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

Yue Gao Sandeep K. Gupta Melvin A. Breuer
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, USA
yuegao@usc.edu, sandeep@usc.edu, mb@ceng.usc.edu

***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

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

**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.

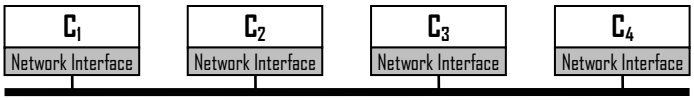
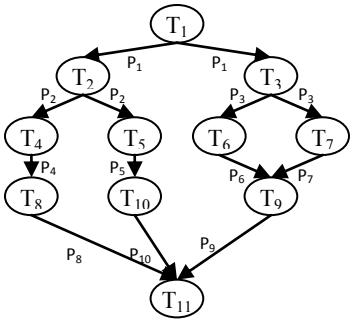


Figure 1. System Hardware Configuration

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 set: $\{P_i, L_i, L'_i\}$, where L_i is the native latency of T_i and L'_i is the EED latency of T_i. We define $\alpha_i = L'_i / L_i$ as the latency overhead of EED. In Fig. 2, α_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_{i_j} 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 $D_{i\_j} \cdot P_i$. We assume $D_{i\_j} = 1$ 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].



Task Table

	P	L	α	L'
T_1	1	3	1.67	5
T_2	1	4	1.50	6
T_3	1	4	1.75	7
T_4	1	5	1.6	8
T_5	1	6	1.5	9
T_6	1	7	1.43	10
T_7	1	7	1.57	11
T_8	2	8	1.50	12
T_9	1	8	1.63	13
T_{10}	3	9	1.56	14
T_{11}	1	9	1.67	15

Figure 2. Application Model

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 such 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.

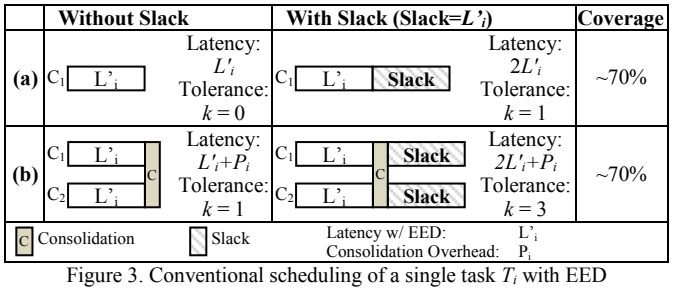


Figure 3. Conventional scheduling of a single task T_i with EED

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.

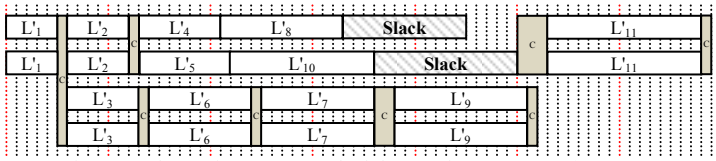


Figure 4. The Optimum EED Schedule (Total Latency: 69, $k = 1$)

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 $k = 1$.

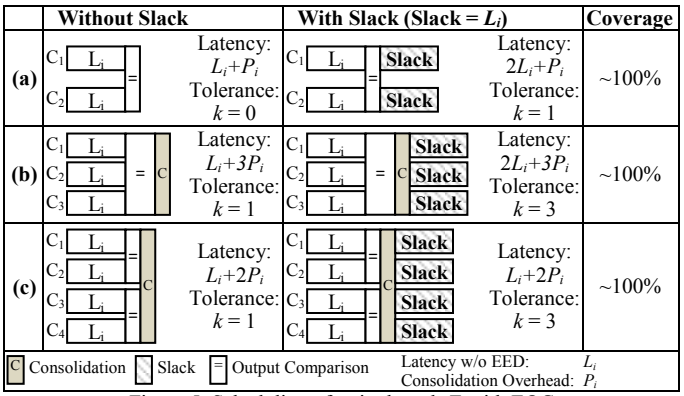


Figure 5. Scheduling of a single task T_i with EOC

Fig. 5b shows the concept of the classic TMR, which can eliminate the slack for $k = 1$. The voting procedure is assumed to require time $3P_i$. TMR is suitable when hardware resources are abundant. TMR will not be directly employed by our algorithm. However, we do opportunistically utilize a similar method described in Fig. 5c. In Fig. 5c, if comparison fails either for the C_1/C_2 pair or the C_3/C_4 pair, the correct results will be taken from other pair of cores during consolidation.

Two drawbacks of EOC are as follows: (1) the additional core requirement, and (2) the comparison overhead. We account for the first factor by using processors with identical number of cores when comparing EED and EOC schedules. Also, we explicitly account for the comparison overhead, which is equal to the size of the data to be compared: P_i.

While EED schedules only employ practices shown in Fig. 3, we believe that using the entire range of methods in Fig. 3 and Fig. 5 would result in much more compact schedules. In this paper however, we will limit ourselves to schedules with EOC alone (Fig. 5) to guarantee ~100% error coverage.

C. Case Study for Scheduling Multiple Tasks

A comparison between Fig. 3 and Fig. 5 may suggest that we are trading performance for higher error coverage. However, the motivational example shown in Fig. 6 will demonstrate that, by adopting EOC, it is possible to increase error coverage and, at the same time, reduce the total latency.

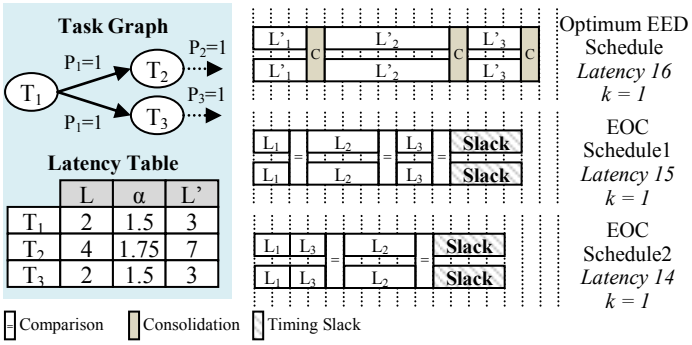


Figure 6. Latency advantages of the EOC schedule

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_2 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_1 and T_3 into one partition, i.e., performing comparison on the outputs of T_3 only. If comparison fails, then re-execution will be initiated on both cores for T_1 and T_3 during the slack. Otherwise we conclude that both T_1 and T_3 have executed correctly. This is assuming that T_3 consumes all the outputs of T_1. 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 αL. T_i is dependent only on T_{i-1}, and $P_i = P\ \forall 1 \le i \le n$. 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 $k = 1$.

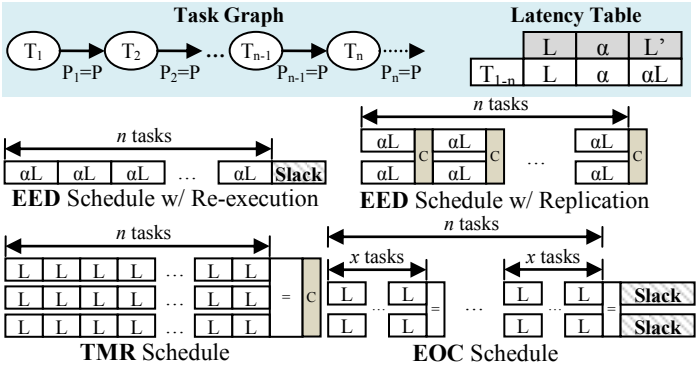


Figure 7. Scheduling a Linear Task Graph

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

$$x = \left\lfloor \sqrt{n \cdot P/L} \right\rfloor \text{ or } \left\lceil \sqrt{n \cdot P/L} \right\rceil.$$

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[1].

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:

$$\alpha > MAX\left(1 + 2\sqrt{\frac{P}{nL}} + \frac{L+P}{nL} - \frac{P}{L},\ 1 + 2\sqrt{\frac{P}{nL}} + \frac{P}{nL}\right)$$

Proof: Deleted due to space limitations[1].

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:

$$slack(T_k) = MAX\left(L_k,\ \sum_{\forall i} Li \Big/ \sqrt{\frac{n \cdot \sum_{\forall i} P_i}{\sum_{\forall i} L_i}}\right).$$

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_j, the mapping process will produce a subset of tasks $\boldsymbol{T_s} = \{T_i \mid T_i \in \{T_1, T_2, ..., T_N\}, T_i \text{ is mapped onto } C_j\}$. Assume that $|\boldsymbol{T_s}| = M$, and the tasks in $\boldsymbol{T_s}$ are labeled T_1-T_M. The detailed scheduling step will:

1) Topologically sort the M tasks in $\boldsymbol{T_s}$.
2) Partition the M tasks in $\boldsymbol{T_s}$ 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:

$$\sum_{i=1}^{M} L_i + \sum_{i=1}^{W}\left[\sum_{\forall x\, T_x \text{ is exit task of } G_i} P_x\right] + MAX\left(\forall i \sum_{\forall x\, T_x \in G_i} Lx\right)$$

A task T_x is defined as an exit task of the current partition G_i if $T_x \in G_i$ and $\nexists 1 \le j \le n$ such that $T_j \in G_i$ and $D_{x\_j} = 1$.

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 $\boldsymbol{T_s}$, 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
  $P_{avg} = \frac{\sum_{\forall i} P_i}{n}$, $L_{avg} = \frac{\sum_{\forall i} L_i}{n}$, $G = \sum_{\forall i} L_i / \sqrt{\frac{n \cdot P_{avg}}{L_{avg}}} = \sum_{\forall i} L_i / \sqrt{\frac{n \cdot \sum_{\forall i} P_i}{\sum_{\forall i} L_i}}$
  PUSH all entry tasks into < Ready Queue > /* A LIFO queue */
End Initialization
While (< Ready Queue > ≠ ∅) **begin**
  T_i = **POP** < Ready Queue >, *Counter* = 0
  Schedule T_i, mark T_i as scheduled, UPDATE < Ready Queue >
  Counter = Counter + L_i
  for all T_j such that $D_{i\_j} = 1$
    if ($T_j \in$ < Ready Queue > & L_j + *Counter* < G) **begin**
      EXTRACT T_j from < Ready Queue >
      Counter = L_j + *Counter*, Schedule/Mark T_j
      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 */
  $x = \left\lceil \sqrt{\frac{n \cdot P_{avg}}{L_{avg}}} \right\rceil$ /* 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 = Random(0<r<1)
  if ($r < C_1$) **begin** /* Randomly add or merge partitions */
    if ($r < C_2$) **begin**
      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_2
  if $\left(r > e^{-\frac{Scale \cdot (L_2 - L_1)}{T}}\right)$ **begin** /* Evaluate the effect of the move */
    Restore and clear saved configuration /* Reject */
  $T = T$ - *Cooling*
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: $L'_i = MIN(2L_i - 1, \lceil 1.6L_i \rceil)$.

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:

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

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.

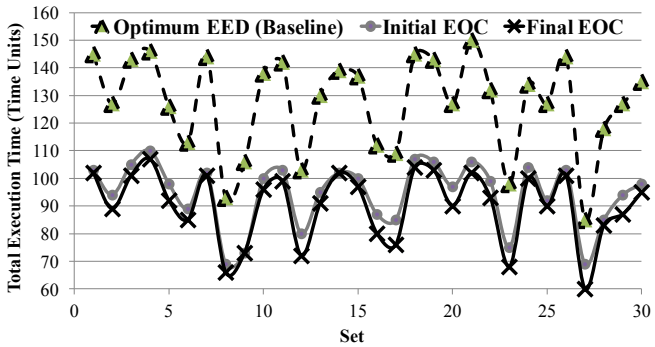


Figure 8. Total Execution Time Comparison (Two Core Architecture)

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

	1	2	3	4	5	6	7	8	Avg
Best EED	136	116	131	110	131	100	90	86	-
EOC	114	110	114	106	105	105	90	91	-
Improvement	16%	5%	13%	4%	20%	-5%	0%	-6%	**6%**

B. R (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 like TMR, EOC favors unbounded hardware resources, thus a large number of cores would provide EOC with an unfair advantage.

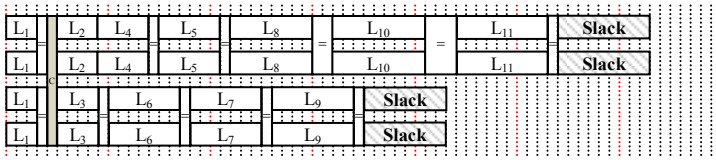


Figure 9. The EOC Schedule (Total latency: 63, k = 1)

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 $\sum_{\forall i} L_i + 3P_{11} = 73$.

EOC Set	Performance Improvement	Resource Usage Reduction
1	19.82%	-1.52%
2	9.90%	-5.35%
3	9.28%	14.19%
4	2.33%	-4.64%
5	15.12%	-4.81%
6	-2.30%	21.14%
7	15.00%	2.81%
8	7.14%	-14.87%
9	7.14%	-4.12%
10	14.29%	15.07%
Avg	**9.77%**	**1.79%**

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

[1] A. Pataricza et al., "Watchdog Processors in Parallel Systems," *Symp. on Microprocessing and Microprogramming*, 1993.

[2] R. Venkatasubramanian et al., "Low-Cost On-Line Fault Detection Using Control Flow Assertions," *Int'l On-Line Testing Symp.*, 2003.

[3] C. Weaver et al., "Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor," *Int'l Symp. on Computer Architecture*, 2004.

[4] R. E. Lyons and W. Vanderkulk, "The Use of Triple Modular Redundancy to Improve Computer Reliability," *IBM Journal of Research and Development*, 7(2):200–209, 1962.

[5] J. Chang et al., "The 65nm 16mb On-Die L3 Cache for a Dual Core Multi-Threaded Xeon Processor," *Symp. on VLSI Circuits*, 2006.

[6] N. Oh et al., "Control-Flow Checking by Software Signatures," *IEEE Trans. on Reliability*, 51(1):111–122, 2002.

[7] A. Rajabzadeh et al., "Error Detection Enhancement in COTS Superscalar Processors with Event Monitoring Features," *Pacific Rim Symp. on Dependable Computing*, 2004.

[8] A. Rajabzadeh and S. G. Miremadi, "A Hardware Approach to Concurrent Error Detection Capability Enhancement in COTS Processors," *Pacific Rim Symp. on Dependable Computing*, 2005.

[9] R. Venkatasubramanian et al., "Low-cost on-line fault detection using control flow assertions," *On-Line Testing Symp.*, 2003.

[10] O. Goloubeva et al., "Soft-Error Detection Using Control Flow Assertions," *Int'l Symp. on Defect and Fault Tolerance in VLSI Systems*, 2003.

[11] M. Fazeli et al., "A Software-Based Concurrent Error Detection Technique for PowerPC Processor-based Embedded Systems," *Int'l Symp. on Defect and Fault Tolerance in VLSI Systems*, 2005.

[12] Y. Sedaghat et al., "A software-based error detection technique using encoded signatures," *Int'l Symp. on Defect and Fault Tolerance*, 2006.

[13] H. Kopetz and G. Bauer, "The Time-Triggered Architecture," *Proc. IEEE*, 91(1):112–126, 2003.

[14] H. Kopetz, "Real-Time Systems – Design Principles for Distributed Embedded Applications," Kluwer Academic Publishers, 1997.

[15] Y.-K. Kwok and I. Ahmad, "Benchmarking the Task Graph Scheduling Algorithms," *Int'l Parallel Processing Symp./Symp. on Parallel and Distributed Processing*, 1998.

[16] H. Topcuoglu et al., "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," *IEEE Trans. on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[17] V. Izosimov et al., "Design Optimization of Time- and Cost-Constrained Fault-Tolerant Distributed Embedded Systems," *DATE*, 2005.

[18] P. Pop et al., "Design Optimization of Time- and Cost Constrained Fault-Tolerant Embedded Systems with Checkpointing and Replication," *DATE*, 2005.

[19] V. Izosimov et al., "Analysis and Optimization of Fault-Tolerant Embedded Systems with Hardened Processors," *DATE*, 2009.

[20] V. Izosimov et al., "Synthesis of Fault-Tolerant Schedules with Transparency/ Performance Trade-offs for Distributed Embedded Systems," *DATE*, 2006.

[21] H. Kopetz et al., "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach," *IEEE Micro*, 9(1):25-40, 1989.

[22] N. Kandasamy et al., "Transparent Recovery from Intermittent Faults in Time-Triggered Distributed Systems," *IEEE Trans. on Computers*, 52(2):113–125, 2003.

[23] T. L. Adam et al., "A Comparison of List Schedules for Parallel Processing Systems," *Comm. ACM*, 17(12):685–690, 1974.