiceC [32]. But they do not address the problem of fault tolerance. In addition, we allocate buffers dynamically using a memory pool and we copy very less data than SpiceC. Ke et al. [33] use copying and committing for safe parallel programming. They believe copy-commit is the third method to exchange information between processes, besides shared memory and message passing. They consider the safety during parallelization, not the hardware fault tolerance.

Although some self-scheduling schemes [12, 18, 32] take fault tolerance into account, their implementations of fault tolerance are straightforward. The whole chunk in which fault occurs is re-assigned to an active processor to be re-executed. In our scheme, the failing chunk is not re-executed as a whole, but re-partitioned into a few smaller chunks. Such runtime splitting improves load balancing.

3. FAULT TOLERANCE TECHNIQUES FOR PARALLEL LOOPS

In this section, we present the fault tolerance techniques which are applied in our loop scheduling algorithm.

3.1 Making a Loop Re-executable

Transient faults are temporary bit-flips in memory and logic circuits (so called soft errors). In this paper, we assume that transient fault is detected with an existing technique, such as EDDI [3] or CRT [5]. The program jumps to an error handler when a fault is detected. In the error handler, the program rolls back to the last checkpoint if checkpointing mechanism is applied. That is costly for a loop as we discussed in Section 1. In our loop scheduling scheme, the program jumps to the beginning of a loop iteration to re-execute it directly when a transient fault occurs during the execution of the iteration. This brings up a problem that the loop iterations must be re-executable, that is, the results should be correct after re-executions.

Let us analyze whether a loop is directly re-executable or not with the help of an example. For the loop in Fig. 1 (a), the result will be always correct, regardless of how many times it is executed. In contrast, the loop in Fig. 1 (b) is not directly re-executable...
because the first execution modifies the value of b[i] and the second execution would load the modified value into a[i] and e[i]. b[i] is read by the statements (1) (2) and written by the statement (3), which incurs anti-dependence in the loop body. The following theorem is proposed to determine whether the loop can be directly re-executed or not.

**Theorem 1:** The loop is directly re-executable if and only if there is not anti-dependence in the input data set of the loop body.

We classify the data involved in the loop body into two categories:

- The input data set $\Phi$. Any data which is read first in the loop body belongs to $\Phi$. $\Phi$ consists of the read-only data and the write-after-read (WAR) data\(^1\).
- The output data set $\Psi$. Any data which is written first in the loop body belongs to $\Psi$. $\Psi$ consists of the write-only data and the read-after-write (RAW) data\(^1\).

For a variable $\gamma \in \Psi$, the value of $\gamma$ is initialized with another variable belonging to $\Phi$ or a constant value. $\gamma$ will be correct if the value used to initialize it does not change in many times re-executions. Therefore, the correctness of the results of re-execution is not dependent on $\Psi$, but dependent on $\Phi$. Clearly, read-only data is not related to the re-execute ability of the loop because the values of read-only variables will never change regardless of how many times the loop is executed. The only remaining case is WAR in $\Phi$, that is, anti-dependence. If a variable belonging to $\Phi$ is write-after-read, it will have a new value after the first execution, and the loaded value will not be the original value in the second execution, which produces incorrect results. Therefore, the loop is not re-executable if anti-dependence exists in $\Phi$, and vice versa.

We know that anti-dependences could be removed by renaming, that is, buffering, of variables. The loop in Fig. 1 (b) can be transformed to the loop in Fig. 1 (c) to eliminate the anti-dependences. The values of the array b are buffered into b1 before the execution of the loop. All the variables b[i] before the first writing, the statement (3), are copied (b, b');

```c
for (i=0; i<n; i++)
    copy (b1, b);
(a)
```

```c
for (i=0; i<n; i++)
    (1) a[i] = b[i];
    (2) e[i] = b[i];
    (3) b[i] = c[i];
    (4) g[i] = b[i];
    (5) b[i] = f[i];
    ...
(b)
```

```c
for (i=0; i<n; i++)
    (1) a[i] = b1[i];
    (2) e[i] = b1[i];
    (3) b[i] = c[i];
    (4) g[i] = b[i];
    (5) b[i] = f[i];
    ...
(c)
```

```c
copy (b1, b);
(d)
```

Fig. 1. Parallel loops: (a) no anti-dependence; (b) anti-dependences in the loop body; (c) no anti-dependence in the input data set; (d) Illustration of the buffer-commit method making a loop re-executable.

---

\(^1\) WAR (R-W) represents that the data is read firstly and then written in the loop body. The accessing order of the data does not have to be R-W exactly. It could be R-W-R, R-W-W, etc. WAR (R-W) / RAW (W-R) only indicates the order of the first two accesses to the data.
renamed as b1[i] in the loop body. Thus, the correct value of b[i] will be loaded from b1[i] when the iteration is re-executed after transient fault occurs. The anti-dependence checking and the above loop transformation can be implemented at compile time.

Above transformation makes the loop in Fig. 1 (c) directly re-executable under the assumption that two executions are in time order. We call it sequential re-execution. Our transient fault tolerant mechanism is in this case because the iteration is re-executed by the same processor when fault occurs on it. But in the case of concurrent re-execution, that is two processors execute a same iteration simultaneously, above transformation cannot guarantee the correctness of the results. For example, the variable b[i] is W-R-W in the statements (3), (4) and (5) in Fig. 1 (c). Sequential re-execution has no problem because b[i] is in $\Psi$ of the loop (see proof of Theorem 1). But concurrent re-execution will produce incorrect results because of the data hazard of b[i]. For details, if processor $P'$ executes the statement (4) immediately after processor $P$ executes the statement (5) of a same iteration, the value of b[i] which $P'$ stores into the memory at last would be wrong because $P'$ loads the value of b[i] which has been changed by $P$ in the statement (5). We call this the concurrent re-execution error. To avoid such error, we propose a buffer-commit mechanism which stores the data with anti-dependence in a buffer and commits it after the execution of the loop. As shown in Fig. 1 (d), different processor writes the value of b[i] to different buffer in the statement (3). After that any access to b[i] is changed to the per-thread private buffer b'[i]. The buffer b'[i] is copied to b[i] after the execution of the loop or a chunk of iterations. We call this a commit operation.

It is time consuming to buffer and commit the data and additional synchronization costs are required. Therefore, we give an optimized implementation to reduce the number of the iterations which is re-executed concurrently in our scheme. See Section 4 for details.

Remark: Techniques described above can not only apply to parallel loops but also any section of a program.

3.2 Work-Stealing Loop Scheduling with Initial Partitioning

The iterations of the loop are assigned to multiprocessors to be executed in parallel. When one of the processors crashes, the other processors should help it to finish the remaining work. Therefore, a dynamic scheduling scheme is required to re-schedule the remaining iterations at runtime. Work-stealing is a widely used dynamic scheduling scheme which can achieve optimal load balancing. Thus, we establish the fault tolerance approach on the base of work-stealing scheduling.

A pseudocode description for the work-stealing scheduling (WSS) we adopted can be found in Algorithm 1. We assume $N$ iterations and $p$ processors. There are corresponding $p$ worker threads and each thread is bound to a processor. First, WSS partitions the iterations of the loop into $p$ parts (or batches). We call it the initial partitioning. See the $\text{Initialize}(N, p, k, \theta)$ function in Algorithm 1. $k$ and $\theta$ are user-inputted parameters controlling chunk sizes, which will be discussed later. In traditional work-stealing algo-

---

2 Cores and hardware threads are all processor elements (PEs). A processor refers to a PE in the next article.
rithms, there is not initial partitioning because traditional work-stealing algorithms are designed for task parallel divide-and-conquer applications in which tasks are generated at runtime and cannot be partitioned statically before running. However, for a parallel loop, the loop bounds are known before running the loop. Thus, the loop can be easily partitioned in iterations. Consecutive iterations are assigned to a processor in initial partitioning phase, which improves data locality.

Algorithm 1: Work-stealing loop scheduling

\( p \) : number of processors;
\( T = \{ T_1, T_1, \cdots, T_p \} \) : worker threads;
\( Q_i \) : local work queue of \( P_i \) (i.e., \( T_i \));
\( L_i \) : a lock to protect \( Q_i \);

The master thread:

Initialize(\( N, p, k, \theta \)) // \( k \) and \( \theta \) are user-inputted constants.
\( N \) iterations is partitioned into \( p \) parts \( I_1, I_2, \ldots, I_p \).

\[
\text{for } (i = 0; i < p; i++) \{
\text{R} = u_i - l_i; \quad \text{// The remaining iterations in a part.}
\text{while } (R > \theta )\{
\text{Take } R/k \text{ iterations of } I_i \text{ as a chunk } C.
\text{Enqueue } C \text{ into } Q_i.
\text{R} = R - R/k;
\}
\text{Take the last } R \text{ iterations as a chunk and enqueue it.}
\}
\}
\text{for } (i = 0; i < p; i++) \{
\text{create_thread } (T_i);
\text{Set } T_i \text{ running on processor } P_i;
\}

Wait for all threads to complete.

The \( i^{th} \) worker thread \( T_i \):

Lock (\( L_i \));
\[
\text{if } (Q_i \neq \text{NULL })\{
C = \text{Dequeue } (Q_i);
\text{Unlock } (L_i);
\}\text{else}\{
\text{Unlock } (L_i);
C = \text{Get_remote_chunk } ();
\text{if } (C = \text{NULL}) \quad \text{exit } ();
\}
\text{Execute } (C);
\]

The workload distribution of the loop and the capabilities of processors are two major factors impacting load balancing of the initial partitions. The workload distribution can be obtained via loop profiling. However it is time consuming. In our scheduling algorithm, we only take the capabilities of processors into account. Let \( l_i \) and \( u_i \) denote the lower and upper bounds of the part assigned to processor \( P_i \). We use \( \lambda_i \) to represent the
capability of the processor $P_i$ ($i=1,2,\ldots,p$). Then, we normalize $\lambda_i$ with $\lambda_1$. For instance, $\lambda_1=1$ and $\lambda_2=2$, which means that the execution time of the same workload on $P_1$ is twice as that on $P_2$. Subsequently, the loop bounds of each part can be calculated using the following equations.

$$l_i = 1; \quad l_{j+1} = u_j + 1; \quad u_j = \left\lceil \frac{\sum_{i=1}^{j} \lambda_i}{\sum_{i=1}^{\lambda_p} N} \right\rceil, \quad j = 1, 2, \ldots, p-1; \quad u_p = N.$$  \hfill (1)

The initial partitioning makes the workload equally distributed onto the processors approximately. It is a static load balancing procedure. After that, each part is further partitioned into several chunks according to the following equations and work-stealing is applied at runtime to achieve dynamic load balancing.

$$R_0 = u_j - l_j, \quad R_{i+1} = R_i - C_i, \quad C_i = \begin{cases} \left\lceil \frac{R_i}{k} \right\rceil & (R_i \geq \theta) \\ \frac{R_i}{\theta} & (R_i < \theta) \end{cases}$$  \hfill (2)

The $i$th part is namely the local work queue ($Q_i$) of the worker thread $T_i$. Each worker thread obtains work by removing a chunk from its local work queue. When the local work queue is empty, the thread will try to steal a chunk from another worker thread by calling Get_remote_chunk. Get_remote_chunk checks the work queues of the other threads. If there is a non-empty work queue, a chunk in it will be stolen. Note that every work queue is protected with a lock associated with it during checking and stealing.

The constants $k$ and $\theta$ are parameters of our algorithm. They are used to tradeoff between load balancing and the scheduling overhead. The number of the iterations that a processor fetches each time decreases while $k$ is increased. Thus, the number of chunks increases which results in the increase of the scheduling cost, while the load balancing is improved because the last few chunks have less iterations. In our experiments, we set $k$ to 2. Then the sorted chunk sizes of all parts are same with the chunk sizes generated with FSS on the entire iteration space. This indicates that WSS can achieve same load balancing with FSS, but WSS has better data locality than FSS. Previous work in [34] shows that the chunk size should range from $R/2p$ to $R/p$ in a self-scheduling scheme with a central work queue to have reasonable load imbalance and synchronization overhead. $R$ is the number of the remaining iterations. Therefore, the range of $k$ should be $[1, 2]$ in our algorithm. The threshold $\theta$ is the minimal size of a chunk. The best value of $\theta$ is machine and application dependent. Similar parameters and the selection of their values can be found in self-scheduling schemes, such as GSS [18] and AFS [12].

### 3.3 Dynamic Re-partitioning of Large Faulty Chunk

Work-stealing has native fault tolerance capability. If a processor crashes, the other processors will automatically steal and redo the work of the crashed processor. How to deal with the faulty chunk is a critical issue in work-stealing loop scheduling on systems with faults. Let us illustrate this issue and our solution with the help of an example. Given $p=4$ and $k=2$, the ideal running process, that every iteration has the same execution time and all the threads start working at the same time, is shown in Fig. 2. If no fault
occurs, all the processors will complete the execution of the loop at the time $t$. If processor $P_1$ crashes during running the chunk $a_1$, $P_2$-$P_4$ will steal $a_2$-$a_5$ to execute after $t$ with work-stealing scheme. To deal with the chunk $a_1$, a straightforward solution is to reschedule it to an idle processor that is $P_1$ as shown in Fig. 2 (a). Then the completion time of the loop will be $t'$. This solution brings about significant load imbalance. An improvement approach is shown in Fig. 2 (b). Chunk $a_1$ is re-partitioned into a few chunks ($a_{1,1}$ to $a_{1,4}$). These chunks are next scheduled with WSS. Then the completion time will be $t''$ that is earlier than $t'$ and the workloads on $P_2$, $P_3$ and $P_4$ are more balancing.

In shared memory systems, the iteration index is stored in memory, so the other processors could get the index of $P_1$ after it crashes during running the chunk $a_1$ in Fig. 2. The output data of the finished iterations in $a_1$ has been stored correctly. Therefore, it is not necessary to re-execute the whole chunk $a_1$. Let $R$ denote the number of remaining iterations in $a_1$ and $p_{\text{crash}}$ denote the number of the processors which have crashed. We partition the remaining iteration of $a_1$ into new chunks according to the following equations and schedule them with work-stealing scheme.

$$\begin{align*}
R_0 &= R, \\
R_{i+1} &= R_i - C_i, \\
C_i &= \left\lfloor \frac{R_i}{(p - p_{\text{crash}})} \right\rfloor \quad (R_i \geq \theta) \\
&= \frac{R_i}{(p - p_{\text{crash}})} \quad (R_i < \theta) \text{ or } p_{\text{crash}} = 1
\end{align*}$$

Work-stealing scheme gives rise to the following problem: a same iteration may be executed by different processors concurrently if a processor crashes and recovers quickly, that is, a processor is intermittently unavailable. The proposed buffer-commit mechanism can deal with this problem. See the description of this mechanism in Section 3.1 and the details of its implementation in Section 4.

3.4 Processor Discarding

For the type of intermittent faults which are repeats of transient fault, a discarding mechanism is used to determine whether a processor on which transient faults frequently occur should continue to be used or not. For parallel loops, the frequency of transient faults can be expressed in fault recurrence interval (the number of iterations). We implement a discarding algorithm (Algorithm 2) in our fault tolerant scheduler. The basic idea of the discarding algorithm is that a processor would be discarded if the intervals between the last few faults on this processor are all smaller than a preset threshold $\Delta$. In
Algorithm 2, counter $C_{\text{iter}}$ is incremented by one whenever the worker thread completes an iteration. When a transient fault occurs, the error handler increments counter $C_{\text{faults}}$ if $C_{\text{iter}}$ is less than $\Delta$, which indicates that the interval between the last two faults is short. If $C_{\text{faults}}$ is greater than a preset value $d$, faults occur $d$ times in succession with an interval smaller than $\Delta$. That means the fault frequency is high. Then, the worker thread is stopped to be used.

### Algorithm 2  Processor discarding algorithm

- $C_{\text{iter}}$, $C_{\text{faults}}$: two counters for each processor, which are initialized to 0;
- $\Delta$, $d$: thresholds for counter $C_{\text{iter}}$, $C_{\text{faults}}$.

**Worker thread:**
Get a chunk of loop iterations according to scheduling algorithm.
for each iteration do
  Execute the loop iteration.
  $C_{\text{iter}}$ ++;
end for

**Error handler:**
if $C_{\text{iter}} < \Delta$ then
  $C_{\text{faults}}$ ++;
else
  $C_{\text{faults}}$ = 0;  // Reset the counter $C_{\text{faults}}$ if the interval is long.
end if

if $C_{\text{iter}} = 0$;  // Reset the counter $C_{\text{iter}}$ whenever a fault occurs.
if $C_{\text{faults}} \geq d$ then
  Discard this worker thread.
end if

### 4. FAULT TOLERANT LOOP SCHEDULING ALGORITHM

Based on the techniques in Section 3, we implemented the fault tolerant loop scheduling in Algorithm 3, called *FT-WSS* (Fault Tolerant Work-Stealing loop Scheduling). To support direct re-execution after transient fault occurs, the loop is transformed with the technique in Section 3.1 in advance. Then the iterations of the loop are scheduled with FT-WSS to $p$ worker threads dynamically. The same parts with WSS (in Algorithm 1) are not shown in Algorithm 3, such as the initial partitioning of the batches and chunks.

In FT-WSS, the worker thread $T_i$ checks its local work queue $Q_i$ at first. If $Q_i$ is not empty, a chunk $C$ is dequeued from $Q_i$ and assigned to $T_i$ to execute. We use $E_i$ to denote the chunk running on $T_i$. $E_i$ is set to $C$ before the execution of the chunk and $E_i$ is set to NULL after the execution. If $Q_i$ is empty, $T_i$ will try to steal a chunk from another processor (worker thread), called “victim”. There are three methods to select the victim:

1. **Random choice:** The worker randomly selects a victim. If the victim has no work, the thief worker moves to the next worker. The main advantage of this method is that the victim choice does not require more information than the total number of processors. Therefore, it is the minimum-cost implementation of victim selection.
(2) Selecting the most loaded processor: The thief worker steals a chunk of iterations from the most loaded worker, which avoids unnecessary inter-processor communications caused by randomly probing work queues. Set a counter for each processor. Upon the completion of an iteration, the counter is incremented. Then the most loaded processor could be found by comparison of the counters of the processors.

(3) Locality aware choice: Considering the increasingly complex memory hierarchy on modern shared memory systems, an efficient scheduling should be locality aware by allowing a thread to steal work firstly from the threads within the same chip before across the chips. This method is recently used in hierarchical work-stealing algorithms [35, 36] and is provably efficient.

In FT-WSS, we use the locality aware choice. Get_remote_chunk checks the work queues of the other threads in order of locality-aware choices in Algorithm 3. If Get_remote_chunk returns NULL, all the work queues are empty. \( T_i \) will exit in normal scheduling algorithm without the supporting of fault tolerance, such as WSS in Algorithm 1. Then the loop will never complete if another processor \( P_j \) crashes while normal scheduling algorithm is applied. To tolerate such fault, thread \( T_i \) does not exit in FT-WSS when no chunk exists in any work queue. Instead, \( T_i \) looks for another worker thread \( T_j \) (processor \( P_j \)) which is still running a chunk \( C' \). Find_unfinished_thread realizes the search operation by checking the flag \( E \) of each worker thread in Algorithm 3. Based on the discussion in Section 3.3, the chunk \( C' \) should be re-partitioned if its size is large. Note that we do not know whether \( P_j \) crashes or not. Therefore, the concurrent re-execution error mentioned in Section 3 may occur. Thus, we need to determine which iteration may be executed concurrently. In shared memory system, \( T_i \) can read the current iteration index of \( T_j \), denoted as \( \hat{I} \). A flag \( S_j \) is set before the index \( \hat{I} \) is read. Each worker thread has such a flag. The code of the execution of a chunk \( C \) is shown in Fig. 3. \( C_l \) and \( C_u \) are the lower and upper bounds of iteration index in chunk \( C \). From the Fig., we can see that \( S_j \) is used to suspend the thread \( T_j \) if processor \( P_j \) does not crash. The remaining iterations of \( C' \) range from index \( \hat{I}+1 \) to \( C_u' \). \( T_j \) will not continue executing these iterations because it is blocked by the flag \( S_j \). The iterations from \( C_l' \) to \( \hat{I}-1 \) were finished by \( P_j \). So we set \( C_u' \) to \( \hat{I}-1 \). Thus, only the iteration \( \hat{I} \) may be executed concurrently by \( T_i \) and \( T_j \) in the case that \( P_j \) does not crash or recovers very quickly. Because the concurrent re-execution error may occur for the iteration \( \hat{I} \), we have different implementation of FT-WSS for different loop.

If there is not anti-dependence existing in the loop body, the concurrent re-execution will not make the results incorrect. Therefore, we spawn new chunks from the iter-

---

```c
k = C_l;
while (1) {
    while (S_j == 1);
    if (k > C_u) 
        break;
    Original loop body...
    k = k + 1;
}
```

Fig. 3. The code of the execution of a chunk.
tions that range from index $I$ to $C_u'$ directly. The new chunks are enqueued into $Q_j$ and scheduled with work-stealing scheme.

If there is anti-dependence, we spawn new chunks from the iterations that range from index $I+1$ to $C_u'$ and enqueue them into $Q_j$. Because the iteration $I$ may be executed concurrently, we adopted the buffer-commit mechanism which is implemented in brackets in Algorithm 3. In the worker thread, the buffer is allocated firstly, that is to store the data with anti-dependence in the iteration $I$. Let $z$ denote the size of the buffer needed for an iteration. Then the total buffer size will not be larger than $p \times z$ normally, that is the case each processor crashes once. But the total size will be larger than $p \times z$ if some processors crash and recover many times during the execution of the loop. So we use memory pool to manage the buffers. The initial pool size is $p \times z$. After the buffer needed for the iteration $I$ is allocated, the worker thread executes $I$ and writes data into the buffer. Finally, the index $I$ and the entrance of the buffer is stored in a list $ComList$. In the master thread, all the buffered data is committed (copied) to their original memory addresses after the worker threads complete the execution of the loop. The pair $<I, Buffer>$ is enough for committing the data of the iteration $I$ because the target addresses can be obtained with the index $I$. That is the benefit of the regular loop. The reasons for committing the data in the master thread are following: (a) The data of one iteration will never be read or written by another iteration because the loop is DOALL loop. Therefore, the committing can be done after the execution of the loop. (b) The results may be incorrect if committing the data in the worker threads. For example, $T_i$ recovers and continues the execution of $I$ while $T_j$ commits the data of $I$. Data hazard occurs in this case because $T_i$ accesses the original data directly.

Algorithm 3  Fault tolerant work-stealing loop scheduling

$p$: number of processors;
$T = \{T_1, T_2, \ldots, T_p\}$: worker threads;
$Q_i$: local work queue of $P_i (T_i)$;
$L_0$: a lock to protect $Q$;
$C_l, C_u$: lower and upper bounds of iteration index in chunk $C$.

The $i^{th}$ worker thread $T_i$:

Lock ($L_0$);
if $Q_i \neq NULL$ then
$C = Dequeue (Q_i)$;
Unlock ($L_0$);
else
Unlock ($L_0$);
$C = Get\_remote\_chunk ()$;
if $C = NULL$ then
$j = Find\_unfinished\_thread ()$;
if $j = 0$ then
$All\_job\_finished = 1$;
exit();
else  // $P_j$ may crash on $C'$
Lock ($L_0$);
$S_j = 1$;
$I = Current\_position (P_j)$;
// re-partition the iterations from $I$ (or $I+1$) to $C_u'$;
NewChunks = Re_partition (Ī+1 or Ī, C*);
Enqueue (NewChunks, Q);
C* = Ī - 1;  S* = 0;
C = Dequeue (Q);
Unlock (L*);
end if
end if
end if
Ei = Ī;
Execute (Ī);
if anti-dependence exists then
Allocate Buffer to store the data with anti-dependence.
Execute the iteration Ī using Buffer.
Add <Ī, Buffer> into ComList.
end if
Ei = 0;

The master thread:
...//same with WSS
Wait for All_job_finished = 1;
if ComList ≠ NULL then
Commit the data in the buffers.
end if
...

5. EXPERIMENTS

In this section, the effectiveness of the FT-WSS scheduling algorithm is evaluated by running a set of loops on a real machine. Table 2 gives the detail of the loops used in our experiments. JI and TC are loop nests with sequential outer loop and parallel inner loop. The rests are outmost parallel loops. MM and MT are natively implemented, no tiling or other optimizations applied. For MT, equake and ammp, anti-dependences exist in their loop bodies. Therefore, buffering (like checkpointing) is required to achieve fault recovery.

<table>
<thead>
<tr>
<th>Loop</th>
<th>Source &amp; Description</th>
<th># of Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>JI</td>
<td>Jacobi Iteration[12,23], loop nests</td>
<td>2000 (inner)</td>
</tr>
<tr>
<td>TC</td>
<td>Transitive Closure[12,23], loop nests</td>
<td>2000 (inner)</td>
</tr>
<tr>
<td>milc</td>
<td>433.milc, SPEC 2006, quark_stuff.c, 1523</td>
<td>160000</td>
</tr>
<tr>
<td>mgrid</td>
<td>172.mgrid, SPEC 2000, mgrid.f, 189</td>
<td>128</td>
</tr>
<tr>
<td>MM</td>
<td>Matrix Multiplication</td>
<td>3200</td>
</tr>
<tr>
<td>MT</td>
<td>Matrix Transposition</td>
<td>3200</td>
</tr>
<tr>
<td>equake</td>
<td>183.equake, SPEC 2000, quake.c, 462</td>
<td>7280</td>
</tr>
<tr>
<td>ammp</td>
<td>188.ammp, SPEC 2000, rectmm.c, 405</td>
<td>160000</td>
</tr>
</tbody>
</table>

The experiments are conducted on a Sun T5140 server, which has 2 chips. Each chip has 6 cores, and each core supports 8 hardware threads, making the total number of hardware threads 96. Table 3 lists the machine details. Next, we report our experimental results. Each result is the average of 100 executions.
5.1 Fault-Free Performance

In this experiment, we first run the benchmarks with three scheduling algorithms, GSS, WSS and FT-WSS, in a fault-free environment. GSS is a widely used loop scheduling algorithm adopted in OpenMP. WSS is the work-stealing algorithm for loop scheduling (Algorithm 1). GSS and WSS show the state-of-the-art performance of parallel loop scheduling without fault tolerance supporting. For each loop and each scheduling algorithm, we run experiments varying the number of worker threads from 2 to 32. Fig. 4 shows the execution times normalized with respect to GSS. For the first five loops, there is no anti-dependence in their loop bodies. Therefore, the data accessed in the loops does not need to be copied before the execution of the loops and the buffer-commit mechanism is not required. The penalty of introducing fault tolerance is 6.7% increase of the execution time in average for these loops, comparing FT-WSS to WSS. We believe this performance penalty can be mainly attributed to the following two factors: (a) the increase in task steals due to the dynamic re-partitioning, and (b) additional synchronization and task queue operations for the re-partitioning. However, the execution time of a parallel program running on a real machine is influenced by a lot of factors. It is difficult to Fig. out the exact reasons why FT-WSS performs worse than WSS. For MT, equake and ammp, the loops are transformed with the approach described in Section 3.1. The data copying in the transformation makes the overhead of FT-WSS higher than it for the other five loops. The cost of data copying is dependent on the loop itself. More data with anti-dependence brings more cost. Although the copying is time consuming, it is inevitable to support fault recovery.

GSS shows its native disadvantage in Fig. 4. For MT, WSS outperforms GSS by 26%. MT has a decreasing triangular workload distribution. Thus, the first few iterations...
are much heavier than the others. GSS incurs the load imbalance because of putting too many iterations in the first few chunks. Compared to GSS, WSS has smaller chunk sizes and improves locality of reference by exploiting processor affinity. We expected WSS could achieve much better performance than GSS. However, we can see from Fig. 4 that WSS achieves very little performance improvement except for MT. The results suggest that the selection of dynamic scheduling algorithm is not very important if the workload distribution of the loop is not significant uneven like MT.

We denote the time cost of copying data before the execution of the loop as “CP Overhead”. As mentioned above, this overhead is inevitable for fault recovery. For equake and ammp, anti-dependences still exist in their loop bodies after the data copying and loop transformation in Section 3.1. Therefore, the runtime buffer-commit mechanism is required to avoid the concurrent re-execution error. The time cost of the buffering and committing is denoted as “B-C Overhead”. We break down the execution times of FT-WSS into “CP Overhead”, “B-C Overhead” and the remaining in Fig. 5. Because all the anti-dependences are eliminated after the data copying and modification for MT, there is not “B-C Overhead” in Fig. 5 (c). In addition, the data is buffered in parallel using available worker threads for MT. Therefore, the “CP Overhead” decreases as the number of threads increases in Fig. 5 (c). Fig. 5 shows that the “B-C Overhead” is negligible, which is less than 0.3% of the execution time in Fig. 5 (a) and less than 0.01% in Fig. 5 (b). It indicates that the proposed buffer-commit mechanism is inexpensive and the fault tolerance overhead introduced by FT-WSS is low.

The above experiment compares FT-WSS with non-fault-tolerant scheduling schemes. Next, we compare FT-WSS with the modified GSS (denoted by CR-GSS) and work-stealing (denoted by CR-WSS) which use checkpoint/restart technique to support fault recovery. In CR-GSS and CR-WSS, the failing chunk running on a crashed processor will be re-executed completely by another processor, that is suggested in [11, 12, 32]. A checkpoint which consists of the necessary data for the recovery is saved into the shared memory before executing each chunk. Fig. 6 shows the fault-free performance of CR-GSS, CR-WSS and FT-WSS. The experiment setting is the same as the previous setting for the comparison between GSS, WSS and FT-WSS. The results are normalized with respect to FT-WSS. As we can see, FT-WSS outperforms CR-GSS and CR-WSS, especially for MT, equake and ammp. The average speedups are 7.1% and 2.1% respectively.
5.2 Fault Tolerance Evaluation

CR-GSS and CR-WSS represent the state-of-the-art loop scheduling schemes supporting software-based fault tolerance. We compare FT-WSS with them to demonstrate the benefits brought by our technique. To simulate permanent faults of processors, we randomly killed one worker thread during the parallel execution of a loop when 2 to 4 threads are used, and 2 threads are killed when 8 to 32 threads are used. Fig. 7 shows the average execution times normalized with respect to CR-GSS. Compared to CR-GSS, FT-WSS achieves 17% speedup and CR-WSS achieves 9% speedup in average. Hence FT-WSS outperforms CR-WSS by about 8%. The performance gains of FT-WSS can be attributed to the runtime splitting of the large failing chunks and avoiding checkpointing. The runtime splitting makes FT-WSS achieve better load balancing. The implementation of checkpointing in CR-GSS and CR-WSS is in application level and the size of the checkpoint data is minimized. The total size of checkpoints in CR-GSS or CR-WSS is almost same with the size of the buffered data in FT-WSS. Therefore, we believe the runtime splitting is a major contributor to the performance gains of FT-WSS. Moreover, the maximum speedup (FT-WSS over CR-GSS) of 60% is achieved on MT in Fig. 7, indicating FT-WSS can achieve much better performance on non-uniform loop.

In fault-free environment, WSS has similar performance to GSS (see Fig. 4). However, CR-WSS performs better than CR-GSS for all the eight benchmarks when fault occurs. The reason is that GSS algorithm produces some large chunks at the beginning of the execution and the chunks generated by WSS is much smaller than these chunks of GSS. If a fault occurs during the execution of a chunk, both CR-GSS and CR-WSS restart the chunk from a checkpoint stored before running the chunk. Therefore, smaller chunk brings lower cost of fault recovery.
To evaluate the benefits of the runtime splitting, we remove the \textit{Re\_partition} procedure in Algorithm 3. The remaining iterations from $\tilde{I}$ (or $\tilde{I}+1$) to $C_{\tilde{I}}'$ form a new chunk, which is no longer split and enqueued to a task queue directly. The modified algorithm is denoted by TWSS. We run TWSS for each benchmark with 2 to 32 threads. In each execution, we also randomly kill one or two threads when 2-4 threads or 8-32 threads are used. The execution times are reported in Fig. 8. Compared to TWSS, FT-WSS achieves 7.7% speedup in average. To be sure, TWSS and FT-WSS are all based on work-stealing dynamic scheduling, and they address load balancing, locality and fault tolerance at same time. Therefore, the speedup is attributed to the runtime splitting.

![Fig. 8. Performance comparison of FT-WSS with TWSS.](image)

In above experiments, we only killed one or two worker threads to simulate the injections of permanent faults. To evaluate the performance of the fault tolerant scheduling schemes in different probabilities of permanent fault occurrence, we kill 1 to 8 threads respectively when mile, MT and ammp are executed with 16 threads in parallel. The completion times are reported in Fig. 9. Note that FT-WSS does not distinguish between the real processor crash and the case that the processor is temporarily unavailable because it is occupied by other jobs. In above two cases, FT-WSS takes the same strategy to finish the job as soon as possible. Therefore, the results in Fig. 9 also show the performance of FT-WSS in heavy loaded system\textsuperscript{3}. From the figure, we see that FT-WSS performs better than CR-GSS and CR-WSS in all cases except for ammp when 4 threads are killed. We also observe that as the number of killed threads increases, the execution times increase not much in general. It is because we do not kill threads at the beginning of the execution, but kill threads randomly during the execution and report average result of one hundred runs. In addition, the three schemes are all dynamic scheduling schemes. When a thread is killed, the work on it will be completed by other threads in parallel. Therefore, the completion time does not increase linearly. Compared with CR-GSS and CR-WSS, the curves of FT-WSS show little variance, indicating that FT-WSS is more stable than the others and is more applicable to volatile runtime environment.

Transient fault recovery is supported by re-executing the failing iteration directly. To make it re-executable, the data is copied before the execution of the loop. The overhead of data copying is presented in Fig. 5 and it should not change in the environments with faults. In addition, transient fault detection is not the objective of our work. Any

---

\textsuperscript{3}It is uncommon that half of the processors crash. But it is common that half of the processors are busy on other jobs, especially in multi-user systems.
Fault Tolerant Scheduling for Parallel Loops

(a) milc                  (b) MT                 (c) ammp

Fig. 9. Comparison of FT-WSS with CR-GSS and CR-WSS when varying the number of permanent faults.

existing transient fault detection technique can be applied in our system. Therefore, we have not done experiment for transient fault detection.

FT-WSS adopts a processor discarding mechanism (Algorithm 2). We want to know whether it is worth applying in the scheduler or not. If it is worth applying, how much should the discarding threshold (Δ in Algorithm 2) be? We run milc, MT and ammp with 4 threads and 8 threads respectively and inject faults on one of the threads periodically. We compare two implementations. One is denoted by “No Discarding”, in which the thread always redoes the faulty iteration. Another is denoted by “Discarding”, in which the thread exits when faults occur 5 times on it (d = 5 in Algorithm 2). Fig. 10 reports the execution times of these two implementations in different fault recurrence intervals. From the figure, we can see “Discarding” outperforms “No Discarding” when the interval is small. In Figs. 10 (b) and (e), two lines meet between the intervals of 2 and 5. Thus, the threshold Δ should be less than 5. It indicates “Discarding” is a better choice only when the fault frequency is very high. Although Δ should be about 8000 for milc with 4 threads (Fig. 10 (a)), the total number of iterations in milc is 160000 and the execution time of each iteration is about 30μs. Thus the execution time of 8000 iterations would be about 240 ms, which still expresses a high frequency. In summary, the results suggest that the discarding mechanism is worth applying while the fault frequency might be very high. In addition, we observe that the threshold value (Δ) depends on both the application and the number of threads. For milc, Δ should be about 8000 when 4 threads are used, but 200 when 8 threads are used. For ammp, Δ should be about 3 when 4 threads are used, but 300 when 8 threads are used. From Figs. 10 (d) and (e), we observe that the execution time does not increase monotonically with increasing the fault frequency. The reason is that many aspects such as synchronization and data locality impact the performance in a real environment. The influence will be greater when more threads are used.

(a) milc, 4 threads       (b) MT, 4 threads        (c) ammp, 4 threads

Fig. 10. Impact of processor discarding mechanism on the execution time in different fault frequencies.
6. CONCLUSIONS

In this paper, we present the techniques to schedule parallel loops in the presence of hardware faults. To recover from transient faults, we introduce a method to make the loop iteration re-executable and re-execute a failing iteration directly. To tolerate permanent faults, a fault tolerant work-stealing scheme, FT-WSS, is proposed. Three novel mechanisms are adopted in FT-WSS: (a) runtime chunk splitting which improves load balancing; (b) a lightweight buffer-commit mechanism which replaces traditional check-point-roll-back fault recovery; (c) a discarding mechanism is applied to determine whether a processor on which faults frequently occur should continue to be used or not. We describe the implementation of FT-WSS for shared memory systems and evaluate it on a multi-socket multicore machine. Experimental results indicate that FT-WSS adds statistically insignificant runtime overhead and avoids a crash when faults occur.

As future work, we want to investigate the behavior of the discarding algorithm and try to give directions on how to get the discarding rate. Furthermore, we would like to extend FT-WSS to heterogeneous multicore systems.

ACKNOWLEDGEMENTS

The authors thank Professor Alexander V. Veidenbaum and Professor Feng Shi for their inputs during discussion sessions pertaining to the present study, and the anonymous reviewers for their valuable comments on the manuscript. This work was supported by the National Natural Science Foundation of China under grant NSFC-61300011.

REFERENCES


Yizhuo Wang (王一拙) received the Ph.D. degree in Computer Science from Beijing Institute of Technology, China, in 2005. He was a visiting scholar at University of California, Irvine from 2010 to 2011. Now he is an Assistant Professor of Computer Science at Beijing Institute of Technology. He is a member of CCF, IEEE and ACM. His research interests include parallel programming, fault tolerant computer systems, multi/many core architectures.

Rosario Cammarota received the Ph.D. degree in Computer Science from University of California, Irvine in spring 2013. Now he is part of The Office of The Chief Scientist R&D, in Qualcomm Technologies, Inc., San Diego, CA. His research interests fall in areas of high-performance architectures and their compilers, machine-learning assisted program optimization, and system security.

Alexandru Nicolau obtained his Ph.D. (’84) from Yale University where he was one of the founding members of the Bulldog/VLIW project. He was on the faculty of Cornell University from 1984 till 1988, when he joined University of California, Irvine as an Associate Professor and where he is currently a Full Professor. He is the author of over 250 peer-reviewed publications including several books, and serves as Editor in Chief of IJPP, the oldest journal in the field of parallel processing.