Using Explicit Output Comparisons for Fault Tolerant Scheduling (FTS) on Modern High-Performance Processors

Yue Gao
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, USA
yuegao@usc.edu, sandeep@usc.edu, mb@ceng.usc.edu

Sandeep K. Gupta
Melvin A. Breuer

Abstract—Soft errors and errors caused by intermittent faults are a major concern for modern processors. In this paper we provide a drastically different approach for fault tolerant scheduling (FTS) of tasks in such processors.

Traditionally in FTS, error detection is performed implicitly and concurrently with task execution, and associated overheads are incurred as increases in software run-time or hardware area. However, such embedded error detection (EED) techniques, e.g., watchdog processor assisted control flow checking, only provide approximately 70% error coverage [1, 2]. We propose the idea of utilizing straightforward explicit output comparison (EOC) which provides nearly 100% error coverage. We construct a framework for utilizing EOC in FTS, identify new challenges and tradeoffs, and develop a new off-line scheduling algorithm for EOC. We show that our EOC based approach provides higher error coverage and an average performance improvement of nearly 10% over EED-based FTS approaches, without increasing resource requirements. In our ongoing research we are identifying a richer set of ways of applying EOC, by itself and in conjunction with EED, to obtain further improvements.

I. INTRODUCTION

Providing reliable computation is the most basic requirement for any hardware system. However, with the scaling of technology, this is becoming more difficult to achieve [3]. Errors can occur in circuits due to many factors such as noise, high energy cosmic particles, radiation, and fatigue-induced intermittent faults in hardware. Any errors propagating to user outputs can potentially have detrimental effects. Much effort has been devoted at the level of circuit design and manufacturing to reduce the frequency of error occurrence, but these cannot entirely eliminate errors.

Fortunately, approaches at the software level can help. In this paper we consider fault tolerance in the context of task scheduling in distributed computing systems. There are two major aspects in fault tolerant scheduling (FTS):

1. Errors that can potentially corrupt outputs and cause operational failures must be detected.
2. The schedule must be able to tolerate errors via some form of redundant execution.

We begin with some background information.

A. Error Detection

To detect occasional errors, the most intuitive method is to execute multiple copies of the same program simultaneously on different processors and compare their outputs. We refer to this concept as Explicit Output Comparison (EOC). Triple Modular Redundancy (TMR) [4] is a commonly discussed example of EOC. EOC requires minimal or no program modifications and provides high error coverage, but is generally considered as imposing prohibitively high hardware overheads. To avoid these overheads, alternative methods have been proposed, where error detection is implicitly embedded during execution without the need for EOC. We refer to the broad collection of these error detection techniques, such as control flow checking (CFC), as embedded error detection (EED). EED inevitably increases the time required to complete tasks. We define the time to complete execution of a specific task on a standard processor/core without EED as the native latency (L), and the time needed for execution when EED is enabled as the EED latency (L'). For error detection using EOC, the total task execution time is the native latency L, plus the time required for output comparison which will be modeled in sections ahead.

EED can be efficiently implemented in hardware for certain components. Error correcting codes (ECC) [5] is a well-known error detection/correction method for regular structures, especially memories. As for techniques that are universally applicable in general purpose processors, the current prevailing methods are CFC and signature checking [6]. The basic idea is to verify that the program is branching to legal destinations and executing only expected types of instructions. For such techniques, the expected behavior of the program must be pre-computed and stored either in memory structures or the compiled code, incurring area or program size overheads. During execution, program signatures must be computed in real time, which incurs performance overheads. Recent research has proposed hardware-assisted error detection using separate watchdog processors [1]. But even with hardware assistance, EED still impacts performance.

The other shortcoming of any form of EED is incomplete error coverage. Although most EED methods report high error coverage (Table I provides an overview of the contemporary watchdog processor assisted EED techniques), their fault injection is often limited to faults that result in control flow violations. Hence, even if the reported coverage is 100%, it is only 100% of the control flow errors, which are around 70% of all the soft errors that can occur during execution [1, 2].

Table I. Overview of CFC Techniques with Watchdog Assistance

<table>
<thead>
<tr>
<th>CFC Technique</th>
<th>Memory Overhead</th>
<th>Performance Overhead</th>
<th>Reported Error Coverage</th>
<th>Modifies Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIC [7]</td>
<td>5% - 28%</td>
<td>52% - 189%</td>
<td>91% - 98%</td>
<td>Yes</td>
</tr>
<tr>
<td>CFCET [8]</td>
<td>3.5%</td>
<td>33% - 141%</td>
<td>80% - 85%</td>
<td>No</td>
</tr>
<tr>
<td>CFCSS [6]</td>
<td>26.6% - 63.6%</td>
<td>16% - 70%</td>
<td>96% - 98%</td>
<td>Yes</td>
</tr>
<tr>
<td>ACFC [9]</td>
<td>48% - 112.2%</td>
<td>41% - 136%</td>
<td>10% - 95%</td>
<td>Yes</td>
</tr>
<tr>
<td>YACCA [10]</td>
<td>91% - 96%</td>
<td>10% - 254%</td>
<td>Near 100%</td>
<td>Yes</td>
</tr>
<tr>
<td>CFCET [11]</td>
<td>33% - 44%</td>
<td>110% - 304%</td>
<td>89% - 94%</td>
<td>No</td>
</tr>
<tr>
<td>SWTEST [12]</td>
<td>90.9% - 174.8%</td>
<td>113% - 191%</td>
<td>81% - 98%</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*Only specific types of errors such as control flow errors are considered

In systems running safety-critical applications which require high error coverage, EED’s coverage is a limitation.

978-3-9815370-0-0/DATE13/©2013 EDAA
B. Fault Tolerant Scheduling

We consider the static off-line scheduling of aperiodic tasks in time-triggered architectures (TTA) [13] which can provide the desired predictability for safety-critical applications [14]. Tasks are mapped to processors and assigned start times prior to execution. The primary objective function is to minimize the total execution time for all tasks, also referred to as total latency or the length of the schedule. [15, 16] provide a summary of performance-effective static off-line scheduling algorithms. The drawback of static schedules is that they are susceptible to unpredictable events such as randomly occurring faults or upsets. To this end many ways of incorporating temporal and/or spatial redundancy to achieve fault tolerance have been proposed [17, 18, 19, 20].

Typically, fault tolerant scheduling (FTS) techniques assume that error detection is performed concurrently during execution, and any error detection overhead is integrated into the worst case execution time of each individual task. Furthermore, any area or performance overhead required for error detection is conceptually accepted but not explicitly accounted for. Simply put, scheduling begins after error detection is implemented in the form of EED and its impact on a task’s run time is taken as a given; this run time is never compared with the task’s run time without EED. Such conventional schedules will be referred to as EED schedules.

In this paper, we revisit the seemingly straightforward EOC concept for error detection in the context of FTS. We reveal the benefits of EOC in terms of achieving near-perfect error coverage while simultaneously reducing total latency, without increasing resource requirements. In this process we will present a comprehensive analysis of FTS that leverages tradeoffs in error detection, and thus expanding the scope of FTS. Such schedules will be referred to as EOC schedules.

II. OVERVIEW OF THE SYSTEM CONFIGURATION

A. Hardware Configuration

We model the hardware system as a set of R processing nodes \{C_1, C_2, ..., C_R\} interconnected by a communication channel. We will refer to them as cores for clarity. Each core C_i \in \{C_1, C_2, ..., C_R\} contains a main execution engine and a network interface that arbitrates communication. We assume homogeneous cores for simplicity, but our approach can be extended to heterogeneous cores. We also assume that the communication channel is implemented as a broadcast bus, where only one core can write to the bus at any given time, but all cores can simultaneously read from the bus. Other network topologies can potentially relax the constraints on channel access, but such variations are not central to our discussion.

Fig. 1 depicts a system with four cores.

EED and EOC can both be implemented completely in software with little or no change to the existing architecture. However, both approaches can also utilize additional hardware to reduce the performance overheads of error detection. Redesigning the hardware to reduce performance overheads of error detection is beyond the scope of this paper.

B. Application Model

We model the application as a directed acyclic graph G(V, E) called a task graph, such as the example shown in Fig. 2. Each vertex \(T_i \in V\) represents a task. Each task \(T_i\) is coupled with a 3-tuple: \(\{P_i, L_i, L'_i\}\), where \(L_i\) is the native latency of \(T_i\) and \(L'_i\) is the EED latency of \(T_i\). We define \(a_i = L'_i / L_i\) as the latency overhead of EED. In Fig. 2, \(a_i\) is set according to Table I. \(P_i\) is the size of the output data of \(T_i\).

An edge from \(T_i\) to \(T_j\) denotes that \(T_j\) is dependent on the output of \(T_i\). A task with no predecessors will be called an entry task, and a task with no successors will be called an exit task. \(D_{ij}\) represents the amount of time it takes to transport one unit of data from \(T_i\) to \(T_j\), hence the total time needed for data transmission between the two tasks is \(D_{ij} \cdot P_i\). We assume \(D_{ij} = 1\) if \(T_j\) is dependent on \(T_i\) and both tasks are not mapped onto the same core, 0 otherwise. We assume that each task can execute on any core, and that the message passing itself is fault tolerant, using protocols like the TTP [21].

C. Fault Model and Error Coverage

Faults in circuits could be permanent or intermittent; also circuits can have transient or soft errors due to external noise or radiation. In this paper we address intermittent faults and transient errors (henceforth referred to as faults), which are major concerns for modern processors [3]. We only consider faults that are not masked logically or architecturally, and eventually produce errors at task outputs and cause operational failures. For the techniques used in this paper, fault duration is irrelevant and is therefore considered to be atomic. If one or more fault occurs during the execution of \(T_i\) or the input to \(T_i\) is erroneous, the output of \(T_i\) will be deemed incorrect.

Naturally, any errors in the task output can be detected by EOC, namely comparing outputs of copies of the task executed on different cores. EOC achieves near-perfect error coverage, as the only exception is the unlikely event when multiple errors corrupt the outputs of various copies of the task running on distinct cores in identical ways. EED techniques such as CFC [6], in contrast, may only detect faults that cause control flow violations, resulting in 70% error coverage. In FTS, the error coverage does not depend on the task schedule, but on the underlying error detection mechanism.

For a given time frame, we consider \(k\) detectable faults which occur at arbitrary times. Under these conditions, if the schedule is guaranteed to deliver correct results within the timing constraints, it will be considered as having a fault tolerance level of \(k\). The value of \(k\) is determined by how tasks are scheduled, i.e., how redundant execution is applied. A schedule may have high fault tolerance level, i.e., a large \(k\), but
low error coverage due to incomplete coverage provided by EED. In this work we assume \( k = 1 \), and stricter fault tolerance requirements can be satisfied by decreasing the time frame.

### III. FAULT TOLERANT SCHEDULING TECHNIQUES

#### A. Scheduling a Single Task using EED

We begin with a brief review of FTS. Fig. 3 shows two ways of scheduling a single task \( T_i \). In Fig. 3a, \( T_i \) is scheduled with its EED latency \( L_i \); since EED is assumed. Without the slack, upon detection of an error, the system can halt operation or alert the user. If the slack is scheduled, then re-execution is possible and fault tolerance is achieved through this temporal redundancy [22]. The advantage of re-execution lies in slack sharing, which will be demonstrated later. Note that the length of the slack is equal to \( L_i \) in order to tolerate faults that occur towards task completion, so the ability to detect errors as soon as they occur cannot help reduce the slack. Re-execution can be applied at fine granularity if tasks can be divided into smaller subtasks, which is the basic principle of checkpointing [18]. Checkpointing will not be examined in this paper.

![Figure 3. Conventional scheduling of a single task \( T_i \) with EED.](image)

Fig. 3b illustrates another key FTS technique: replication, an instance of spatial redundancy. Two copies of the same task are mapped onto two different cores. If one core detects errors, the results of the other core will be carried to the next dependent task or system output through the broadcast bus during the following consolidation period. It is usually assumed that this consolidation overhead is equal to \( P_i \). This schedule achieves \( k = 1 \) without any slack. In [17], the authors combined re-execution with replication, thus fully exploiting the capabilities of EED. As an illustrative example, for the task graph in Fig. 2, the optimum EED schedule on four cores is shown in detail in Fig. 4.

![Figure 4. The Optimum EED Schedule (Total Latency: 69, \( k = 1 \)).](image)

#### B. Scheduling a Single Task using EOC

The missing coverage in EED can be addressed via the use of a second core and EOC (Fig. 5a). \( T_i \) is scheduled with its native latency \( L_i \). The two cores will mirror each other’s operation, but clock by clock synchronization is not necessary. Without the scheduling of the slack, errors can only be detected, but the error coverage is high. In cases where slack is included, when comparison fails, a contingency schedule is triggered, and \( T_i \) is re-executed on both cores during the slack. At time \( 2L_i + P_i \), both cores will contain correct outputs, and no more comparison is needed if we assume \( k = 1 \).

![Figure 5. Scheduling of a single task \( T_i \) with EOC.](image)

![Figure 6. Latency advantages of the EOC schedule.](image)

In the optimum EED schedule of three tasks in Fig. 6, consolidation is required after each task to support transparent execution in TTA [13]. Transparent execution allows any core to start execution of scheduled tasks regardless of faults occurring in other cores. In this paper all EED and EOC schedules will conform to transparent execution. The dashed edges of the exit tasks represent the consolidation time needed.
for the system to recognize which core contains the correct outputs, in case the exit tasks encountered errors.

In EOC schedule 1 shown in Fig. 6, comparison is done after every task completion. A slack of four time units is needed for the possible re-execution of the longest task \( T_2 \). \( T_1 \) and \( T_3 \) are shorter than \( T_1 \) and thus can share the slack. The total latency is 6.25% less than the optimum EED schedule. Also note that the EED schedule has very high resource usage as both cores do not have any idle time during the schedule.

The EOC schedule 2 shown in Fig. 6 further reduces the total execution time by (1) rearranging the tasks while still satisfying the precedence constraints, and (2) grouping \( T_i \) and \( T_j \) into one partition, i.e., performing comparison on the outputs of \( T_j \) only. If comparison fails, then re-execution will be initiated on both cores for \( T_j \) and \( T_j \) during the slack. Otherwise we conclude that both \( T_j \) and \( T_j \) have executed correctly. This is assuming that \( T_j \) consumes all the outputs of \( T_i \). In the following section we will detail those two important procedures, namely sorting and partitioning.

IV. EOC BASED FAULT TOLERANT SCHEDULING

A. Observations about EOC

We first present some observations that reveal new complications and tradeoffs in EOC scheduling. As illustrated in Fig. 6, the place where we choose to insert output comparisons is an important factor affecting the schedule length. Comparing after each task is completed will minimize the slack, but incur large comparison overheads. On the other extreme, comparing only after the exit tasks will minimize comparison overheads but maximize the slack. The optimal solution will typically be a point in between. The example below will provide further insight and important theorems.

Assume a set of \( n \) tasks, identical in latency. The native latency of each task is \( L \), and the EED latency is \( \alpha L \). \( T_i \) is dependent only on \( T_{i-1} \), and \( P_i = P \forall 1 \leq i \leq n \). The task graph is illustrated in Fig. 7, referred to as a linear task graph of \( n \) tasks. Also the hardware consists of \( R \) = 2 cores, and \( k = 1 \).

![Task Graph and Latency Table](image)

Theorem 1: When performing EOC scheduling under the above assumptions, to achieve minimum latency, comparisons should be made between every \( x \) tasks, where:

\[
x = \left\lfloor \sqrt{n \cdot P / L} \right\rfloor \quad \text{or} \quad \left\lfloor \sqrt{n \cdot P / L} \right\rfloor.
\]

The lower bound of total latency is: \( n \cdot L + 2\sqrt{n} \cdot P \cdot L \).

The upper bound of total latency when \( x \) takes the above values is: \( n \cdot L + 2\sqrt{n} \cdot P \cdot L + L + P \).

Proof: Deleted due to space limitations.

We notice that \( x \) is proportional to \( P \) and inversely proportional to \( L \). This is expected, since if the output data size is so large that communication and comparison overheads outweigh the execution latencies, the time saved from grouping tasks together to skip comparisons can offset the cost of increased slack. Similarly, the optimum EED schedule in this case is either one that only uses replication or one that only uses re-execution depending on the relationship between \( P \) and \( L \). A comparison of the optimum EED schedule with the optimum EOC schedule yields the following.

Theorem 2: The optimum EOC schedule will outperform the optimum EED schedule in terms of total latency if:

\[
a > \frac{1}{\alpha} \left( 1 + \frac{P}{nL} + \frac{L + P}{L} \right) \sqrt{\frac{nL}{P}}.
\]

Proof: Deleted due to space limitations.

Now we are ready to present our EOC scheduling flow, which is similar to [22] at a high-level. Here we mainly focus on the above key differences of EOC. In the next section we compare our EOC schedules with nearly optimum EED schedules that can use any combination of replication and re-execution as in [17] to eliminate bias in our results.

The inputs to the EOC scheduling algorithm are the task graph (Fig. 2) and the hardware configuration (Fig. 1). The first step of our EOC scheduling is very similar to [22], yet the second and third steps are specific to EOC scheduling only.

1) Mapping: Map each task to one of the available cores.

2) Detailed Scheduling: Schedule all tasks in each core while satisfying their mapping/precedence constraints.

3) Adjustments: Adjustments are made to ensure a legal schedule. Some ad-hoc optimizations are also applied.

B. Mapping

Task to core mapping is the first step of EOC or EED scheduling. A mapping heuristic for EED based on critical path clustering is presented in [22]. During this mapping process, the task with the highest critical path priority will be mapped to a core that minimizes the estimated schedule length. This process is repeated until every task has been mapped to a core. The schedule length can be easily and accurately estimated in [22] before the tasks are actually scheduled. We adopt this heuristic with one modification, namely we estimate the schedule lengths differently.

In EOC scheduling, the output of each task can only be delivered to other cores when the slack of the current and all previous partitions have been accounted for. However, the partitioning step has not yet been carried out. As a result, we replace the slack computation step in [22] by the following, which stems from Theorem 1:

\[
\text{slack}(T_k) = \text{MAX} \left( L_{x_0} \sum_l L_l / \sqrt{\sum_{i=1}^{n} P_i} \right).
\]

The rest of the mapping process is similar to [22].

C. Detailed Scheduling

The detailed scheduling step will determine the start time of every task. In [22], this is done using the list scheduling

---

1 For the proofs please contact the DATE program chair
algorithm [23], where the task with the highest priority is selected and scheduled on its allocated core. List scheduling is not applicable for EOC scheduling because it cannot handle the selection of the insertion of data comparisons. In the following subsections, we formulate the problem, and then provide optimization algorithms tailored to EOC scheduling.

**Problem Formulation**

Unlike list scheduling, detailed scheduling is recursively applied to each core pair (recall that in EOC two cores will mirror each other’s operation). For each core pair the detailed scheduling process is formulated as a sorting problem followed by a multi-way partitioning problem. Consider a task graph with \( n \) tasks. For a particular core \( C_i \), the mapping process will produce a subset of tasks \( T_i = \{ T_i, T_2, ..., T_N \} \), \( T_i \) is mapped onto \( C_i \). Assume that \( |T_i| = M \), and the tasks in \( T_i \) are labeled \( T_1-T_M \). The detailed scheduling step will:

1) Topologically sort the \( M \) tasks in \( T_i \).
2) Partition the \( M \) tasks in \( T_i \) into \( W \) groups labeled \( G_1-G_w \).

The objective of the detailed scheduling algorithm is to minimize the total execution time of the \( M \) tasks, defined as:

\[
\sum_{i=1}^{M} L_i + \sum_{j=1}^{W} \sum_{x \in T_j \text{ is task of } G_1} P_x + \text{MAX} \left( \forall i \sum_{x \in T_i \in G_1} L_1 \right)
\]

A task \( T_k \) is defined as an exit task of the current partition \( G_i \) if \( T_k \in G_i \) and \( \exists 1 \leq j \leq n \) such that \( T_j \in G_i \) and \( D_{ij} = 1 \).

Output comparisons will be made between partitions and after the completion of exit tasks of the current partition. In addition, the schedule must contain a timing slack, the length of which is equal to the latency of the longest partition.

**Guided Topological Sort**

For the \( M \) tasks in \( T_i \), any topological sort for the tasks would be legal. However, a sort that would minimize total latency is desired. Procedure 1 is a customized version of the basic topological sorting algorithm. It utilizes results from Theorem 1. Since the assumption of uniformity in Theorem 1 no longer holds, we use average values as approximations.

**Procedure 1: Guided Topological Sort**

**Initialize**

\[
P_{avg} = \frac{\sum P_i}{n}, L_{avg} = \frac{\sum L_i}{n}, G = \sum L_i \frac{n}{P_{avg}}, \text{ \{for } i \text{ from 1 to } N \}\text{ \}} \frac{\sum_{i=1}^{n} L_i}{\sum P_i}\]

PUSH all entry tasks into \(<\text{Ready Queue}>\) */A LIFO queue */

**End Initialization**

While (\(<\text{Ready Queue}>\neq 0\) ) begin

1) \( T_j = \text{POP} <\text{Ready Queue}>, \text{Counter} = 0 \)
2) Schedule \( T_j \), mark \( T_j \) as scheduled, UPDATE <Ready Queue> \text{, Counter} = \text{Counter} + L_j \)
3) for all \( T_j \) such that \( D_{ij} = 1 \)
   \( \text{if } (T_j < \text{Ready Queue} \&\& L_j + \text{Counter} < G) \text{ begin} \)
   \( \text{EXTRACT } T_j \text{ from <Ready Queue> \text{, Counter} = L_j + \text{Counter \text{, Schedule/Mark } T_j \text{ UPDATE <Ready Queue>}} \)

Output sorted tasks for Partitioning

**Partitioning**

We will now present a simulated annealing based algorithm (Algorithm 1) to perform partitioning of the sorted task sequence. To facilitate fast convergence to near-optimum results, we initialize the partitions according to Theorem 1. One can make a valid argument that Algorithm 1 is oblivious of the tasks in other cores. While true, optimizations that are geared towards inter-core dependencies such as list scheduling can be harmful to the partitioning step in our EOC framework.

Despite considerable efforts, we have not found an alternative that significantly outperforms Algorithm 1.

**Algorithm 1: Partitioning Algorithm Pseudo Code**

**Initialize**

Execute Procedure 1 /* Perform guided topological sort */

\[
x = \sqrt{\frac{n \cdot P_{avg}}{\sum P_i}} /* The initial number of tasks in one partition */
\]

Initialize partitions according to \( x \)

**End Initialization**

While (\( T > T_{final} \)) begin

Evaluate total execution time: \( L_1 \), save current configuration \( r = \text{Random}(0 \leq r < 1) \)

if (\( r < C_i \)) /* Randomly add or merge partitions */

Evaluate total execution time: \( L_1 \)

if (\( r < C_i \)) /* Append a new partition else Merge last two partitions */

Select a random partition, move the last task to the next partition

Evaluate total execution time: \( L_1 \)

if (\( r < e^{-\frac{\text{Length}(EED)-\text{Length}(EOC))}{T}} \)) /* Evaluate the effect of the move */

Restore and clear saved configuration /* Reject */

Evaluate total execution time: \( L_{final} \)

**D. Adjustments**

Unlike [22], the partitioning step in EOC scheduling will insert output data comparisons and alter the schedule timeline. As a result, the final schedule needs to be adjusted. This step also applies an ad-hoc optimization: for each entry task \( T_i \) it attempts to apply the method described in Fig. 5c for \( T_i \) if it is beneficial to the total latency.

**V. EXPERIMENTAL RESULTS**

**A. Two Core Architectures**

We assume that the system consists of two cores, the minimum amount of hardware for EOC to operate. In this case the mapping step can be skipped for EOC schedules. We examine two types of task graphs: linear and randomly generated. Randomly generated task graph will have native latencies uniformly distributed between 1 and \( L_{max} \) and output sizes uniformly distributed between 1 and \( P_{max} \). The EED latencies are defined as: \( L_i = \text{MIN}(2L_i - 1, [1.6L_i]) \).

We first examine 30 sets of distinct linear task graphs with \( n = 10 \) tasks, \( L_{max} = 11 \) and \( P_{max} = 3 \). The resulting EOC schedules are shown in Fig. 8; our EOC schedules provide an average 29% improvement over the optimum EED schedule, where improvement is defined as:

\[
\text{Improvement} = \frac{\text{Length}(EED) - \text{Length}(EOC)}{\text{Length}(EED)} \times 100\%.
\]

To demonstrate the effects of the simulated annealing procedure, we compare results of the initial and the final schedules, which revealed that our final schedules provide an average improvement of 4.6% over our initial schedules.

Figure 8. Total Execution Time Comparison (Two Core Architecture)
For random task graphs, the optimal EED schedule cannot be easily obtained. For our experiments, we implement the heuristics in [17] and apply extensive manual adjustments to obtain a good EED baseline schedule. We present eight sets of randomly generated task graphs each containing $n = 25$ tasks. As shown in Table II, our EOC schedules provide an average 6% improvement over the baseline EED schedule.

Table II. Total Execution Time for Randomly Generated Task Graphs

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best EED</td>
<td>136</td>
<td>116</td>
<td>131</td>
<td>110</td>
<td>131</td>
<td>100</td>
<td>90</td>
<td>86</td>
<td>-</td>
</tr>
<tr>
<td>EOC</td>
<td>114</td>
<td>110</td>
<td>114</td>
<td>106</td>
<td>105</td>
<td>105</td>
<td>90</td>
<td>91</td>
<td>-</td>
</tr>
<tr>
<td>Improvement</td>
<td>16%</td>
<td>5%</td>
<td>13%</td>
<td>4%</td>
<td>20%</td>
<td>-5%</td>
<td>0%</td>
<td>-6%</td>
<td>6%</td>
</tr>
</tbody>
</table>

B. $R(>2)$ Core Architectures

When the system consists of more than two cores, task to core mapping decisions must be made prior to partitioning. We will showcase the results of EOC scheduling for a four core system. We limit ourselves to $R = 4$ since TMR, EOC favors unbounded hardware resources, thus a large number of cores would provide EOC with an unfair advantage.

Figure 9. The EOC Schedule (Total latency: 63, $k = 1$)

For the task graph in Fig. 2 we illustrate the EOC scheduling results in detail. For $R > 2$ core architectures, TMR can be applied with three cores without slack. The EOC schedule shown in Fig. 9 has a 7.4% improvement over the EED schedule shown in Fig. 4, and 13.7% improvement over the TMR schedule, which is calculated as $\sum w_iL_i + 3P_{13} = 73$.

![EOC Set Performance Improvement Resource Usage Reduction](image)

<table>
<thead>
<tr>
<th>EOC</th>
<th>Performance Improvement</th>
<th>Resource Usage Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19.82%</td>
<td>-1.52%</td>
</tr>
<tr>
<td>2</td>
<td>9.90%</td>
<td>-3.53%</td>
</tr>
<tr>
<td>3</td>
<td>9.28%</td>
<td>14.19%</td>
</tr>
<tr>
<td>4</td>
<td>2.33%</td>
<td>-4.64%</td>
</tr>
<tr>
<td>5</td>
<td>15.12%</td>
<td>-4.81%</td>
</tr>
<tr>
<td>6</td>
<td>-2.30%</td>
<td>21.14%</td>
</tr>
<tr>
<td>7</td>
<td>15.00%</td>
<td>2.81%</td>
</tr>
<tr>
<td>8</td>
<td>7.14%</td>
<td>-4.87%</td>
</tr>
<tr>
<td>9</td>
<td>7.14%</td>
<td>-4.12%</td>
</tr>
<tr>
<td>10</td>
<td>14.29%</td>
<td>15.07%</td>
</tr>
<tr>
<td>Avg</td>
<td>9.77%</td>
<td>1.79%</td>
</tr>
</tbody>
</table>

Figure 10. Improvements provided by EOC over EED (four core architecture)

We now study 10 sets of randomly generated task graphs each consisting of $n = 25$ tasks. The generation process is slightly tuned to produce task graphs that resemble real applications. Fig. 10 summarizes the results, where EOC schedules provide an average 9.8% performance improvement.

It may seem that this latency improvement might have come from an increased use of resources, in other words, the EED schedule cannot fully utilize four cores. However, our results show that the resource usage of EOC schedules is slightly lower than that of the EED schedules. Therefore, compared to traditional EED schedules, EOC schedules can offer higher error coverage and nearly 10% performance improvement, without increasing resource requirements.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we overturn the accepted norms in fault tolerant scheduling (FTS) of assuming concurrent embedded error detection (EED) during execution, and outline a drastically different approach towards the problem. We first evaluate the coverage limitations and overheads of various error detection mechanisms, and revisit the seemingly simple idea of explicit output comparison (EOC). We construct a new framework of using EOC in FTS, identify the associated challenges and tradeoffs, and propose a new EOC based FTS algorithm. We show that compared to conventional EED scheduling methods, our new approach can provide higher (near-perfect) error coverage and nearly 10% improvement in performance, without increasing resource requirements.

In our ongoing research, we are identifying a richer set of ways of applying EOC, by itself and in conjunction with EED, and scheduling algorithms to obtain greater improvements.

REFERENCES