Time-Triggered Scheduling of Query Executions for Active Diagnosis in Distributed Real-Time Systems

Sarah Amin
Embedded Systems Lab
University of Siegen
Siegen, Germany 57076
Email: sarah.amin@uni-siegen.de

Roman Obermaisser
Embedded Systems Lab
University of Siegen
Siegen, Germany 57076
Email: roman.obermaisser@uni-siegen.de

***Abstract*—In recent years, many control applications have replaced safety critical mechanical systems with distributed real-time embedded systems comprising of many processors, sensors and actuators interlaced together with a dedicated communication network and without any mechanical backup e.g., steer-by-wire used in steering systems of automotive vehicles. Such systems demand a high level of reliability and performance. These systems also have severe cost constraints so including redundant components in the end product is not a viable solution for improving reliability and performance. Another solution is to continuously monitor the system and introduce fault diagnosis to ensure that the dependability of the system is greater than the dependability of its constituent hardware and software components. Active diagnosis is one such technique that improves the reliability of the system by diagnosing facts at run-time for fault isolation and error recovery. The presented work addresses an active diagnosis scenario that uses diagnostic queries and a real-time database to find faults within a distributed system that has limited resources and strict deadlines. Since scheduling the diagnostic tasks is an important aspect of a timely analysis of the system, a list schedule has been proposed that calculates the points in time when the diagnostic queries are executed and data is replicated to the database. This a priori knowledge about the behavior of the query executions will bound the time required for inferring faults that will lead to a realizable diagnostic framework. The proposed algorithm utilizes a priority scheme to schedule the diagnostic tasks onto free processors within minimum time while respecting their precedence and periodicity constraints. The paper presents the approach in detail with the help of examples and results with different design constraints.**

***Index Terms*—distributed fault analysis, diagnosis, active diagnosis, real-time, distributed systems, list scheduling, time-triggered**

I. INTRODUCTION

In recent years, many control applications, such as railway systems, electrical power distribution systems, automotive systems and command/control systems have replaced safety-critical mechanical systems with distributed real-time embedded systems. For example, in automotive vehicles - steer-by-wire (SBW) is such a component where the traditional steering-system was replaced by a distributed system consisting of microprocessors, sensors, actuators etc., without any mechanical backup [1]. These kind of applications demand a high level of safety, performance and reliability. These systems also have severe cost constraints so including redundant components in the end product is not a viable solution for improving reliability and performance [2]. Another solution is to continuously monitor the system for faults and ensure that the embedded computer operates with a dependability that is higher than the dependability of its constituent hardware and software components. Since the electronic components can fail at any given time, this level of dependability can only be achieved if the system endorses fault tolerance [3]. An important step in fault analysis is diagnosis that determines the reasons of failures in terms of localization and state of the system [4]. Diagnosis can either be performed by storing the information and analyzing it later for maintenance and engineering feedback (passive diagnosis) or by analyzing the information at run time so that an immediate recovery action can be taken (active diagnosis). In safety-critical systems, it is important that a fault is determined and the problem is solved within predictable time before the system reaches an unsafe state but most existing active diagnosis solutions do not support strict timing constraints that are essential for fault analysis in real-time applications [4]. Most control applications manage the loss of control inputs only for a few cycles but longer outages shut off the system completely due to the incurring fault. For example, in a steer-by-wire the maximum time for which the control systems can freeze the actuators is 50ms [5]. This aspect raises the need of a diagnostic solution that respects the timing constraints in control applications and identifies faults within predictable time. Prior work has been performed by authors in [2] and [6] but their work is application specific and focuses mainly on actuator diagnosis in automotive vehicles.

This paper addresses fault analysis in a distributed system that has limited resources and stringent timing constraints. The system identifies faults using a graph of queries that is modeled on resource description framework data. This model has been previously used by authors in [4]. Each query is associated with a sensor or group of sensors that provide data with respect to their specific periods. These sensors utilize local-error identification methods, e.g. message classification, to provide diagnostic facts of the system that serve as the starting point for the analysis. For controlled analysis, these facts are

978-1-5090-6505-9/17/$31.00

divided into intermediate steps called symptoms. Each fact is realized as a query in the real-time database and is used for the generation of symptoms or identification of faults. The output data from the sensors is stored in this real-time database, and is timely and consistently replicated to ensure distributed execution of the queries. Since scheduling the diagnostic tasks is an important aspect for a timely analysis of the system, this paper proposes a heuristic to schedule the diagnostic queries used in such a system. The purpose of the scheduler is to guarantee fault identification within predictable time while respecting the precedence and periodicity constraints of the diagnostic tasks. Prior to the execution of the queries, the scheduler will decide the points in time at which the data will be replicated or the query will be executed. This a priori knowledge about the behavior of the query executions will bound the time required for inferring faults that will lead to a realizable diagnostic framework [4].

The rest of the paper is structured as follows. Section II provides an overview of the related work. Section III discusses the structural and application model used for the analysis. Section IV discusses the heuristic in detail along with an example and experimental results. Lastly, section V provides a discussion of the proposed heuristic and also illustrates some future work.

II. Related Work

In the state-of-the-art, a lot of work has been performed in the field of distributed fault analysis and diagnosis. One of the famous examples for active diagnosis is the classical PMC model [7] that was later modified to identify transient faults in distributed systems [8]. In these models researchers worked on the assumption that processors test each other and swap test results to identify underlying faults. Since it is hard to acquire test results, many comparison-based techniques have been proposed where tasks are duplicated onto processing elements and their end results are compared to identify the faulty-ones [9]–[12]. These techniques were modified by authors in [13] and [14], where the on-line scheduler executes multiple copies of tasks on different processors using the spare processing capacity of the system. Bayesian inference and dynamic decision networks were used in [15] for fault identification and recovery in an on-board architecture. An on-line diagnosis technique is presented by authors in [16], where they used residuals from parametric estimations to identify faults in a non-linear model. Another on-line diagnosis technique used Kalman filters with a set of precision samples to identify faults in non-linear systems [17]. There are many other model-based techniques where system behavior is compared with a reference model to identify faults [18]–[22]. In [23] and [24], authors have used model-based methods to self validate actuators and broadcast their status to the rest of the system. Most of these aforementioned methods do not consider stringent timing constraints and reliability requirements that are essential for analysis in real-time distributed systems. Moreover, they assume that the defaulted processors do not effect the end results and are masked by the faultless ones. On the other hand, this approach diagnosis failed components within the strict timing constraints of the distributed system and takes into account the defaulted processors while diagnosing the system.

In [2] and [6], authors have proposed time-constrained fault identification techniques for steer-by-wire and fly-by-wire systems of automotive vehicles. They utilize model-based techniques to diagnose actuators in a distributed fashion. Although these techniques also follow the strict timing constraints required in real-time distributed systems, our approach is different in the following aspects: 1). our diagnosis model follows rule-based inference methods and semantic web techniques to identify both hardware and software failures [4], 2). in addition to identifying faults, our technique implements the relevant recovery actions including the application-specified ones that are not considered in the mentioned approaches, 3). in the aforementioned approaches, authors have considered a contention free communication network while our approach considers the possibility of limited communication resources, 4). we have used non-preemptive scheduling in contrast to preemptive one to avoid unnecessary overheads and 5). our application model is based on a directed multi-query graph (DMG) that has a specific deadline and consists of different periodic queries. Therefore, in addition to the deadline constraint, the scheduler has to consider the cyclic period of each query.

List scheduling [25] is one of the most studied class of heuristics. The basic structure of list scheduling consists of two parts: assigning priorities to the tasks on the basis of task level or path length or path time of the input task graph, and then scheduling the tasks in decreasing order of their priorities. Once calculated, the task priorities remain constant and do not change during their assignment to the processors. Heterogeneous Earliest Finish Time (HEFT) [26] is one such scheduling algorithm that uses a recursive approach in the bottom-up direction to determine the order of the nodes. The Critical Path/Most Immediate Successors First (CP/MISF) [27] algorithm executes the nodes in decreasing order of their bottom level. The Dynamic Critical Path (DCP) [28] scheduling algorithm follows the critical pattern traversal approach and attempts to minimize the schedule length at each step considering the remaining critical path. A final schedule is unattainable until all the nodes have been processed. In another algorithm [29], critical path nodes are scheduled first and then non critical path nodes are scheduled on the basis of their bottom level. Modified Critical Path (MCP) [30] heuristic orders the nodes according to their bottom level and for nodes with equal bottom levels, the bottom levels of the successor nodes are considered. The aforementioned techniques are generally used for scheduling aperiodic tasks in parallel systems. Since our application model considers periodic tasks so the heuristic has to be modified to fulfill the requirements of active diagnosis in distributed real-time systems.

III. System Model

This section elaborates the architecture and application model of a distributed system with active diagnosis. In addition, we outline the restrictions and constraints that are necessary for the proposed algorithm.

A. Architecture Model

The model for the target parallel system consists of N nodes comprising R routers and C cores with L links between them. The nodes are connected with each other using bi-directional communication links. The system has the following restrictions:

1) The system has homogeneous cores with heterogeneous links and routers. Input data essential for the diagnosis is coming from the sensors present in the system.
2) The system runs both the target and diagnostic applications in parallel with each other.
3) The cores are designated for the processing of queries only and will not take part in any kind of communication. Similarly, routers are only available for the transmission of messages.
4) The schedule is computed on the same parallel system but it has its own dedicated resources which do not take part in the execution of the scheduled DMG. Since there are no resource conflicts between the execution and computation of the schedule, they both are executed in parallel without any conflicts.
5) The scheduling is non-preemptive i.e. the cores are designated to process one task at a time only. The scheduler decides the point in time and the core at which each task is processed prior to the start of execution.
6) Communication is carried out via a deterministic protocol such as Time Division Multiple Access (TDMA) and only one message is transmitted through a link at a given time.
7) If two tasks are scheduled on the same core then their communication cost is negligible. This is in regard with the fact that in many parallel systems, distant communication is generally more costly than local communication.
8) If dependent tasks are scheduled on different cores then they incur a communication cost that is computed by the scheduler depending on the cost of the path upon which the communication is carried out and the size of the communicated message.
9) Once a task has been assigned to a processor, it executes all its periodic iterations on that processor only. The computation time of the task does not change and all the iterations require the same amount of time to be executed.
10) Similarly, a communicating task executes all its iterations on the path that has been assigned to it. Also, the execution time of the communicating task remains the same for all of its iterations.
11) Each core has dedicated memory and there are no memory constraints, i.e. the cores have enough memory to store the data without any problems. Moreover, there is no conflict for memory resources.

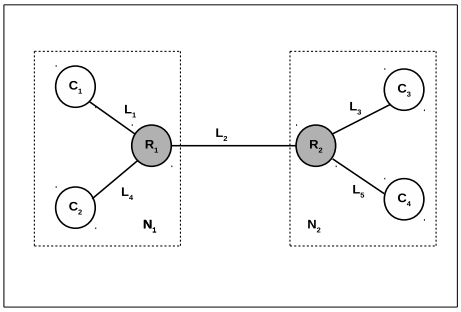


Fig. 1. Example of a Parallel System

There is no restriction on the structure of the parallel system and it can follow any topology with an arbitrary number of nodes and with an arbitrary number of communication links between them. An example of the parallel system is depicted in Fig. 1.

B. Application Model

We consider a diagnostic scenario where the input is a directed graph of queries. This graph is formally termed as Diagnostic Multi-Query Graph (DMG) and has been previously used by the authors in [4]. A DMG is similar to a directed task graph with the exception that it is implemented using a real-time database.

In a DMG, each root node is associated with a set of sensors that provide continuous input data according to their periods. This input data is stored in a real-time database. The sensors utilize local-error detection methods to formulate diagnostic facts that are further used to identify symptoms, faults and to propose respective recovery actions. These diagnostic facts are realized as queries in this real-time database and are executed repeatedly to keep the diagnosis up-to-date with the input data. In the DMG, the queries are represented by nodes and the relationship between the queries within the real-time database is represented by edges. The nodes are labeled with the worst-case execution time of their associated queries. Since we are dealing with real-time systems, each node is cyclic with a strict period [31]. A strict period means that if a task A has a period T_A then the difference between the starting points of two of its consecutive iterations should be equal to T_A. Here, nodes are termed as features (without incoming edges), symptoms (with both incoming and outgoing edges) and faults (without outgoing edges) respectively. The edges are labeled with the amount of data the nodes are transmitting to the database and a history interval that identifies the past executions of the parent node from which the data is required for the execution of the target node. As a query repeats its execution each time with a different set of input data so it is possible that its corresponding child query requires data from one of its previous or following repetitions rather than its current one. The start of the history interval shows the number of the repeated instance of the parent node that starts the data transference and the end of

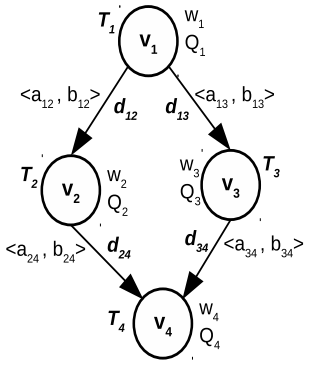


Fig. 2. Directed Multi-Query Graph - G

the interval represents the number of the repeated instance till which the data is required by the child node. In simpler words, the output data obtained between these repeated instances of the parent node is required for the execution of the child node. The history interval also determines the instances when features and symptoms can be discarded from the real-time database and the acceptable communication delay between parent and child node.

Such a graph can be represented by:

$$G = (V, E, T, w, d) \tag{1}$$

where each vertex $v_i \in V$ is a non-divisible periodic task. Each directed edge $e_{v_i v_j} \in E$ illustrates the precedence constraint between $v_i \in V$ and $v_j \in V$ such that v_i is the parent vertex to v_j. A positive weight w_{v_i} represents the worst-case execution time of vertex $v_i \in V$, Q_{v_i} describes the query associated with this vertex while T_{v_i} represents its time period. Data $d_{e_{v_i v_j}}$ on edge $e_{v_i v_j}$ account to the amount of data being transferred from the parent node v_i to the child node v_j. This amount of data effects the communication cost between the tasks as larger data will take more time to transfer [2]. The edge $e_{v_i v_j} \in E$ is also labeled with a history interval, $<a_{ij}, b_{ij}>$. Here a_{ij} and b_{ij} represent the *ath* and *bth* instances of the parent vertex $v_i \in V$. The subsequent child node $v_j \in V$ cannot start its execution before all of the data between these particular executions of v_i has been successfully transmitted to its assigned processor. The size of the DMG is not restricted and it can have any arbitrary size and structure. An example of the DMG is shown in Fig. 2.

In our system, we assume that the structure of the application and the target parallel system is completely known prior to execution.

C. Graph Properties

This section will elaborate some characteristics related to the graph G that are utilized in the algorithm.

1) *Time Period*
Since we are dealing with periodic tasks, for a task graph G represented by (1), each vertex $v \in V$ will repeat itself after an interval T_v. This interval is termed as time period and is the exact time elapsed between two consecutive iterations of v [32]. It can be represented with the following equation:

$$T_v = s_{v_{i+1}} - s_{v_i} \quad \forall v_{\mathrm{i}} \in V \tag{2}$$

where $s_{v_{i+1}}$ and s_{v_i} are the start times of *(i+1)th* and *ith* iteration of task v respectively.
In this paper we are restricting the time periods of two communicating tasks to be either equal to or be multiples of each other for successful transmission of data between them. If $\mathrm{v_p}$ is transmitting data to $\mathrm{v_c}$ then we can use (3) or (4) to calculate the number of times each task has to be repeated for complete transmission of data. This restriction has also been used by authors in [33].

$$n_{\mathrm{v_p}} = \frac{T_{\mathrm{v_c}}}{T_{\mathrm{v_p}}} \quad if \quad T_{\mathrm{v_c}} \geq T_{\mathrm{v_p}} \quad \forall v_{\mathrm{p}} \in V \quad \forall v_{\mathrm{c}} \in V \tag{3}$$

$$n_{\mathrm{v_c}} = \frac{T_{\mathrm{v_p}}}{T_{\mathrm{v_c}}} \quad if \quad T_{\mathrm{v_p}} > T_{\mathrm{v_c}} \quad \forall v_{\mathrm{p}} \in V \quad \forall v_{\mathrm{c}} \in V \tag{4}$$

where n_{v_p} and n_{v_c} are the number of times v_p and v_c have to be repeated for successful transmission of data between them.

2) *Hyper-Period*
For a set V of n periodic tasks in graph G represented by (1), the hyper-period H_p is the minimum time interval after which V will repeat its execution [32]. It is calculated by taking the least common multiple of time periods of all tasks present in V.

$$H_p = LCM(T_{v_1}, T_{v_2}, ..., T_{v_n}) \quad \forall v \in V \quad \forall T_{\mathrm{v}} \in G \tag{5}$$

The hyper-Period is an important characteristic of a task set and is used to calculate the number of times each task is repeated within one complete execution of the graph [33].

$$n_v = \frac{H_p}{T_v} \quad \forall v \in V \quad \forall T_{\mathrm{v}} \in G \tag{6}$$

where n_v is the number of times each task $v \in V$ has to be repeated within one interval and H_p is the hyper-period of the task graph G. It has to be noted that the final schedule length for one complete iteration of task graph G will never be less than its hyper-period.

3) *Utilization Factor*
For a set V of n periodic tasks in task graph G represented by (1), the utilization factor U_T is the fraction of processor time taken by the task set to complete its execution [32]. It can be represented as:

$$U_T = \sum_{i=1}^{n} \frac{w_{v_i}}{T_{v_i}} \quad \forall v \in V \quad \forall T_{\mathrm{v}} \in G \tag{7}$$

where w_{v_i} is the execution time of $\mathrm{v_i}$ and T_{v_i} is its corresponding time period.

4) *Path Length*
For a task graph G represented by (1), a path p from

vertex $v_0 \in V$ to vertex $v_i \in V$ is a sequence of vertexes $<v_0,v_1,...,v_i> \in V$ such that they are connected by edges $<e_{v_0v_1},e_{v_1v_2},...,e_{v_{i-1}v_i}> \in E$ [25].

The sum of the weights of the vertexes and edges, belonging to the path $p \in G$, is termed as path length [25] pl_p and can be described by (8).

$$pl_p = \sum_{v \in p,V} w_{\mathrm{v}} + \sum_{e \in p,E} d_{\mathrm{e}} \quad \forall p \in G \tag{8}$$

where w_v is the computation time of the vertex v and d_e is the amount of data being transmitted on the edge e.

5) *Bottom Level*

For a graph G represented by (1), the bottom level bl_{v_i} of a task $v_i \in V$ is the longest path in the graph from v_i to v_n (sink node) [25]. It can be calculated by determining the maximum path length using (8) from v_i to v_n.

$$bl_v = max_{v_i \in (desc(v) \cap sink(G))} \quad pl(v \rightarrow v_i) \tag{9}$$

where *desc(v)* is a set of descendants of task v and *sink(G)* is the task $\mathrm{v_n}$ in a set V of n periodic tasks with no further successors. If *desc(v)* = $\emptyset$ then $bl_{\mathrm{v}} = w_{\mathrm{v}}$.

IV. Scheduler

Our algorithm is based on list scheduling heuristic and prioritizes the tasks on the basis of the bottom level of each node. The ordered list is then scheduled onto a free processor while respecting the precedence and periodicity constraints of each task. Initially, the algorithm performs a test on the input DMG to determine whether it can be scheduled or not. This test is performed in accordance with the following two conditions.

- First, the algorithm calculates the utilization factor of the DMG using (7). For every schedulable DMG, the following condition holds true [32].

$$U_T \leq 1$$

where U_T is the total utilization factor of the DMG. According to (7), this condition holds true only when for each vertex $v \in V$, $T_{\mathrm{v}} \geq w_{\mathrm{v}}$. This statement is logical because if a task has greater execution time than its time period then it will definitely miss its deadline irrespective of the algorithm used and thus cannot be scheduled.

- If parent-child tasks have unrelated time periods then the DMG cannot be scheduled [33]. If a task $v_p \in V$ with a time period T_p is transmitting data to $v_c \in V$ with a time period T_c then one of the following two conditions should hold true for the DMG to be schedulable.

$$\frac{T_p}{T_c} = 0 \quad if \quad T_p > T_c$$

$$\frac{T_c}{T_p} = 0 \quad if \quad T_c \geq T_p$$

If the above two conditions hold true then the scheduler calculates the hyper-period of the DMG using (5) and the number of instances required by each task to complete its execution within one complete execution of the DMG using (6). These number of instances are important to calculate the schedule length for one complete iteration of the DMG. The scheduler then identifies the paths in the DMG through parent-child relationship of each vertex. Each vertex is assigned a priority according to its bottom level that is calculated via (9), as illustrated in Fig. 6. The tasks are then sorted and are systematically assigned to free processors when they become ready. A task is said to be ready for execution when all of its parent tasks have completed their execution. The scheduler calculates the number of instances required by each child vertex to start its execution and then adds a vertex to the ready list only when all of its parent vertexes have completed their required number of executions. The list is then sorted in descending order of the task priorities. This has been illustrated in Fig. 7.

Each task from the ready list is then assigned to a free processor. An important characteristic during this assignment is the successful transmission of data required for the execution of the task. If parent-child tasks are assigned to different core then corresponding communication tasks are created having priority and time period equivalent to that of the receiving task. A message is ready for transmission when its sending task has completed its required number of executions. The time required for transmitting a message $m_{v_iv_j}$ from v_i to v_j is expressed by (10) when both tasks are assigned to two different cores with a total of n links between them. Here the amount of data being transmitted is expressed by $d_{v_iv_j}$ and $R(L_i)$ represents the transmission rate of link L_i. For example, consider two cores C_1 and C_2 with links L_1 and L_2 between them. The transmission rate for L_1 is 2 units/sec while 3 units/sec for L_2. Now if a task v_1 is transmitting 6 units of data from C_1 to task v_2 that is assigned to C_2 then the total time required for transmitting the data is 5 seconds.

$$c(m_{v_iv_j}) = d_{v_iv_j} \sum_{i=1}^{n} \frac{1}{R(L_i)} \tag{10}$$

Moreover, in order to ensure that the data is completely transferred from the parent to the child task, the algorithm calculates the number of instances a communicating task has to be repeated for complete transmission of its output data using (3) or (4) depending upon the time periods of the tasks. This concept has been extracted from [33]. For example, consider a task v_1 with a time period T_1 of 2 units that has to transfer data to a task v_2 with a time period T_2 of 6 units. Then three iterations of v_1 should be completed and data should be completely transferred from these iterations to the core assigned to v_2 before it starts its execution. This has been illustrated in Fig. 3. An important point to note here is that the output data is being buffered at the sending processor until a processor is assigned to the receiving task. This is because, in our scheduler, the tasks are assigned to processors only when they become ready and the sending task is unaware of this assignment until it has completed all of its required executions

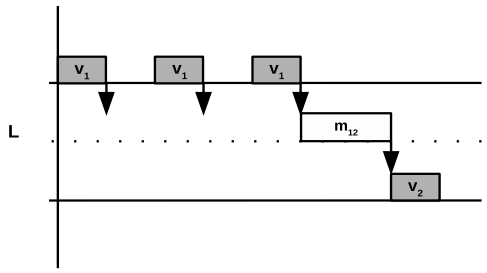


Fig. 3. Example of data transmission between two periodic tasks with different time periods

at which point all of the buffered data is transferred to the processor of the receiving task.

After generation of the ready task list, the scheduler traverses through the list of cores present in the system. If a core is free, the first task from the ready task list is assigned to it. If a parent task of the assigned task is executed on a different core, then the scheduler generates a communicating task. This communicating task is assigned to a path available between the cores that produces least communication time. The duration of this communicating task is calculated using (10) as depicted in Fig. 8. If a transmission path is unavailable or the task conflicts with other tasks assigned to the link then the child task is assigned to the other free processor and the process is repeated. This has been illustrated in Fig. 9. The conflicts for processors and links are resolved with the help of following conditions.

- A task can only be assigned to a processor p if it does not hinder the execution of the tasks already assigned to that processor. Consider a task $v_j \in V$ that needs to be assigned to a processor p that has executed *(k-1)* iterations of a task $v_i \in V$, then v_j can only be assigned if,

$$S_{v_j}[0] + w_{v_j} \leq S_{v_i}[k]$$

  where $S_{v_j}[0]$ is the start time of the iteration *0* of v_j and w_{v_j} is its corresponding computation time. $S_{v_i}[k]$ is the start time of the current periodic iteration of v_i.
- A message can only be transmitted through two routers R_a and R_b on a link L_{ab} if it does not hinder the transmission of the messages already assigned to that link. Consider a message $m_{v_k v_l}$ that needs to be transmitted from R_a to R_b through a link L_{ab} that has executed *(k-1)* iterations of a message $m_{v_i v_j}$ then the former message can only be assigned to this link if,

$$S_{m_{v_k v_l}}[0] + c(m_{v_k v_l}) \leq S_{m_{v_i v_j}}[k]$$

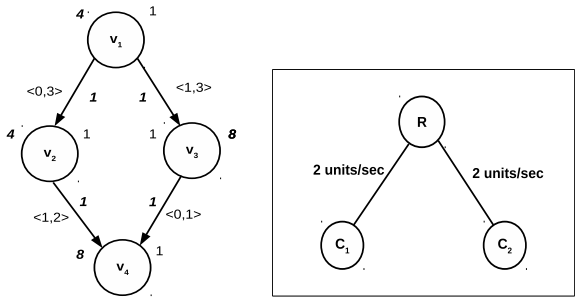
  where $S_{m_{v_k v_l}}[0]$ is the start time of the iteration *0* of $m_{v_k v_l}$ and $c(m_{v_k v_l})$ is its corresponding computation time on link L_{ab}. $S_{m_{v_i v_j}}[k]$ is the start time of the current periodic transmission of $m_{v_i v_j}$.

The algorithm continues execution until all the tasks in the DMG are successfully scheduled. The end results represent the schedule length of one complete iteration of the DMG and also the points in time at which each vertex is executed or a message is transmitted. As the DMG is periodic, each vertex repeats itself maximum amount of times required for the successful completion of its child vertexes. This amount is determined through the history interval represented on each edge of the DMG. The DMG continues its execution until the fault, is found i.e. the sink node is completely executed.

A. Basic Example

This section illustrates a basic example of the aforementioned algorithm. Consider the directed multi-query graph given in Fig. **??** that has to be scheduled on a parallel system represented by Fig. **??**. The worst case execution time of each vertex is one and the amount of data being transferred is also one. The corresponding system has one node consisting of two cores with a single router between them.

The utility factor of this DMG, calculated using (7), is 0.75 and since the time periods of the related vertexes are multiples of each other so the DMG can be scheduled. In Fig. **??**, vertex v_1 has two paths A = $< v_1, v_2, v_4 >$ and B = $< v_1, v_3, v_4 >$. Both paths have the same length, i.e. pl_A = 1+1+1+1+1 = 5 and pl_B = 5. So according to (9) the bottom level for vertex v_1, bl_{v_1} = 5. Similarly, bl_{v_2} = bl_{v_3} = 3 and bl_{v_4} = 1. Thus the scheduling order is $< v_1, v_2, v_3, v_4 >$. The hyper-period of the DMG according to (5) is 8, so according to (6), n_{v_1} = n_{v_2} = $\frac{8}{4}$ = 2 and n_{v_3} = n_{v_4} = 1. The corresponding system has a single node with two cores and one router, see Fig. **??**. As v_1 has no prior restrictions and is ready for execution, it starts its execution at core C_1 at 0 seconds and repeats itself after every 4 seconds. As both v_1 and v_2 have the same time periods, v_2 is ready for execution after 1 second. The next iteration of v_1 on C_1 starts after 4 seconds while v_2 ends its execution after 1 + 1 = 2 seconds thus it does not cause any problem with the execution of v_1. Therefore v_2 starts its execution on C_1 at 1 second and repeats itself after every 4 seconds. Now, T_{v_1} = 4 and T_{v_3} = 8, so v_1 has to be repeated twice for v_3 to be ready for execution. v_1 completes its second iteration at 5 seconds at which point v_3 is ready. Now, C_1 is not free at 5 seconds as v_2 also starts its second iteration at this point so v_3 is assigned to C_2. This means that the output data of v_1 required by v_3 needs to be transmitted from C_1 to C_2. One unit datum needs to be transmitted for one instance of v_1 - as two instances are required for v_3 so the total amount of data to be transferred is 2 units/second. The total duration for this message transmission, according to (10), is $2 \cdot [\frac{1}{2} + \frac{1}{2}]$ = 2 seconds. So the message m_{13} starts its transmission after 5 seconds and ends after 7 seconds. At this point v_3 starts its execution on C_2 and repeats after every 8 seconds. Now, v_4 requires two instances of v_2 and one instance of v_3 to start its execution. So v_4 is ready after 8 seconds but still requires data from v_2. The duration for this transmission is also 2 seconds and as it does not hinder the transmission of m_{13} so m_{24} starts its transmission after 8 seconds. Vertex v_4 starts its transmission after 10 seconds and completes at 11 seconds. Thus the total schedule length for one complete iteration of the given DMG is 11 seconds. This example has been illustrated by a time-line diagram in Fig. 5.



(a) DMG (b) Parallel System

Fig. 4. Example used to illustrate the algorithm

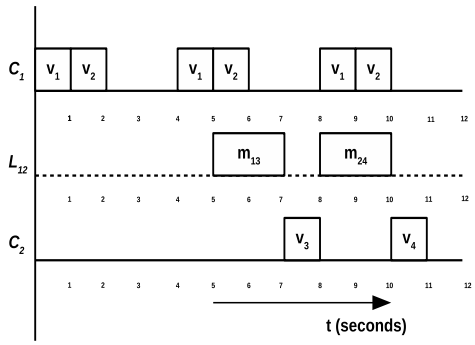


Fig. 5. Time-Line representation of the Schedule

```
1:  P ← Set of all paths in G
2:  for each path p_i ∈ P do
3:      PathLength ← PathLength_{p_i}
4:      Difference ← 0
5:      for each vertex v_i ∈ p_i do
6:          TaskID ← ID of the next vertex v_j ∈ p_i
7:          d_{v_i v_j} ← GetData_{v_i}(TaskID)
8:          Priority ← PathLength − Difference
9:          PathLength ← Priority
10:         Difference ← w_{v_i} + d_{v_i v_j}
11:         if Priority > Priority_{v_i} then
12:             Priority_{v_i} ← Priority
13:         end if
14:     end for
15: end for
```

Fig. 6. Pseudo-code representation of algorithm used to calculate Bottom Level of vertexes

B. Algorithm Complexity and Experimental Results

The worst-case time complexity of the aforementioned algorithm is $O(PVE \cdot log(V) \cdot max(N_L))$ where P represents the total number of processors in the system, V and E are the number of vertexes and edges in the DMG while max(N_L) is the maximum number of links present between two processors in the system. Here, Vlog(V) represents the worst-case time complexity of the sorting algorithm. The complexity shows that the size of the DMG effects the time taken by the algorithm to compute the schedule length. The greater the size

```
1:  V ← Set of all vertexes in G
2:  R ← Set of Ready Tasks
3:  n_v ← 0, instances need to be completed for task to execute
4:  for each vertex v_i ∈ V do
5:      Check ← true
6:      if v_i ∉ R then
7:          Parents ← Set of parents of v_i
8:          for each vertex v_j ∈ Parents do
9:              n_v ← T_{v_i} / T_{v_j}
10:             if n_v < 1 then
11:                 n_v ← 1
12:             end if
13:             if n_v ≠ completed − instances_{v_j} then
14:                 Check ← false
15:                 break
16:             end if
17:         end for
18:         if Check == true then
19:             Add v_i to R
20:         end if
21:     end if
22: end for
23: Sort R in descending order.
```

Fig. 7. Pseudo-code representation of algorithm used to generate the Ready-Task list

```
1:  function - GetDuration(inst, C1, C2, D, L)
2:  L12 ← Set of connections between C1 and C2
3:  Duration ← 0, Duration of a single message
4:  for each connection l_i in L12 do
5:      Total − Rate_{l_i} ← 0
6:      for each route r_i in l_i do
7:          Total − Rate_{l_i} = Total − Rate_{l_i} + 1 / Rate_{r_i}
8:      end for
9:  end for
10: Sort L12 in increasing order of their total rates
11: for each connection l_i in L12 do
12:     if l_i is free then
13:         Duration ← inst ∗ Data ∗ Total − Rate_{l_i}
14:         if message of this Duration is schedulable on l_i then
15:             l_i ← Active
16:             L ← l_i
17:             Return Duration
18:         end if
19:     end if
20: end for
21: Return -1
```

Fig. 8. Pseudo-code representation of algorithm used to assign links to messages

of the DMG, the greater would be the computation time of the algorithm. This scenario will be further discussed in future works.

For experimental purposes, the scheduling lengths for the DMGs given in Fig. 10 were calculated. The parallel system represented in Fig. 1 was used as the targeted system. Here, each link had a rate of 2 units/second. Both the worst-case execution time and time periods of the DMGs are given in milliseconds. The schedule length for Fig. 10a was calculated as 15 ms, for Fig. 10b it was 12 ms, for Fig. 10c it was calculated as 27 ms and for Fig. 10d it was 19 ms. The algorithm was programmed in C and executed on a linux based operating system. It took approximately 30 ms to calculate schedule lengths for Fig. 10a, 10b and 10d while it took approximately 34 ms to calculate schedule length for Fig. 10c. The results were not compared with other diagnostic techniques because the aforementioned scenario for diagnosis is different from other scenarios and the input DMG has only

```
1: E ← Set of Executing Tasks
2: R ← Set of Ready Tasks
3: Cores ← Set of all Cores
4: M ← Set of Executing Messages
5: total − time ← total time elapsed till now
6: for each core c_i ∈ C do
7:     for each task r_i ∈ R do
8:         if c_i is free then
9:             P ← Parents_{r_i}
10:            Total − Message − Duration ← 0
11:            index ← 0
12:            Messages ← ∅
13:            Link ← NULL, Connection used for message transmission
14:            for each task p_i in P do
15:                if c_i ≠ core_{p_i} then
16:                    Data ← GetData_{r_i}(index)
17:                    instances ← T_{r_i} / T_{p_i}
18:                    if instances < 1 then
19:                        instances ← 1
20:                    end if
21:                    Duration ← GetDuration_{instances,c_i}(core_{p_i}, Data, Link)
22:                    if Duration ≠ −1 then
23:                        Total − Message − Duration ← Total − Message −
                           Duration + Duration
24:                        Create a message m_{p_i r_i}
25:                        RecTask_{m_{p_i r_i}} ← r_i
26:                        SendTask_{m_{p_i r_i}} ← p_i
27:                        T_{m_{p_i r_i}} ← T_{p_i}
28:                        D_{m_{p_i r_i}} ← Duration
29:                        Link_{m_{p_i r_i}} ← Link
30:                        Add-Start-Times_{m_{p_i r_i}}(total-time)
31:                        Add m_{p_i r_i} to Messages
32:                    end if
33:                end if
34:                index++
35:            end for
36:            Start − Time ← total − time + Total − Message − Duration
37:            if r_i is schedulable on c_i then
38:                P ↘ c_i ← Active
39:                core_{r_i} ← c_i
40:                Add r_i to E
41:                Add-Start-Times_{r_i}(Start-Time)
42:                Remove r_i from R
43:                Copy contents of Messages to M
44:            else
45:                Free the links assigned to M
46:            end if
47:        end if
48:    end for
49: end for
```

Fig. 9. Pseudo-code representation of algorithm used to assign tasks to cores

been used before in [4]. Our results cannot be compared with the results described in [4] because they have considered non-cyclic queries while our diagnostic queries are periodic in nature.

V. Discussion and Future Work

In many applications of embedded systems, it is essential to provide fault tolerance and temporal guarantees to avoid accidents and failures. The present research focuses on providing a time-triggered schedule for an active diagnosis scenario that utilizes queries to identify faults within a system and propose corresponding recovery actions. The described algorithm uses a bottom level priority scheme to schedule the nodes onto free processors in such a way that the nodes on the critical path are processed first. Thus the algorithm tries to minimize the overall schedule length so that a fault can be detected within a certain time limit to ensure a successful recovery action before the system shuts itself down completely. As described through examples and implementations, the algorithm works successfully and produces a schedule of one complete iteration of the DMG. However, there is still room for improvement in the algorithm.

As illustrated in Section IV, the queries are related to each other and require data from previous intervals to complete their execution. In list schedulers, a task is assigned to a core only when all of its precedence constraints are fulfilled. Hence the data required for the execution of a task can only be transmitted to its assigned core when all of its parent tasks have completed their execution. This means that the output data of a parent task needs to be buffered till all other parent tasks are executed. This creates an unwanted delay that effects the overall schedule length. We propose a two step execution plan to solve this problem. The tasks are initially assigned to cores according to their priorities with the assumption that there is a precedence constraint but no data transfer between them. These assignments are then used to compute the actual schedule without any assumption. If according to the initial assignment, two tasks assigned to the same core hinder each other then the schedule will be recomputed until this problem is solved.

As described in Section IV, the proposed algorithm considers the shortest route of transmission between two cores but tasks are randomly assigned to the cores. Another improved feature would be to assign two related tasks either to the same core or two different cores that are closest to each other. This will reduce the communication time between the tasks and will hence improve the overall schedule length.

For future work, we will modify the algorithm according to the above mentioned scenarios to improve the overall schedule length. Moreover, currently we are assuming that there are no memory restrictions in the parallel system but we will introduce these restrictions to formulate a more realistic approach.

Acknowledgment

This work has been supported in part by the European project FP7 DREAMS under project No. 610640 and by the German Research Foundation (DFG) under project ADISTES (OB384/6-1, 629300).

References

[1] R. Isermann, R. Schwarz, and S. Stolzl, "Fault-tolerant drive-by-wire systems," *IEEE Control Systems*, vol. 22, no. 5, pp. 64–81, 2002.

[2] N. Kandasamy, J. P. Hayes, and B. T. Murray, "Time-constrained failure diagnosis in distributed embedded systems: application to actuator diagnosis," *IEEE Transactions on parallel and distributed systems*, vol. 16, no. 3, pp. 258–270, 2005.

[3] H. Kopetz, "On the fault hypothesis for a safety-critical real-time system," in *Automotive Software Workshop*. Springer, 2004, pp. 31–42.

[4] R. Obermaisser, R. I. Sadat, and F. Weber, "Active diagnosis in distributed embedded systems based on the time-triggered execution of semantic web queries," in *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on*. IEEE, 2014, pp. 222–229.

[5] G. Heiner and T. Thurner, "Time-triggered architecture for safety-related distributed real-time systems in transportation systems," in *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*. IEEE, 1998, pp. 402–407.

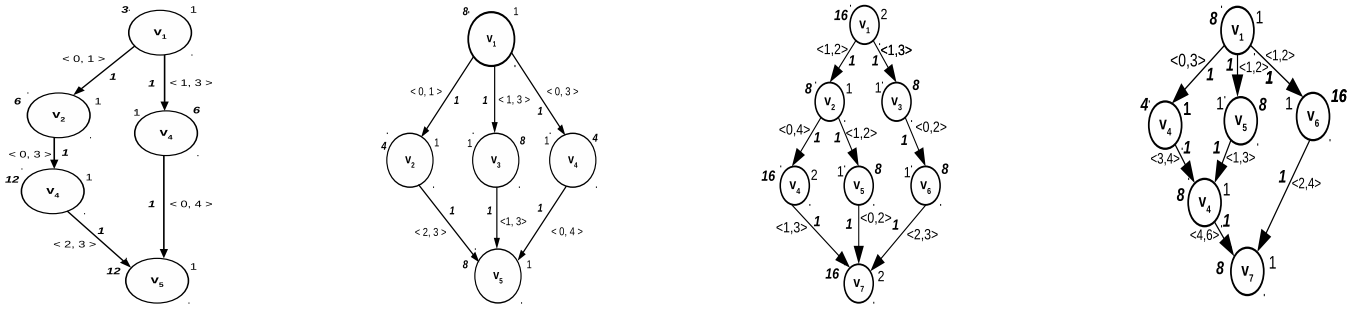


Fig. 10. DMGs used for experimentation

[6] P. M. Khilar and S. Mahapatra, "A distributed diagnosis approach to fault tolerant multi-rate real-time embedded systems," in *Information Technology,(ICIT 2007). 10th International Conference on*. IEEE, 2007, pp. 167–172.

[7] F. P. Preparata, G. Metze, and R. T. Chien, "On the connection assignment problem of diagnosable systems," *IEEE Transactions on Electronic Computers*, no. 6, pp. 848–854, 1967.

[8] M. Barborak, A. Dahbura, and M. Malek, "The consensus problem in fault-tolerant computing," *aCM Computing Surveys (CSur)*, vol. 25, no. 2, pp. 171–220, 1993.

[9] R. C. Dorf and R. H. Bishop, "Modern control systems," 1998.

[10] A. Pelc, "Optimal fault diagnosis in comparison models," *IEEE Transactions on Computers*, vol. 41, no. 6, pp. 779–786, 1992.

[11] A. Sengupta and A. T. Dahbura, "On self-diagnosable multiprocessor systems: diagnosis by the comparison approach," *IEEE Transactions on Computers*, vol. 41, no. 11, pp. 1386–1396, 1992.

[12] C. J. Walter, P. Lincoln, and N. Suri, "Formally verified on-line diagnosis," *IEEE Transactions on Software Engineering*, vol. 23, no. 11, pp. 684–721, 1997.

[13] A. T. Dahbura, K. K. Sabnani, and W. J. Hery, "Spare capacity as a means of fault detection and diagnosis in multiprocessor systems," *IEEE Transactions on Computers*, vol. 38, no. 6, pp. 881–891, 1989.

[14] S. Tridandapani, A. K. Somani, and U. R. Sandadi, "Low overhead multiprocessor allocation strategies exploiting system spare capacity for fault detection and location," *IEEE Transactions on Computers*, vol. 44, no. 7, pp. 865–877, 1995.

[15] L. Portinale and D. Codetta-Raiteri, "Using dynamic decision networks and extended fault trees for autonomous fdir," in *Tools with Artificial Intelligence (ICTAI), 2011 23rd IEEE International Conference on*. IEEE, 2011, pp. 480–484.

[16] T. Jiang, K. Khorasani, and S. Tafazoli, "Parameter estimation-based fault detection, isolation and recovery for nonlinear satellite models," *IEEE Transactions on control systems technology*, vol. 16, no. 4, pp. 799–808, 2008.

[17] F. Hutter and R. Dearden, "Efficient on-line fault diagnosis for non-linear systems," in *Proceedings of the 7th international symposium on artificial intelligence, robotics and automation in space*, 2003.

[18] S. E. Chodrow, F. Jahanian, and M. Donner, "Run-time monitoring of real-time systems," in *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*. IEEE, 1991, pp. 74–83.

[19] M. Diaz, G. Juanole, and J.-P. Courtiat, "Observer-a concept for formal on-line validation of distributed systems," *IEEE Transactions on Software Engineering*, vol. 20, no. 12, pp. 900–913, 1994.

[20] F. Jahanian, R. Rajkumar, and S. C. Raju, "Runtime monitoring of timing constraints in distributed real-time systems," *Real-Time Systems*, vol. 7, no. 3, pp. 247–273, 1994.

[21] B. Plale and K. Schwan, "Run-time detection in parallel and distributed systems: Application to safety-critical systems," in *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*. IEEE, 1999, pp. 163–170.

[22] S. Sankar and M. Mandal, "Concurrent runtime monitoring of formally specified programs," *Computer*, vol. 26, no. 3, pp. 32–41, 1993.

[23] G. Tortora, "Fault-tolerant control and intelligent instrumentation," *Computing & Control Engineering Journal*, vol. 13, no. 5, pp. 259–262, 2002.

[24] J. C. Yang and D. W. Clarke, "The self-validating actuator," *Control Engineering Practice*, vol. 7, no. 2, pp. 249–260, 1999.

[25] O. Sinnen, *Task scheduling for parallel systems*. John Wiley & Sons, 2007, vol. 60.

[26] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002.

[27] H. Kasahara and S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," *IEEE Transactions on Computers*, vol. 33, no. 11, pp. 1023–1029, 1984.

[28] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE transactions on parallel and distributed systems*, vol. 7, no. 5, pp. 506–521, 1996.

[29] Y. K. Kwok and I. Ahmad, "Link contention-constrained scheduling and mapping of tasks and messages to a network of heterogeneous processors," *Cluster Computing*, vol. 3, no. 2, pp. 113–124, 2000.

[30] M.-Y. Wu and D. D. Gajski, "Hypertool: A programming aid for message-passing systems," *IEEE transactions on parallel and distributed systems*, vol. 1, no. 3, pp. 330–343, 1990.

[31] L. Cucu and Y. Sorel, "Non-preemptive multiprocessor scheduling for strict periodic systems with precedence constraints," in *Proceedings of the 23rd Annual Workshop of the UK Planning and Scheduling Special Interest Group, PLANSIG*, vol. 4, 2004.

[32] G. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 2011, vol. 24.

[33] O. Kermia and Y. Sorel, "A rapid heuristic for scheduling non-preemptive dependent periodic tasks onto multiprocessor," in *Proceedings of ISCA 20th international conference on Parallel and Distributed Computing Systems, PDCS'07*, 2007.