Cui XT, Wu KJ, Wei TQ *et al.* Worst-case finish time analysis for DAG-based applications in the presence of transient faults. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 31(2): 267–283 Mar. 2016. DOI 10.1007/s11390-016-1626-6

Worst-Case Finish Time Analysis for DAG-Based Applications in the Presence of Transient Faults

Xiao-Tong Cui [1,2], Kai-Jie Wu [1,2,*], *Member, CCF, IEEE*, Tong-Quan Wei [3,*], *Member, CCF, IEEE*, and Edwin Hsing-Mean Sha [1,2], *Member, CCF, IEEE*

[1] *Key Laboratory of Dependable Service Computing in Cyber Physical Society, Chongqing University Chongqing 400044, China*

[2] *College of Computer Science, Chongqing University, Chongqing 400044, China*

[3] *Department of Computer Science and Technology, East China Normal University, Shanghai 200241, China*

E-mail: {xiaotong.sd, kaijie}@gmail.com; tqwei@cs.ecnu.edu.cn; edwinsha@gmail.com

Received January 12, 2015; revised April 21, 2015.

Abstract Tasks in hard real-time systems are required to meet preset deadlines, even in the presence of transient faults, and hence the analysis of worst-case finish time (WCFT) must consider the extra time incurred by re-executing tasks that were faulty. Existing solutions can only estimate WCFT and usually result in significant under- or over-estimation. In this work, we conclude that a sufficient and necessary condition of a task set experiencing its WCFT is that its critical task incurs all expected transient faults. A method is presented to identify the critical task and WCFT in $O(|V|+|E|)$ where $|V|$ and $|E|$ are the number of tasks and dependencies between tasks, respectively. This method finds its application in testing the feasibility of directed acyclic graph (DAG) based task sets scheduled in a wide variety of fault-prone multi-processor systems, where the processors could be either homogeneous or heterogeneous, DVS-capable or DVS-incapable, etc. The common practices, which require the same time complexity as the proposed critical-task method, could either underestimate the worst case by up to 25%, or overestimate by 13%. Based on the proposed critical-task method, a simulated-annealing scheduling algorithm is developed to find the energy efficient fault-tolerant schedule for a given DAG task set. Experimental results show that the proposed critical-task method wins over a common practice by up to 40% in terms of energy saving.

Keywords fault tolerance, worst-case analysis, simulated annealing, energy conservation, dynamic voltage scaling (DVS)

1 Introduction

Real-time embedded systems which are used for mission-critical applications such as navigation systems, process control, and surveillance systems demand a high level of fault tolerance. Typically, static task scheduling mechanisms[1-3] are often used in real-time systems, which are usually designed to tolerate up to a specified number of transient faults in a given time interval by reserving enough redundancy (including hardware and/or time) to recover from fault(s). The upper bound on the number of faults is derived from the rate of faults under the designated operating conditions, i.e., the mean time to failure (MTTF) of hardware components and the rate of single event upsets (SEU). Due to the severe resource constraints of embedded systems, optimizing redundancy overheads is of great importance. Among all the fault tolerance techniques, re-execution-based fault tolerance schemes have received broad attention due to their efficiency of resource usage and easiness in system design.

Re-execution based schemes usually assume that the transient faults that occur during the execution of a task are detected by the concurrent error detection

Regular Paper

A preliminary version of the paper was published in the Proceedings of ASP-DAC 2014.

This work is partially supported by the National High Technology Research and Development 863 Program of China under Grant Nos. 2015AA015304 and 2013AA013202, the National Natural Science Foundation of China under Grant No. 61472052, and Chongqing Research Program under Grant No. cstc2014yykfB40007.

*Corresponding Author

mechanisms (e.g., watchdogs) supported by the processor. Therefore, the focus is on determining and scheduling the re-executions of the (faulty) tasks. The schemes can be broadly classified into two categories — passive and active. An active scheme replicates tasks on multiple processing units such that all copies are executed (almost) simultaneously. If one of the replicas is found faulty, its result is replaced by the result of another replica. A passive scheme, on the other hand, executes only the primary replica of a task. Secondary replicas are executed only if the primary replica is found faulty. Feasibility tests of both strategies need to estimate the amount of slack that can be devoted to tolerating the worst-case fault occurrences. Several strategies have been developed for a variety of applications[4-8].

Recently, minimizing power and/or energy consumption gains considerable attention for real-time embedded systems since many of these systems are severely energy-constrained. Even if passive re-execution schemes are adopted for the sake of energy, it is important to note that reserving more than necessary time for passive replicas degrades the opportunities to engage power management techniques such as dynamic voltage scaling (DVS). Hence, the knowledge of the precise worst-case finish time of a task set in the presence of faults is critical not only to testing feasibility but also to minimizing energy consumption. Speaking of minimizing energy consumption, heterogeneous computing environment is very popular[9-11] since it provides benefit for energy conservation. Compared with homogeneous computing environments, different types of processors with diverse performance and power cost are used for optimizing energy cost in heterogeneous environments. Hence it can be used for specific applications which are energy-constrained.

Though feasibility analysis of fault-prone real-time systems has been studied for several years, the majority of the attention has been paid to the task sets that comprise only independent tasks. Ghosh *et al.* studied the problem by assuming the intervals between any two faults are no less than twice the longest task[12]. Burns *et al.* continued the work by assuming a similar fault model[13]. They extended the feasibility analysis given in [14], by modifying the analysis of task response time to include fault-induced additional processing requirements. Liberato *et al.* presented an efficient algorithm with the time complexity of $O(N^2X)$, where N is the number of tasks and X is the maximum number of faults to be tolerated[15]. Their work was extended by Aydin to allow multiple re-executions of a task to have different execution time[16]. Chrobak *et al.* presented several fast algorithms to determine the feasibility of a task schedule based on several different fault models[17]. Thekkilakattil *et al.*[18] guaranteed "fault tolerance feasibility" by resource augmentation, specifically through processor speed-up.

None of these approaches[12-18], however, handles the task sets that have inter-task dependencies and are scheduled in multi-processor systems. For such task sets, the classical processor demand analysis does not work anymore. This is because processors could be idle before all tasks are executed due to inter-task dependencies. In [7], this is simply done by comparing all possible cases and choosing the worst one. Such straightforward approach does not scale well as the number of cases grows rapidly while finding the worst-case finish time is always the most important issue in real-time multi-processor system design.

In this paper, we will present such a method that tests the exact feasibility in $O(|V|+|E|)$ where $|V|$ and $|E|$ are the number of tasks and dependencies between tasks, respectively. Task sets of many real-time applications are usually defined using processing graphs, such as directed acyclic graphs (DAGs)[19-20]. The method finds its application in testing the feasibility of the DAG task set scheduled on a wide variety of multi-processor systems, where the processors could be either homogeneous or heterogeneous, DVS-capable or DVS-incapable, etc. The schedule of a DAG task set includes the task-to-processor mapping, the task-to-task order on each processor, and the task-to-speed assignment of each task if its mapped processor supports DVS. Experimental results show that the common practices could either underestimate the worst case by up to 25%, or overestimate by up to 13%. Based on this method, a simulated-annealing scheduling algorithm is developed to find the energy efficient fault-tolerant schedule for a given DAG task set.

In summary, the contributions of this paper are as follows.

1) We prove that for a schedule of a DAG task set, there is at least one task such that the schedule will experience its worst-case finish time (WCFT) when this task incurs all the expected transient faults. The task is named as critical task. The critical-task theory gives the sufficient and necessary condition of a schedule experiencing its WCFT.

2) We develop an efficient feasibility test method that identifies the critical task and the WCFT of a schedule, and runs in $O(|V|+|E|)$ where $|V|$ and $|E|$ are

the number of tasks and dependencies between tasks, respectively. This is referred to as critical-task method. It is important to note that unlike the existing solutions that take the same time complexity but could result in significant under- or over-estimation, the proposed method finds the exact WCFT.

3) We propose, as an example application of the critical-task method, a simulated annealing algorithm to find the energy efficient fault-tolerant schedule for a DAG task set scheduled on a heterogeneous DVS-capable multi-processor system. The algorithm uses the proposed critical-task method for feasibility test.

The rest of this paper is organized as follows. Section 2 presents the system models. Section 3 presents the critical-task theory and Section 4 presents the critical-task method. Section 5 gives an example application of the proposed critical-task method. It is an SA-based algorithm where the critical-task method is used for feasibility test. Section 6 shows the experimental results, and Section 7 concludes this paper.

2 System Models

2.1 System Architecture

Consider a closely-coupled M-processor real-time system

$$R = \{p_1, p_2, ..., p_M\},$$

where $M \geqslant 1$. Although processors used in examples are assumed to be identical, the analysis and conclusions directly apply to heterogeneous systems comprising of processors with different characteristics. Communications in closely-coupled real-time systems are carefully designed to guarantee invariable, minimum, and known overheads. This is often achieved by using shared memory or special communication protocols, such as ultra-fast serial links. In a multi-processor system, inter-task communication overhead is determined 1) by the memory access latency and bandwidth in the case of shared-memory architecture, or 2) by the end-to-end link latency and bandwidth in the case of a networked architecture. In this work, a common communication model described below is used for the two architectures.

• Communication between a pair of tasks scheduled on the same processor incurs constant worst-case delay. For simplicity, the example assumes the delay is 0.

• Communication between a pair of tasks scheduled on different processors incurs constant worst-case delay.

• Communication between a task and the tasks scheduled on different processors may experience different worst-case delays.

We believe the proposed communication delay model is practical and accurate in the context of closely-coupled systems. The underlying assumption is that the worst-case outputs generated by the primary and secondary replicas of a task, in terms of memory space or communication traffic, are identical and hence the inter-task communication delay between a given pair of tasks remains independent of fault occurrences of tasks.

2.2 DAG Task Set

For a task set with data dependencies, the WCFT of a task depends on 1) itself, 2) the tasks where it receives inputs, and 3) the task scheduled immediately before it on the same processor — its schedule predecessor. Fig.1(a) shows a task set with data dependencies (shown using arrows) and the worst-case execution time (referred to as C) of each task. For simplicity, the example assumes that the worst-case execution time equals the re-execution time of task T when it is re-executed upon a fault, i.e., $C_T = Cr_T$, but this assumption is not required for the analysis to hold. Fig.1(b) shows a schedule that preserves the data dependencies and assumes the worst-case intra- and inter-processor communication delays between any tasks are 0 and 1 cycle, respectively. Schedule predecessors are introduced after the schedule is determined, and are shown using dotted lines in Fig.1(c). For example, T_3 is T_6's schedule predecessor, and T_6 cannot begin its execution until T_3 finishes. It is not necessary to distinguish data and schedule dependencies during the analysis of a given schedule.

In our analysis, the task graph is converted to a weighted directed acyclic graph (WDAG), $G = (V, E)$, where V is a set of vertices (tasks) and E is a set of directed edges (schedule and data dependencies), as shown in Fig.1(d). We hence use $|V|$ and $|E|$ to represent the number of tasks and dependencies between tasks, respectively. In the following context, tasks and dependencies are used interchangeably with vertices and edges respectively. The edge from T_i to T_j has a weight equal to the communication delay between T_i and T_j, and is denoted as $W_E(T_i, T_j)$. A vertex T also has a weight, denoted as $W_v(T)$, equal to the C_T of the task if it incurs zero fault, or $C_T + xCr_T$ if it incurs x transient faults. All weights are non-negative. Two dummy vertices, $Source$ and $Sink$, are added into G.

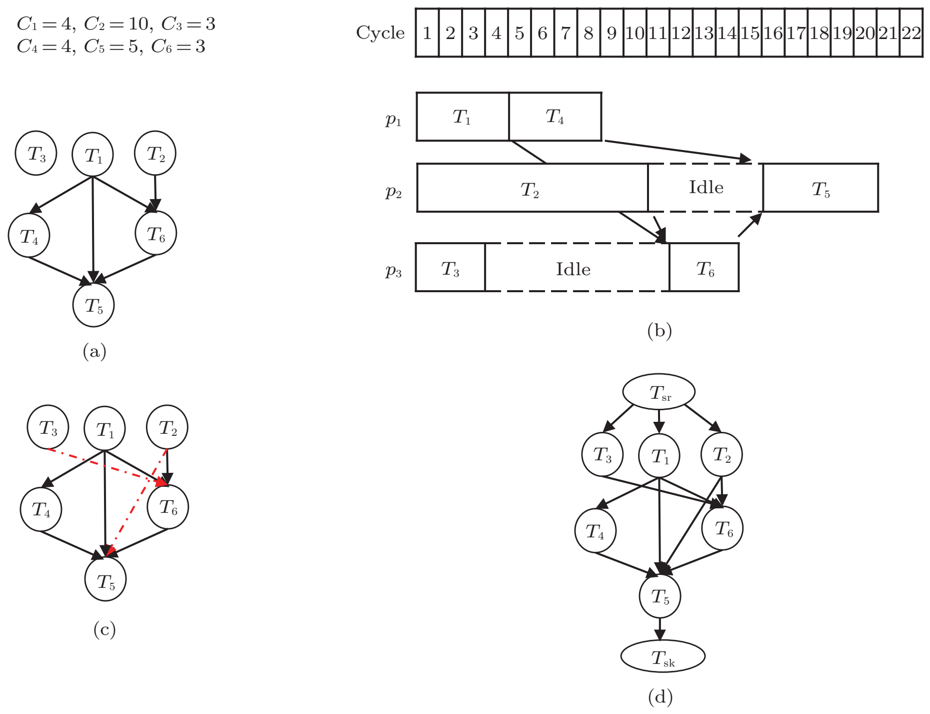

$C_1=4$, $C_2=10$, $C_3=3$
$C_4=4$, $C_5=5$, $C_6=3$
(a)
Cycle 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
p_1 T_1 T_4
p_2 T_2 Idle T_5
p_3 T_3 Idle T_6
(b)
(c)
T_{sr}
T_{sk}
(d)

Fig.1. Example of task set and its schedule. (a) Data dependency. (b) Schedule of tasks. (c) Data/schedule dependency. (d) WDAG.

The *Source* vertex, denoted as $T_{\rm sr}$ in Fig.1(d), has an edge to each vertex that has no incoming edge in the original G; and the *Sink* vertex, denoted as $T_{\rm sk}$, has an edge from each vertex that has no outgoing edge in the original G. The weights of both *Source* and *Sink* are 0 and the weights of the edges from *Source* and to *Sink* are 0. A parent of vertex T_i is a vertex with an edge to T_i. Vertex T_i is then a child of its parents.

A simple path from vertex T_i to vertex T_j, denoted as $Path(T_i, T_j)$, is a sequence of non-repeated vertices and edges by which the start vertex T_i and the end vertex T_j are connected together. Note that there may be no path, one path, or more than one path between a pair of vertices. Since the graph is directed and acyclic, paths are directed as well. If $Path(T_i, T_j)$ exists, $Path(T_j, T_i)$ must not exist. An ancestor of vertex T_i is a vertex that has at least one path to T_i. Vertex T_i is then a descendant of its ancestors. A parent (child) of a vertex is one of the ancestors (descendants) of the vertex. The length of a path, denoted as $|Path(T_i, T_j)|$, is the summation of the weights of the vertices and edges along the path. The longest path from vertex T_i to vertex T_j, denoted as $LPath(T_i, T_j)$, is the path that has the longest length among all paths from vertex T_i to vertex T_j, and its length is denoted as $|LPath(T_i, T_j)|$. There may be more than one longest path between a pair of vertices, and all of them have the same length.

A critical path of vertex T, denoted as $CPath(T)$, is one of the longest paths from the *Source* $T_{\rm sr}$ to vertex T. A vertex may have multiple critical paths, and all of them have the same length, which is denoted as $|CPath(T)|$ and is equal to $|LPath(T_{\rm sr}, T)|$. It is easy to see that the finish time of task T is equal to $|CPath(T)|$, and the start time of the task is equal to $|CPath(T)| - W_v(T)$, where $W_v(T) = C_T$ if T incurs zero fault, or $C_T + xCr_T$ if T incurs x faults. It is important to note that the length of a path, the longest paths between two tasks, and the critical paths of a vertex vary with the fault occurrences of involved vertices. Hence, a different case of fault occurrences will result in a different WDAG because the weights of involved vertices have changed. Let Tc be the current task under investigation, $BCFT_{Tc}$ be its best-case finish time when no fault occurs, and $WCFT_{Tc}$ be its worst-case finish time when all X faults have occurred during or before its execution. In order to derive $WCFT_{Tc}$, two useful task subsets are introduced:

- Parents set of Tc (PS_{Tc}): task subset comprising the parents of Tc;
- Ancestors set of Tc (AS_{Tc}): task subset comprising all ancestors of the current task Tc.

It is easy to see that only the fault occurrences of the tasks in AS_{Tc} could affect the start time of Tc. Obviously, $PS_{Tc} \subseteq AS_{Tc}$.

2.3 Frame-Based Systems for DAG Task Set

Presently there are two popular control paradigms of task sets in real-time systems. One is the frame-based system and the other is the event-triggered system. In a frame-based system, all tasks are re-leased at time 0 and should be finished by the end of the frame. In an event-triggered system, each task is released according to its own period and is executed based on its order of priority. In general, while an event-triggered system is preferred by non-safety-critical applications due to its higher flexibility in resource allocation and its higher efficiency in resource sharing, a frame-based system is more preferred in safety-critical applications due to its better hard real-time performance and easier incorporation of fault tolerance[21-26]. This work investigates frame-based systems.

A scheduler first allocates the tasks to the processors according to some partitioning heuristics, and then derives the order among tasks in each processor. Hence, a task set is "scheduled" if the task-to-processor mapping and the task-to-task order in each processor are determined. If the processors support DVS, the execution speed of each task is also assigned. Each task is defined by its worst-case execution time C that denotes the maximum CPU time required by the fault-free execution of the task. C is the product of the worst-case CPU cycles of the task and the clock period of its host processor. Since the proposed analysis technique is applied after the schedule of tasks is determined, C of a task is a constant. The duration of a frame is denoted by D by which all tasks must finish their execution even in the presence of faults[27]. A dependent task cannot begin execution until all the tasks from which it expects inputs complete their executions. Due to non-negligible inter-processor delays, the starting time of a dependent task may be delayed further if input data are communicated from tasks on other processor(s). While this analysis does not explicitly consider control dependencies where the workload may change depending on the run-time resolutions, the proposed technique can still be applied if the control dependencies are cleared, e.g., by either taking one of possible outcomes or considering the worst case of all outcomes.

2.4 Fault Model and Re-Execution

In hardware systems, faults can be permanent, transient, or intermittent. In deep sub-micron and nano regimes, transient faults are the most common faults due to the increasing susceptibility to radiations — a result of the continuously rising level of the integration in semiconductor devices. The number of transient faults that may occur in a frame depends on the fault susceptibilities of the underlying system. For a scheduled task set, the maximum number of transient faults X is defined in the way that the probability of incurring more than X faults in a time frame can be safely ignored. Let X denote the number of fault occurrences in a task, and then the sum of fault occurrences of all tasks in a task set is no larger than X.

Fault occurrences during a task execution are manifested as erroneous outputs. It is assumed that faults are detected by concurrent error detection techniques[28]. Depending on the capabilities of deployed error detection mechanisms, a fault may be detected immediately after its occurrence or by the end of task execution. Immediate fault detection enables earlier re-executions, thereby saving both energy and time. However, since this study focuses on the worst-case finish time, it is assumed that a fault is detected at the end of task execution. Since the re-executions of a task can also incur faults, X fault occurrences in a row (i.e., burst faults) may affect the primary execution and up to the $(X-1)$-th re-executions, and the X-th re-execution finishes fault-free. The worst-case execution time of executing and re-executing a task T is denoted by C_T and Cr_T respectively. We assume each task has only one version of re-execution and the re-execution will be carried right after its primary execution and on the same processor, which is the case of many fault-tolerant systems[6-8,27]. Hence the re-executions of a task have the same time Cr_T. In DVS-capable systems where re-executions could be on different frequencies, it is suggested to always schedule them using the maximal frequency for the sake of energy[29]. Scheduling re-executions using maximal frequency is equivalent to devoting all slack to slowing down normal tasks executions. This strategy minimizes the energy consumption in the situation where no fault occurs, which is the most-occurred situation since fault occurrences are low-probability events. Therefore, if T incurs total X faults, its actual execution time is $C_T + XCr_T$. Many re-execution based fault recovery techniques assume that a task's primary execution and re-executions use the same version of binary code, and therefore take the same amount of time, i.e., $C_T = Cr_T$. Our investigation relaxes the constraint to include the case where $C_T \neq Cr_T$ for the tasks using different (e.g., lighter) versions of binary code for re-executions.

2.5 Problem to Be Addressed

Formally, given a scheduled task set and the maximal number of faults that could occur during a frame, one wants to find the feasibility of the schedule by finding its worst-case finish time. Again, a task set is scheduled if the task-to-processor mapping and the task-to-task order in each processor are determined. Without the schedule information, one cannot determine the feasibility of a task set since its worst-case finish time varies significantly with different schedules. Also, because of this, the proposed technique works on both homogeneous and heterogeneous systems.

The problem is trivial if there are no data dependencies among tasks. In this case, the worst-case finish time of the tasks mapped on a processor can be independently determined by considering only one case: the task with the longest re-execution time of this processor incurs all the faults. The task with the longest re-execution time will be simply referred to as the longest task. The problem, however, becomes more difficult if otherwise. This is because that if a task's data dependencies are not cleared, it may not be executed even if its schedule dependency is cleared, i.e., the processor is idle. From our work, we can identify two common practices.

- *Common Practice* 1. Assume the longest task incurs the maximal number of faults. The worst-case finish time could be underestimated.
- *Common Practice* 2. Reserve the slack for recovery by simply multiplying the re-execution time of the longest task with the maximal fault number. The worst-case finish time could be overestimated.

A motivational example is shown in Fig.2 where tasks are scheduled on three processors. The arrows in the figure show the data dependencies between tasks. The communication delay between tasks scheduled on the same processor or different processors is assumed to be 0 and 1, respectively. The original schedule is shown in Fig.2(a). It is assumed that at most two faults occur. Fig.2(b) shows the true worst case where T_2 incurs all faults, Fig.2(c) shows the underestimation by common practice 1 where T_4, the longest task, is assumed to incur all faults, and Fig.2(d) shows the overestimation by common practice 2 which reserves the slack by multiplying the re-execution time of the longest task T_4 twice directly. The shaded tasks in the figure are the faulty tasks.

3 Critical-Task Theory

In this section, we will prove that for any task Tc, there exists at least one critical task in $AS_{Tc} \cup \{Tc\}$, such that when this critical task incurs all the faults, Tc experiences the worst-case finish time. It is important to note that the critical task of Tc may not be the longest task in AS_{Tc}. We will start by proving Lemma 1∼Lemma 3.

Lemma 1. *Given Tc and a faulty task T, and T is not in any of the critical paths of Tc. All the criti-*

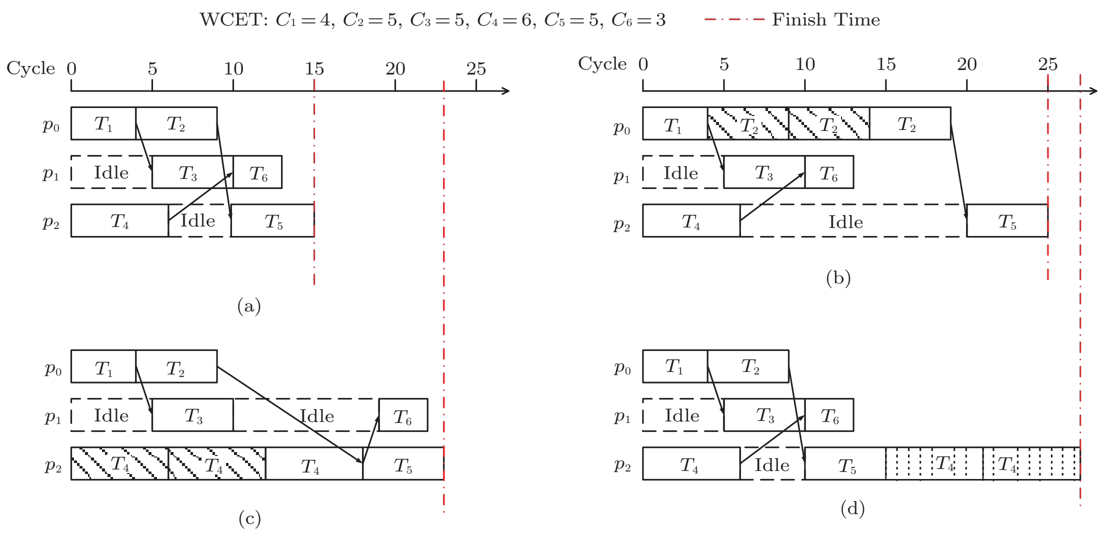

WCET: $C_1=4$, $C_2=5$, $C_3=5$, $C_4=6$, $C_5=5$, $C_6=3$
Finish Time
Cycle 0 5 10 15 20 25
p_0 p_1 p_2
T_1 T_2 T_3 T_4 T_5 T_6
Idle
(a)
(b)
(c)
(d)

Fig.2. Motivational example. (a) Original schedule of tasks. (b) Task T_2 incurs all 2 faults, which results in the real worst case with the finish time being 25. (c) Common practice 1, which assumes the longest task T_4 incurs all 2 faults, underestimates the real worst case. (d) Common practice 2, which reserves the slack by multiplying the re-execution time of the longest task T_4 twice, overestimates the real worst case.

cal paths of Tc when T incurs x faults are still critical paths when T incurs x' faults, and the finish time of Tc does not change, as long as $x > x' \geqslant 0$.

Explanation. This lemma has two implications. First, the critical paths of Tc when T incurs x faults will remain as the critical paths of Tc when T incurs x' faults, as long as $x > x'$. Second, if a path from the *Source* to Tc was not a critical path when T incurred x faults, it will not become a critical path when T incurs x' faults, as long as $x > x'$.

Proof. On one hand, because $x' < x$, the weight of vertex T is reduced, which further reduces the lengths of all the paths from *Source* to Tc that include T. On the other hand, since the length of a path depends only on the weights of the vertices and edges along the path, the change of T's fault occurrences will not change the length of the paths that do not include T, including those critical paths. And since no path will get a longer length, those critical paths remain as critical paths. Therefore the finish time of Tc does not change. This proves Lemma 1. □

Lemma 2. *Assume there are two tasks, T_1 and T_2, which are in AS_{Tc} and incur x_1 and x_2 faults respectively. A worse or status-quo finish time of Tc can always be found by letting a task $T \in AS_{Tc}$ incur all the $x_1 + x_2$ faults. T could be either T_1 or T_2, or another task in AS_{Tc}.*

Proof. If one of the two tasks, say T_1, is not in any critical path of Tc, a worse finish time of Tc can be easily obtained by letting T_1 incur 0 fault, and letting any task that is in a critical path of Tc incur x_1 more faults. This is because, from Lemma 1, as the number of faults occurring in T_1 decreases, all critical paths of Tc will not be affected and the finish time of Tc does not change since T_1 is not in any critical path of Tc. On the other hand, if there is a task T from a critical path of Tc which incurs x_1 more faults, the length of this critical path increases, so does the WCFT of Tc. This proves Lemma 2 if either T_1 or T_2 or neither is in critical paths of Tc.

If both T_1 and T_2 are in the critical paths of Tc, there could be two cases. The first case is that there exists one critical path that includes T_1 but not T_2. The critical path is thus denoted as $CPath(Tc)_{T_1}$. The second case is that critical paths that include T_1 always include T_2. For both cases, we have the following rearrangements of faults.

• *Rearrangement* 1. Let T_1 incur $x_1 + 1$ faults and let T_2 incur $x_2 - 1$ faults.

• *Rearrangement* 2. Let T_1 incur $x_1 - 1$ faults and let T_2 incur $x_2 + 1$ faults.

For the first case, rearrangement 1 reallocates one fault from T_2 to T_1 and the total number of faults remains unchanged. $|CPath(Tc)_{T_1}|$ will be increased by the amount of Cr_{T_1}, which will result in a worse finish time of Tc. Repeating this rearrangement will further worsen the finish time of Tc until T_1 incurs $x_1 + x_2$ faults and T_2 incurs zero fault. This proves Lemma 2 under this situation.

For the second case, rearrangement 1 will change the length of the critical paths that include both T_1 and T_2 by the amount of $Cr_{T_1} - Cr_{T_2}$. If $Cr_{T_1} > Cr_{T_2}$, the length of the path is increased, thereby resulting in a worse finish time of Tc. Repeating this rearrangement will further worsen the finish time of Tc until T_1 incurs $x_1 + x_2$ faults and T_2 incurs zero fault. This proves Lemma 2. If $Cr_{T_1} < Cr_{T_2}$, rearrangement 2 will increase the length of the critical path, thereby resulting in a worse WCFT of Tc. Repeating this rearrangement will further worsen the WCFT of Tc until T_2 incurs $x_1 + x_2$ faults and $T1$ incurs zero fault. This proves Lemma 2. There is a special third case where $Cr_{T_1} = Cr_{T_2}$. Letting either T_1 or T_2 incur all the $x_1 + x_2$ faults will reach a status-quo finish time of Tc. This also proves Lemma 2. □

Lemma 3. *If more than one task in AS_{Tc} incurs faults, a worse or status-quo finish time of Tc can always be found by letting one task $T \in AS_{Tc}$ incur all the X faults.*

Proof. Lemma 3 is a corollary of Lemma 2. Let us denote $T_1, T_2, \ldots, T_k$ as these faulty tasks and $x_1, x_2, \ldots, x_k$ as their fault occurrences respectively, $\sum_{i=1}^{k} x_i = X$. Lemma 3 can be proved by repeating Lemma 2. Every time two random faulty tasks are picked, a worse-case finish time of Tc can be obtained by rearranging the faults occurring in the two faulty tasks to the one in a critical path of Tc, or to a third task if neither of the two tasks is in a critical path. This procedure continues until there is only one faulty task T that incurs total X faults. Since the finish time of Tc is not advanced during all these rearrangements, Lemma 3 is proved. □

Now it is time to prove the critical-task theory as stated in Theorem 1 which gives a sufficient condition for Tc to experience the worst-case finish time.

Theorem 1. *There exists at least one task for any task Tc such that if this task incurs all the expected X faults, task Tc experiences its worst-case finish time. This task, which could be Tc itself or a task in AS_{Tc},* *is thus referred to as the critical task* (Tct) *of* Tc.

Proof. Theorem 1 can be proved by contradiction. The counter statement is that Tc will only experience the worst-case finish time when more than one task incurs faults. This statement directly contradicts to Lemma 3. □

We can further show the other aspect of the critical-task theory as stated in Theorem 2.

Theorem 2. *A* (*weak*) *necessary condition for any task* Tc *experiencing its worst-case finish time is that one of its critical tasks incurs all the expected* X *faults.*

The necessary condition is a weak one because there could be the cases where another pattern of fault occurrence may result in the same worst case, but never worse than the case where one of Tc's critical tasks incurs all the expected X faults. An example is that a task could have more than one critical task sitting in the same critical path of this task. The case where a single critical task catches all the faults results in the same worst case as the case where more than one of those critical tasks catch all the faults.

Theorem 1 and Theorem 2 can be easily extended to identify the critical task of a processor, and then the critical task of the task set. The critical task of a processor is the critical task of the last task scheduled on this processor, and the critical task of the task set is the critical task of $Sink$. Unfortunately, the critical task of a task set may not be the longest task in the set, and hence cannot be identified easily. We hence propose a recursive method that identifies the critical task and the WCFT of a task set in $O(|V|+|E|)$ where $|V|$ and $|E|$ are the number of tasks and dependencies between tasks, respectively.

4 Critical-Task Method to Identify the Worst-Case Finish Time

Consider a task set that consists of $|V|$ tasks and is subject to X faults. There are total $\binom{|V|+X-1}{X}$ distinct cases of fault occurrences[16]. Without the knowledge from the previous section, one needs to compute the WCFT for each of these cases, and then chooses the worst of them. To compute the WCFT for a given fault occurrence, one needs to determine the longest path from $Source$ to $Sink$ which takes $O(|V|^2)$[30]. Therefore the overall time complexity could be

$$O\Big(\frac{(|V|+X-1)!}{X!(|V|-1)!}|V|^2\Big).$$

We here propose a more efficient method that identifies the critical task and calculates the WCFT of a task in $O(|V|+|E|)$ time. Note that while common practice 2 simply reserves the slack by multiplying the re-execution time of the longest task with the fault number, it still needs a Breadth-First Search (or Depth-First-Search) algorithm to identify the "worst-case" finish time. Hence it still incurs $O(|V|+|E|)$ time. The supporting lemma and theorem are given below.

Lemma 4. *If a task is not a critical task of any parents of* Tc, *it will not be a critical task of* Tc.

Proof. According to Theorem 1, Tc will experience its worst-case finish time when either Tc or one of the tasks in AS_{Tc} incurs all the faults. Apparently, Lemma 4 is applicable only when Tc is not a critical task of itself. The following proof thus assumes Tc is fault-free.

Tc could have more than one parent and each parent has its own critical tasks. A parent will experience the WCFT when one of its critical tasks incurs all the faults. Let us denote the random task as $Tnct$. According to the condition of the lemma, $Tnct$ is not a critical task of any of these parents. If $Tnct$ incurs all the faults, none of the parents of Tc will experience their WCFT. As a result, Tc will not experience the WCFT. This indicates that $Tnct$ is not a critical task of Tc. □

Essentially, Lemma 4 tells that the critical tasks of Tc can only be Tc itself or the critical tasks of Tc's parents. To determine the WCFT of task Tc, we have the following two cases.

Case 1. Tc is its own critical task and it incurs all X faults while all tasks in AS_{Tc} finish fault-free. All parents of Tc thus experience the best-case finish time (BCFT). The BCFT of each task can be calculated as long as the schedule is determined (i.e., no need to know the actual fault occurrences). Since Tc incurs all X faults, the re-execution time of Tc for tolerating X faults is $X \times Cr_{Tc}$. The WCFT of Tc thus equals:

$$BCFT_{Tc} + XCr_{Tc}.$$

Case 2. The critical task of Tc is a critical task of one of Tc's parents, and Tc finishes fault-free. Tc could have more than one parent. The WCFT of Tc under this case equals:

$$\max\{BCFT_{Tp} + W_E(Tp, Tc) + C_{Tc}, \forall Tp \in PS_{Tc}\},$$

where $W_E(Tp, Tc)$ is the communication delay from Tp and Tc. Therefore, to determine the worst-case finish time of Tc, we have Theorem 3.

Theorem 3. *The worst-case finish time of* Tc *is either* $BCFT_{Tc} + XCr_{Tc}$, *or* $\max\{WCFT_{Tp} + W_E(Tp,Tc) + C_{Tc}, \forall Tp \in PS_{Tc}\}$, *whichever has the* *larger value. If the former is larger, the critical task of Tc is itself. Otherwise, Tc inherits its critical task from its parent who has the maximum value of $WCFT_{Tp} + W_E(Tp, Tc)$.*

Apparently, the critical task and the WCFT of $Sink$ are the critical task and the WCFT of the task set, respectively. Fig.3 shows the pseudo-code to compute the WCFT and identify the critical task for a task Tc in a scheduled task set. It accepts Tc and X as inputs and outputs the WCFT and one of the critical tasks of Tc. It also uses three arrays, $BCFT(T)$, $C(T)$ and $Cr(T)$, to store the best-case finish time, the worst-case execution time, and the worst-case re-execution time of all the tasks in a task set respectively. $BCFT(T)$ can be calculated from the schedule of the task set assuming zero fault occurrence. The communication delay between any two tasks T and T' is recorded in a two-dimensional array $W_E(T, T')$. The algorithm updates two arrays, $WCFT(T)$ and $Tct(T)$, which are initialized to NULL and will be populated by the WCFT and the critical tasks of all tasks respectively.

```
Initialize WCFT array and Tct array to NULL;
Procedure FindWorstCase(Tc, X)
Input: (Tc, X); Output: (WCFT(Tc), Tct(Tc)) {
1.  if (WCFT(Tc) != NULL)
2.      return (WCFT(Tc), Tct(Tc));
3.  if ($PS_{Tc}$ contains Source) {
4.      WCFT(Tc) = BCFT(Tc) + $X \times Cr(Tc)$;
5.      Tct(Tc) = Tc;
6.      return (WCFT(Tc), Tct(Tc));
7.  }
8.  $WCFT_{Tm}$=0; $W_{ETm}$=0;
9.  do {
10.     Let Tp be a non-visited task in $PS_{Tc}$;
11.     (WCFT(Tp),Tct(Tp)) = FindWorstCase(Tp, X);
12.     if (WCFT(Tp)+$W_E$(Tp, Tc) > $WCFT_{Tm}$+$W_{ETm}$) {
13.         $WCFT_{Tm}$ = WCFT(Tp);
14.         $W_{ETm}$ = $W_E$(Tp, Tc);
15.         $Tct_{Tm}$ = Tct(Tp);
16.     }
17.     Tp is marked as a visited task;
18. } until all tasks in $PS_{Tc}$ are visited.
19. if $WCFT_{Tm}$+$W_{ETm}$+C(Tc) > BCFT(Tc)+$X \times Cr(Tc)$ {
20.     WCFT(Tc)=$WCFT_{Tm}$+$W_{ETm}$+C(Tc);
21.     Tct(Tc) =$Tct_{Tm}$;
22. }
23. else {
24.     WCFT(Tc)=BCFT(Tc)+$X \times Cr(Tc)$;
25.     Tct(Tc) = Tc;
26. }
27. return (WCFT(Tc), Tct(Tc));
28.}
```

Fig.3. Identify the WCFT and a critical task of task Tc.

When the procedure is called the first time with Tc, it begins by checking if the WCFT and Tct of Tc are already available. If so, the procedure returns the recorded values and exits (lines 1 and 2). Otherwise, it continues to determine if PS_{Tc} contains $Source$, which indicates that this is the first task on a processor and does not depend on any other tasks. The WCFT of this task is computed as $BCFT(Tc) + XCr(Tc)$, that is, the task is its own critical task and will experience its WCFT when it incurs all X faults (lines 3∼7).

If PS_{Tc} does not contain $Source$, which indicates that Tc has some parents, the procedure recursively calls itself to compute the WCFT of all tasks in PS_{Tc}. Since a task could be in the parents set of several tasks, it may have been visited before and its WCFT and critical task are available in $WCFT(T)$ and $Tct(T)$. If that is the case, the recursive call simply reads the values and returns (lines 1 and 2). This ensures that the WCFT and Tct of any task will be computed only once. After visiting all tasks in PS_{Tc}, the task with the maximum $WCFT_{Tp} + W_E(Tp, Tc)$ is chosen to be T_p (lines 8∼18).

Next, the algorithm determines which of the two cases results in a worse finish time of Tc. If $WCFT_{\rm Tm} + W_{\rm ETm} + C(Tc) > BCFT(Tc) + XCr(Tc)$, Tc inherits the critical task from $Tct_{\rm Tm}$ (lines 19∼22); otherwise Tc is its own critical task (lines 23∼26).

The WCFT and critical task of a processor can be obtained by calling the procedure with the last task scheduled on the processor as Tc. Similarly, the WCFT and the critical task of the whole task set can be obtained by calling the procedure with $Sink$ as Tc. The algorithm can be viewed as a modified depth-first search algorithm where the starting point is $Sink$. Hence it takes $O(|V| + |E|)$ where $|V|$ and $|E|$ are the number of tasks and dependencies between tasks, respectively.

5 Example Application of Proposed Method

The proposed method identifies the critical task of a scheduled task set and finds its worst-case finish time. This method can be applied to a wide variety of multi-processor systems, where the processors could be homogeneous or heterogeneous, DVS-capable or DVS-incapable. In this section, we use a heterogeneous DVS-capable multi-processor system as the platform to test the proposed method. Given a DAG task set, its schedule information hence includes the task-to-processor mapping, the task-to-task order in each processor, and the task-to-speed assignment of each task. A schedule is optimal if it consumes the least energy if no fault occurs, and guarantees the feasibility even in the worst case of fault occurrences. A simulated annealing (SA) algorithm is proposed. It searches the (near) optimal schedule iteratively. In each iteration, a feasible schedule is found, and is accepted if it consumes less energy than the best schedule so far, or is accepted with a probability if it consumes more. The proposed critical-task method is used in every iteration to test if a newly found schedule is feasible. Hence, the efficiency of the critical-task method contributes significantly to the overall efficiency of the whole algorithm. For a comparison, the common practice 2 is used to replace the critical-task method in the same algorithm to see the effect of its overestimation. We did not implement the common practice 1 as it underestimates the worst-case scenario, thereby resulting in infeasible schedules.

5.1 Review of Energy-Aware Task Scheduling

In this subsection, we present related work of applying DVS technique for energy optimization scheduling problems in real-time embedded systems. Despite of a great deal of related work, only few of them investigate the static dependent tasks scheduling for energy optimization in real-time multiprocessor systems. Luo and Jha took a combined static and dynamic approach in [31] to construct an energy-efficient schedule mainly based on critical path analysis and task execution order refinement. The presented algorithm assumes non-preemptive tasks with a fixed-priority assignment policy on a single processor. Similarly, Liu and Mok designed online and offline algorithms in [32] to reduce the power consumption when executed task cycles are fewer than those expected. The energy efficient scheduling policies have also been investigated in [31-33] for periodic tasks.

Since the task scheduling for energy conservation problem is NP-hard, the probabilistic heuristics including genetic and simulated annealing based algorithms have been proposed in the literature. Kianzad *et al.*[34] proposed a genetic algorithm called CASPER for both homogenous and heterogeneous systems based on the slack distribution algorithm (PDP-SPM)[35] and the power variation DVS algorithm (PV-DVS)[36]. A parallel genetic algorithm was proposed by Lin and Ding[37] to improve search speed, where the population is partitioned into several groups and each group was processed by a genetic algorithm. Huang *et al.*[38] explored a trade-off between the processors and the network links for energy optimization by extending an existing integer linear programming (ILP) formulation. A simulated annealing heuristic with timing adjustment (SA-TA) was then proposed to explore the search space near the accepted mapping for a new feasible mapping and energy minimization.

5.2 System Setup

We are given an M-processors system. The processor set is referred to as $R = \{p_1, p_2, p_3, ..., p_M\}$, where each processor has maximal L different speed levels. Note that the speed levels of different processors could refer to different speeds. A given task graph is modeled by a WDAG $G = (V, E)$, with $V = (u_1, u_2, ..., u_{|V|})$ representing a set of nodes and $E \subseteq V \times V$ representing dependency relations. It is assumed that no more than X faults may occur during the execution frame restricted by a given deadline. The algorithm is to find the optimal schedule — a schedule that finishes the task set on or before the end of execution frame in the case where the task set experiences the worst-case finish time, or consumes the least energy in the case where no fault occurs. The data dependencies between tasks are preserved throughout the algorithm.

5.3 Energy Model

Energy consumption in the given heterogeneous DVS-capable system is estimated using the following parameters:

$$\Pi = (R, L, V, E, ET, \boldsymbol{\Theta}, \Phi, D, X, SA),$$

where

- $R = \{p_1, p_2, p_3, \ldots, p_M\}$ represents the collection of M heterogeneous available resources;
- L represents the number of speed levels supported by each processor. Note that heterogeneous processors may not support the same number of speed levels. Hence some levels, such as L and $L-1$, of a processor could be the same;
- $V = (u_1, u_2, \ldots, u_{|V|})$ represents the collection of $|V|$ interdependent tasks of an application;
- $E \subseteq V \times V$ represents the data dependence relations among $|V|$ tasks;
- ET is a one-dimensional array in which ET_i represents the number of clock cycles needed to perform task u_i;
- $\boldsymbol{\Theta}$ is an $M \times L$ matrix, in which $\Theta_{k,l}$ stands for the clock cycle period corresponding to the l-th speed level of processor p_k. Clock cycle period is the multiplicative inverse of frequency, so $\Theta_{k,l} = \frac{1}{f_{k,l}}$;
- $\Phi = (\boldsymbol{P}^d, \boldsymbol{P}^s)$ represents power consumption which includes both dynamic power consumption P^d

and static power consumption $\boldsymbol{P}^s$: $\boldsymbol{P}^d$ is an $M \times L$ matrix, in which $P^d_{k,l}$ is the dynamic power consumed by performing a task at the l-th speed level of processor p_k; $\boldsymbol{P}^s$ is a one-dimensional array, in which P^s_k is the static power consumed by processor p_k as long as it is powered on;

• D is the given deadline that defines the end of the execution frame;

• X is the maximum number of faults that may occur during the execution frame;

• $SA = (\Omega, t_0, t_1, \delta)$ is the simulated annealing based method in this paper, where Ω is the way of encoding, t_0 is the initial temperature, t_1 is the freezing point which is less than t_0, and δ is cooling rate, where $0 < \delta < 1$.

In a DVS-capable system, energy consumption can be divided into two parts, frequency-dependent and frequency-independent, denoted as E^d and E^s, respectively[37,39-40]. Thus, the overall energy consumption of a task set can be computed by the following equation in which u_i is assumed to be carried at the l-th level of the k-th processor.

$$\begin{aligned} E &= E^d + E^s \\ &= \sum_{i=1}^{|V|} E^d(i) + \sum_{k=1}^{M} P^s_k \times D \\ &= \sum_{i=1}^{|V|} P^d_{k,l} \times ET_i \times \Theta_{k,l} + \sum_{k=1}^{M} P^s_k \times D, \end{aligned}$$

where $E^d(i)$ denotes the energy consumption of task u_i. It is assumed that processors are kept on during the execution frame of the task set.

5.4 Scheduling Using SA

The simulate annealing (SA) method is an adaptation of the Metropolis-Hastings algorithm, a Monte Carlo method to generate sample states of a thermodynamic system invented by Rosenbluth[41]. It was independently described by Kirkpatrick in 1983[42] and by Černý in 1985[41]. Different from the traditional random searching method, the simulated annealing algorithm is a generic probabilistic meta-heuristic for the global optimization problem which is often used when the search space is discrete and large (e.g., traveling salesman problem). In this subsection, we present an SA-based algorithm to search for the optimal schedule.

5.4.1 Schedule Representation

Like a chromosome used in genetic algorithms, the schedule in this work is represented by an ordered list of genes where each gene contains three data items for a single task: the task index, the processor index, and the speed level index[37]. The structure of a schedule is represented by a $3 \times |V|$ array shown in Table 1, where $|V|$ is the total number of tasks in the task set. The structure contains all the information carried by a schedule: the task-to-processor mapping, the task-to-task order in each processor, and the task-to-speed assignment. The first column in Table 1 lists the task ID, the second column and the third column give the processor and the speed level on which each task is carried, respectively. The dependencies can be derived from the array easily. If there are data and/or schedule dependencies from task A to task B, A is always listed on the left of B, though they are not necessarily adjacent.

Table 1. Schedule Representation

Task	Processor	Level
u_1	p_1	L_1
u_2	p_2	L_2
$\vdots$	$\vdots$	$\vdots$
$u_{\lvert V\rvert}$	$p_{\lvert V\rvert}$	$L_{\lvert V\rvert}$

Table 2 shows an example for the task graph shown in Fig.4 and scheduled on a 2-processor system. The following information can be read easily from the table. Tasks u_1 and u_5 are scheduled one after the other on processor p_2, and on L_1 and L_2 speed levels of p_2, respectively. Tasks u_2, u_3 and u_4 are scheduled one after another on processor p_1, and on L_2, L_2, and L_1 speed levels of p_1 respectively. The start time of each task can be derived easily. Each task will be executed on its allocated processor and its assigned speed level as soon as its (data and schedule) dependencies are cleared.

Table 2. Schedule Representation Example

Task	Processor	Level
u_1	p_2	L_1
u_2	p_1	L_2
u_4	p_1	L_2
u_3	p_1	L_1
u_5	p_2	L_2

5.4.2 SA-Based Algorithm

The simulated annealing algorithm that finds the optimal schedule for a given task set and a given processor set is presented in Algorithm 1. The inputs include:

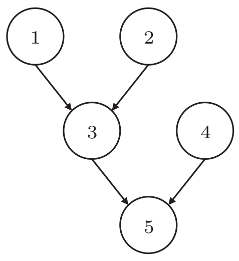


Fig.4. Task graph modeled by a DAG.

• a system with M available heterogeneous processors, and the clock period $\mathbf{\Theta}$, the power profile of each speed level of each processor;

• a task graph modeled by DAG $G = (V, E)$, and each task's worst-case execution time (in terms of cycle numbers) denoted by ET;

• a deadline D that defines the length of the frame, and the maximum number of faults X that may occur during the execution frame.

Algorithm 1. *Find_the_Optimal_Schedule*

```
Require: $R, G = (V, E), ET, \mathbf{\Theta}, \Phi, D, X$;
Ensure: the minimal energy cost $min\_e$ and corresponding
    schedule $min\_s$;
 1: Initialize control parameters $t_0, t_1, \delta$;
 2: Encode and generate a feasible initial schedule $init\_s$;
 3: $min\_s \leftarrow cur\_s \leftarrow init\_s$;
 4: $min\_e \leftarrow cur\_e \leftarrow compute\_energy(cur\_s)$;
 5: $t = t_0$
 6: while $t > t_1$ do
 7:   for $i \leftarrow 1$ to $loop$ do
 8:     $new\_s = generate\_new\_feasible\_schedule(cur\_s)$;
 9:     $new\_e = compute\_energy(new\_s)$;
10:     $diff = new\_e - cur\_e$;
11:     if $diff < 0$ then
12:       $cur\_e = new\_e$;
13:       $cur\_s = new\_s$;
14:     else
15:       $rnd = Random(0, 1)$;
16:       if $\exp(-diff/T) > rnd$ then
17:         $cur\_e = new\_e$;
18:         $cur\_s = new\_s$;
19:       end if
20:     end if
21:     if $cur\_e < min\_e$ then
22:       $min\_e = cur\_e$;
23:       $min\_s = cur\_s$;
24:     end if
25:   end for
26:   $t = t \times \delta$
27: end while
28: print($min\_s, min\_e$);
```

After initializing the control parameters such as the initial and freeze temperatures t_0 and t_1 and the cooling rate δ, the algorithm starts from a randomly generated initial schedule (step 2), and tries to improve it iteratively (steps 6∼27). Each iteration mainly consists of three steps: generating a new feasible schedule based on the current schedule (step 8), computing its energy cost (step 9), and accepting it if its energy cost is lower, or accepting it with a probability otherwise (steps 10∼26). At the end of each iteration (step 26), the current temperature t is updated. The *loop* variable used in step 7 is an integer to restrict the number of inner loops. The proposed method that identifies the critical task of a scheduled task set and finds its worst-case finish time is employed in the function of *generate_a_new_feasible_schedule* (step 8).

5.4.3 *Generate New Feasible Schedules*

This subsection introduces our way to generate a new feasible schedule based on the current schedule. The difficult part is that the randomness used in this process must be nicely controlled so that the generated schedule is legal, i.e., preserving the data dependencies of the original DAG. Please also note that a legal schedule is an intermediate step towards a complete new schedule where the task-to-speed assignment is added. The feasibility test of a complete schedule needs the proposed critical-task method. Step 8 in Algorithm 1 has two steps:

• Step a finds a new legal schedule. A legal schedule must preserve the data dependencies of the input DAG, and might have a different task-to-processor assignment and task-to-task order from the current schedule.

• Step b finds a proper task-to-speed assignment to complete the schedule. This is an iterative process and each iteration randomly finds a speed assignment, and then tests the feasibility of the complete schedule using the method proposed in this paper. The iterative process stops when a feasible schedule is found. If not, return to step a for a new legal schedule.

Algorithm 2 presents our approach that can generate a new legal schedule from the current schedule effectively. It takes the DAG and the current schedule represented in Table 2 as inputs, and outputs a new legal schedule. Step 1 is to randomly change the mapping of the task based on the current schedule. This is done by either changing the mapping of a single task, or swapping the mapping of two tasks. In step 2, it picks up a random task and records its position in variable *pos*. In steps 3∼15, it then finds the nearest predecessor and successor of the task located at *pos* and records their positions into variables *low* and *high*, respectively.

It can be observed that any position in the range of $[low + 1, high - 1]$ is a legal position where the task at *pos* can be moved. It then picks a random position from the range and moves the picked task there in steps 17∼22. $Extract\_and\_Insert(arr, pos, tmp)$ moves the picked task at *pos* to the randomly generated position *tmp* in *arr*. Algorithm 2 can guarantee the legality of its output, and there is a probability that all legal schedules can be reached.

Algorithm 2. *Generate_New_Legal_Schedule*

```
Require: the current schedule represented by $arr[3][1..|V|]$, $G = (V, E)$;
Ensure: a new legal schedule;
 1: $Generate\_New\_Mapping(arr[3][1..|V|])$;
 2: $pos = Random(1, |V|)$;
 3: $low \leftarrow 0$;
 4: $high \leftarrow |V| + 1$;
 5: for each edge $e(i, arr[pos]) \in E$ do
 6:     if $Find\_Position(arr, i) > low$ then
 7:         $low = Find\_Position(arr, i)$;
 8:     end if
 9: end for
10: for each edge $e(arr[pos], j) \in E$ do
11:     if $Find\_Position(arr, j) < high$ then
12:         $high = Find\_Position(arr, i)$;
13:     end if
14: end for
15: if $low + 1 == high - 1$ then
16:     Go to step 1;
17: else
18:     $tmp = Random(low + 1, high - 1)$;
19:     while $tmp == pos$ do
20:         $tmp = Random(low + 1, high - 1)$;
21:     end while
22:     $Extract\_and\_Insert(arr, pos, tmp)$;
23:     return $arr[3][1..|V|]$;
24: end if
```

6 Experiments

Before the critical-task theory proposed in this work, there are two common practices as mentioned in Subsection 2.5.

1) *Common Practice* 1. Assume the longest task incurs all the expected faults. The worst-case finish time could be underestimated.

2) *Common Practice* 2. Reserve the slack for recovery by simply multiplying the re-execution time of the longest task with the expected fault number. The worst-case finish time could be overestimated.

We hence design two experiments to show the significance of the proposed critical-task theory with one experiment for each common practice. The experiments are done on well-known DSP benchmarks from DSPstone[43]. The benchmarks are *motiv* with five tasks, 2-*motiv* and *deq* with 11 tasks, 2*iir* and *floyd* with 16 tasks, 2-*deq* with 22 tasks, and 4-*lat-iir* with 26 tasks.

6.1 Underestimation of Common Practice 1

We have developed a simulator for this purpose. The simulator uses list-scheduling to get a scheduled task set for the benchmarks. It then compares the WCFT calculated using the proposed critical-task method, as well as the WCFT estimated using common practice 1. The simulation results of WCFT analysis are shown in Fig.5. The difference ratio reflects the underestimation of common practice 1. The surface graph contains the results of five benchmarks with the number of processors varying from 2 to 8 and the number of faults varying from 1 to 4, respectively. During the experiment, two points can be observed as follows. 1) The critical task may not be the longest task. This leads to different worst-case finish time. That is to say, in Fig.5, the higher the curve is, the more significant the difference is. And the most significant difference ratio can be close to 23% in some cases. 2) For the same benchmark, the critical task may change if the number of processors and/or faults changes, while the longest task remains the same.

6.2 Overestimation of Common Practice 2

The second experiment is to show the overestimation of common practice 2. Common practice 2 overestimates the worst case of a task set, which not only results in wrong rejections of feasible schedules, but also wastes the energy of the underline systems. Hence we divide the experiment into two parts. In the first part, we check the WCFT overestimated by common practice 2, a similar experiment as the last subsection. The experimental results are shown in Fig.6. From Fig.6, it can be observed that common practice 2 can overestimate the WCFT up to 13% compared with the proposed critical-task method.

The second part focuses on energy conservation in which the SA-based algorithm adopts either the proposed critical-task method or common practice 2. We simulate 2-, 4-, 6-, 8-processor heterogeneous systems with each processor having six speed levels, and the maximum number of faults varies from 1 to 3. The frequencies and operating voltage points configuration refers to the Intel® Pentium® M Processor datasheet①. Table 3 shows the Intel® Pentium® M Processor at 1.6 GHz which supports six frequencies and voltage operating points. As it can be seen from the table, the clock frequency can be stepped down in

①Enhanced Intel® SpeedStep® Technology for the Intel® Pentium® M Processor. http://download.intel.com/design/network/papers/30117401.pdf, Nov. 2015.

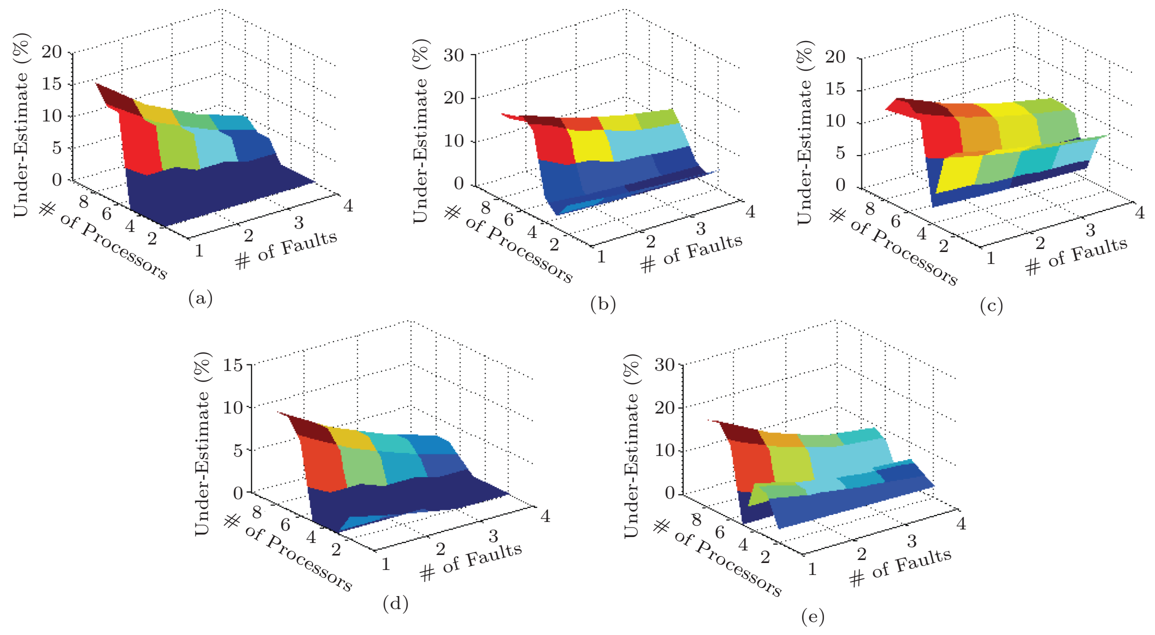


Fig.5. Common practice 1 underestimates WCFT. (a) 2-*motiv*. (b) *deq*. (c) 2*iir*. (d) 2-*deq*. (e) 4-*lat*-*iir*.

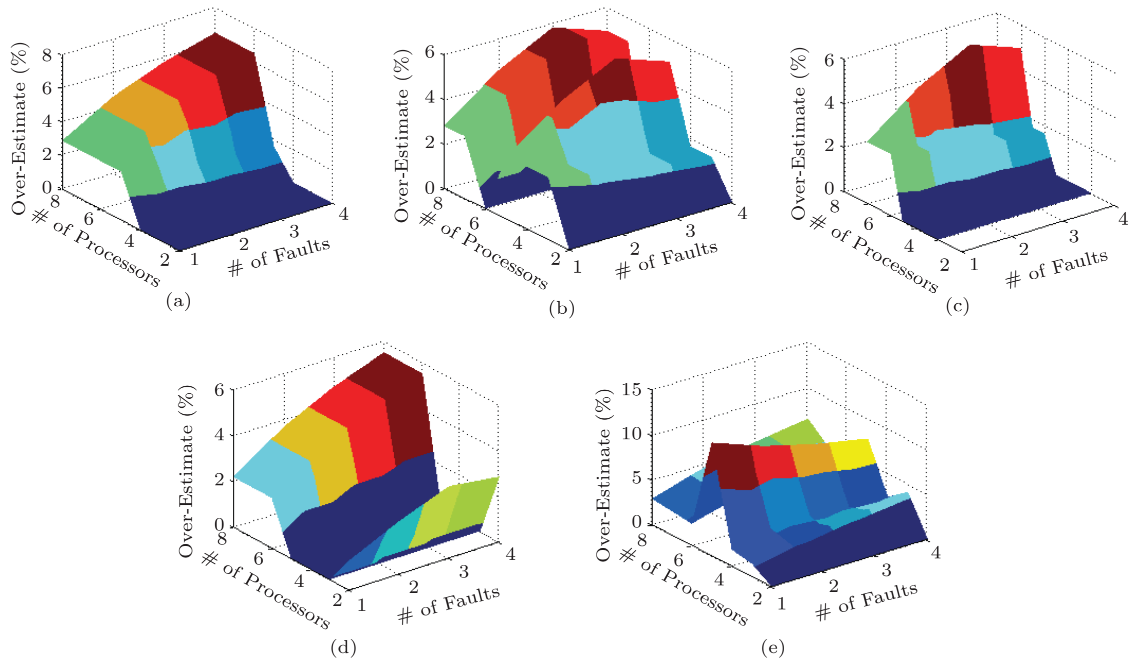


Fig.6. Common practice 2 overestimates WCFT. (a) 2-*motiv*. (b) *deq*. (c) 2*iir*. (d) 2-*deq*. (e) 4-*lat*-*iir*.

200 MHz decrements over the range from 1.6 GHz to 0.6 GHz. At the same time, the voltage requirement decreases from 1.484 V to 0.956 V. According to the dynamic power consumption equation $P \propto CV^2F$, the power consumption goes down by a factor of 6.4. Based on this premise, other heterogeneous cores' configurations are derived.

Table 3. Intel® Pentium® M Processor at 1.6 GHz

Frequency (GHz)	Voltage (V)
1.6	1.484
1.4	1.420
1.2	1.276
1.0	1.164
0.8	1.036
0.6	0.956

The experimental results of energy consumption are shown in Fig.7. In the experiment, common practice 2 and critical-task method are used under the calculation of energy consumption of benchmarks by the SA respectively. The higher energy consumption of the two methods is set to be the standard which is 100%, while the lower energy consumption is set according to the proportion. From Fig.7, we can see that in most cases, the energy consumption of schedules found by the SA using the proposed critical-task method is less than that of using common practice 2. This is because the overestimation of WCFT results in less slack for energy conservation. It is also observed that the energy consumption by the SA using the critical-task method is 60% of that of using common practice 2 at some point, which means that the energy conservation gained by using the proposed critical-task method can be up to (approximately) 40%. This gives the evidence that the precise estimation of WCFT of a schedule can contribute a lot to energy saving compared with common practice 2. We also note that when the number of tasks is larger, the results obtained from common practice 2 are better than those by the critical-task method in very rare cases. This is because that the proposed SA algorithm is after all a heuristic process, and does not guarantee the optimal results all the time.

7 Conclusions

In this paper, we investigated the impact of the fault occurrences on the worst-case finish time of a task set scheduled in a frame-based multi-processor system. It is assumed that tasks could have an inter-task dependency, and the number of fault occurrences in a frame is upper-limited by X. We concluded that there exists at least one critical task for each task. A task undergoes its worst-case finish time when one of its critical tasks incurs all X transient faults assuming one re-execution is dedicated to one fault. Based on the critical-task theory, a recursive algorithm is then designed to identify the critical task and the worst-case finish time of a task set in $O(|V| + |E|)$ where $|V|$ and $|E|$ are the number of tasks and dependencies between tasks, respectively, while the state-of-the-art one takes $O\left(\frac{(|V|+X-1)!}{X!(|V|-1)!}|V|^2\right)$.

Then we conducted experiments on benchmarks, and compared the results with the common practices to show the significance of the proposed critical-task method. Experimental results showed that the common practices either underestimate the worst case by 23%, or overestimate it by 13%. We also presented an example application where the proposed critical-task method is used to find the energy efficient fault-tolerant schedule. Experimental results showed that with the same time complexity as the practice, the proposed method could help save close to 40% energy.

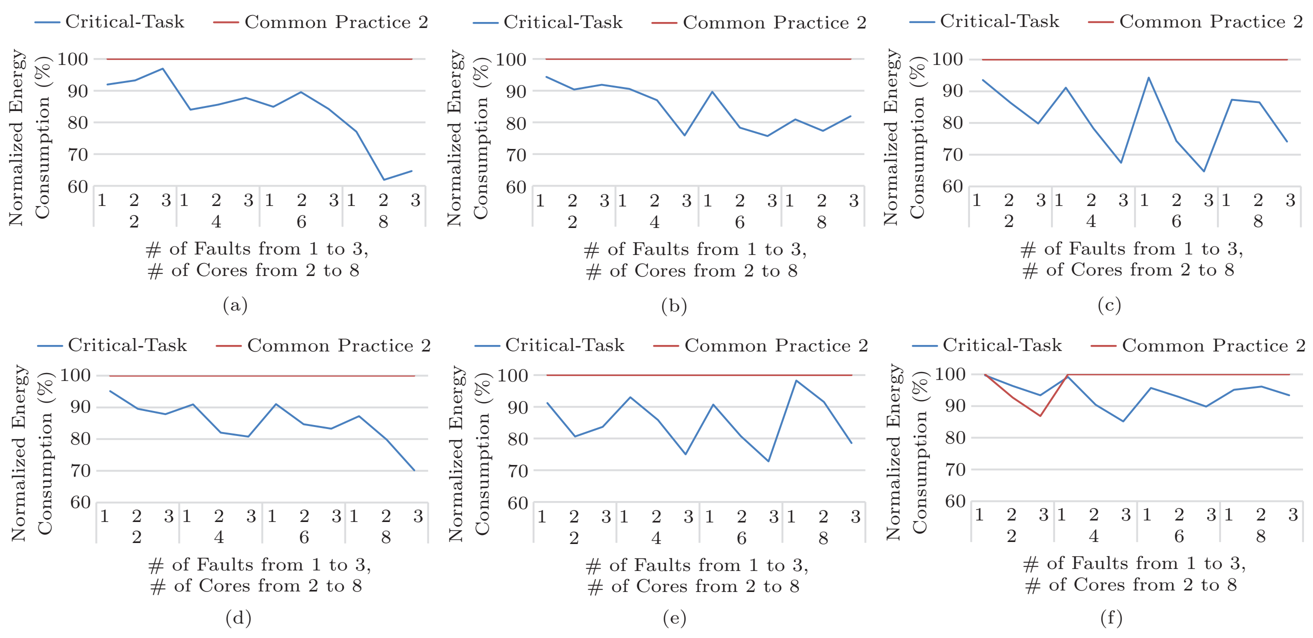


Fig.7. Comparison of energy consumption by the SA-based algorithm using common practice 2 and the proposed critical-task method. (a) *motiv*. (b) 2-*motiv*. (c) *deq*. (d) *floyd*. (e) 2*iir*. (f) 4-*lat-iir*.

References

[1] Wei T, Mishra P, Wu K, Zhou J. Quasi-static fault-tolerant scheduling schemes for energy-efficient hard real-time systems. *J. Systems and Software*, 2012, 85(6): 1386-1399.

[2] Liu C L, Layland J W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM* (*JACM*), 1973, 20(1): 46-61.

[3] Kopetz H, Grunsteidl G. TTP — A protocol for fault-tolerant real-time systems. *Computer*, 1994, 27(1): 14-23.

[4] Chevochot P, Puaut I. Scheduling fault-tolerant distributed hard real-time tasks independently of the replication strategies. In *Proc. the 6th Int. Conf. Real-Time Computing Systems and Applications,* Dec. 1999, pp.356-363.

[5] Dima C, Girault A, Lavarenne C, Sorel Y. Off-line real-time fault-tolerant scheduling. In *Proc. the 9th Euromicro Workshop on Parallel and Distributed Processing*, Feb. 2001, pp.410-417.

[6] Girault A, Kalla H, Sighireanu M, Sorel Y. An algorithm for automatically obtaining distributed and fault-tolerant static schedules. In *Proc. International Conference on Dependable Systems and Networks*, Jun. 2003, pp.159-168.

[7] Pop P, Izosimov V, Eles P, Peng Z. Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication. *IEEE Trans. Very Large Scale Integration Systems*, 2009, 17(3): 389-402.

[8] Kandasamy N, Hayes J P, Murray B T. Transparent recovery from intermittent faults in time-triggered distributed systems. *IEEE Trans. Computers*, 2003, 52(2): 113-125.

[9] Olteanu A, Pop F, Dobre C, Cristea V. A dynamic rescheduling algorithm for resource management in large scale dependable distributed systems. *Computers & Mathematics with Applications*, 2012, 63(9): 1409-1423.

[10] Pop F, Dobre C, Cristea V. Performance analysis of grid DAG scheduling algorithms using MONARC simulation tool. In *Proc. the 7th ISPDC*, Jul. 2008, pp.131-138.

[11] Pop F, Cristea V. Intelligent strategies for DAG scheduling optimization in grid environments. arXiv Preprint, arXiv: 1106.5303, 2011. http://arxiv.org/ftp/arxiv/papers/1106/1106.5303.pdf, August 2015.

[12] Ghosh S, Melhem R, Mosse D. Enhancing real-time schedules to tolerate transient faults. In *Proc. the 16th IEEE Real-Time Systems Symposium*, Dec. 1995, pp.120-129.

[13] Burns A, Davis R, Punnekkat S. Feasibility analysis of fault-tolerant real-time task sets. In *Proc. the 8th Euromicro Workshop on Real-Time Systems* , Jun. 1996, pp.29-33.

[14] Audsley N, Burns A, Richardson M *et al.* Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 1993, 8(5): 284-292.

[15] Liberato F, Melhem R, Mossé D. Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems. *IEEE Transactions on Computers*, 2000, 49(9): 906-914.

[16] Aydin H. Exact fault-sensitive feasibility analysis of real-time tasks. *IEEE Trans. Computers*, 2007, 56(10): 1372-1386.

[17] Chrobak M, Hurand M, Sgall J. Fast algorithms for testing fault-tolerance of sequenced jobs with deadlines. In *Proc. the 28th IEEE RTSS*, Dec. 2007, pp.139-148.

[18] Thekkilakattil A, Dobrin R, Punnekkat S *et al.* Resource augmentation for fault-tolerance feasibility of real-time tasks under error bursts. In *Proc. the 20th Int. Conf. Real-Time and Network Systems*, Nov. 2012, pp.41-50.

[19] Goddard S. On the management of latency in the synthesis of real-time signal processing systems from processing graphs [Ph.D. Thesis]. The University of North Carolina at Chapel Hill, 1998.

[20] Liu C, Anderson J H. Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss. In *Proc. the 31st IEEE RTSS*, Nov. 30-Dec. 3, 2010, pp.3-13.

[21] Bauer G, Kopetz H. Transparent redundancy in the time-triggered architecture. In *Proc. International Conference on Dependable Systems and Networks*, Jun. 2000, pp.5-13.

[22] Bretz E A. By-wire cars turn the corner. *IEEE Spectrum*, 2001, 38(4): 68-73.

[23] Kopetz H. Why time-triggered architectures will succeed in large hard real-time systems. In *Proc. the 5th IEEE FT-DCS*, Aug. 1995, pp.2-9.

[24] Obermaisser R. Event-Triggered and Time-Triggered Control Paradigms. Springer US, 2004.

[25] Poledna S. Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism. Springer US, 1996.

[26] Suri N, Walter C J, Hugue M M. Advances in ULTRA-Dependable Distributed Systems. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994.

[27] Pop P, Eles P, Peng Z. Schedulability analysis for systems with data and control dependencies. In *Proc. the 12th Euromicro Conf. Real-Time Systems,* June 2000, pp.201-208.

[28] Laplante P A. Real-Time Systems Design and Analysis. John Wiley & Sons, 2004.

[29] Liu Y, Liang H, Wu K. Scheduling for energy efficiency and fault tolerance in hard real-time systems. In *Proc. the DATE*, Mar. 2010, pp.1444-1449.

[30] Manber U. Introduction to Algorithms: A Creative Approach. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[31] Luo J, Jha N K. Static and dynamic variable voltage scheduling algorithms for real-time heterogeneous distributed embedded systems. In *Proc. the 15th International Conference on VLSI Design*, Jan. 2002, pp.719-726.

[32] Liu Y, Mok A K. An integrated approach for applying dynamic voltage scaling to hard real-time systems. In *Proc. the 9th IEEE RTAS*, May 2003, pp.116-123.

[33] Cho Y, Chang N, Chakrabarti C, Vrudhula S. High-level power management of embedded systems with application-specific energy cost functions. In *Proc. the 43rd Annual Design Automation Conference*, July 2006, pp.568-573.

[34] Kianzad V, Bhattacharyya S S, Qu G. CASPER: An integrated energy-driven approach for task graph scheduling on distributed embedded systems. In *Proc. the 16th IEEE ASAP,* Jul. 2005, pp.191-197.

[35] Hua S, Qu G. Power minimization techniques on distributed real-time systems by global and local slack management. In *Proc. the 10th ASP-DAC*, Jan. 2005, pp.830-835.

[36] Schmitz M T, Al-Hashimi B M, Eles P. Iterative schedule optimization for voltage scalable distributed embedded systems. *ACM Trans. Embedded Computing Systems*, 2004, 3(1): 182-217.

[37] Lin M, Ding C. Parallel genetic algorithms for DVS scheduling of distributed embedded systems. In *Proc. the 3rd HPCC*, Sept. 2007, pp.180-191.

[38] Huang J, Buckl C, Raabe A, Knoll A. Energy-aware task allocation for network-on-chip based heterogeneous multiprocessor systems. In *Proc. the 19th PDP*, Feb. 2011, pp.447-454.

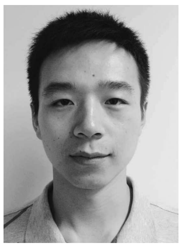
[39] Hung C M, Chen J J, Kuo T W. Energy-efficient real-time task scheduling for a DVS system with a non-DVS processing element. In *Proc. the 27th IEEE International Real-Time Systems Symposium*, Dec. 2006, pp.303-312.

[40] Xu R, Melhem R, Mosse D. Energy-aware scheduling for streaming applications on chip multiprocessors. In *Proc. the 28th IEEE Int. Real-Time Systems Symp.*, Dec. 2007, pp.25-38.

[41] Černý V. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 1985, 45(1): 41-51.

[42] Kirkpatrick S. Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics*, 1984, 34(5/6): 975-986.

[43] Živojnović V, Velarde J M, Schläger C, Meyr H. DSPstone: A DSP-oriented benchmarking methodology. In *Proc. the ICSPAT*, Oct. 1994, pp.715-720.



Xiao-Tong Cui received his B.S. degree in computer science and technology from Chongqing University, Chongqing, in 2013. Currently he is a Ph.D. candidate majoring in computer science and technology of the College of Computer Science, Chongqing University. His current research interests include real-time task scheduling and hardware security.

Kai-Jie Wu received his B.E. degree in circuits and systems from Xidian University, Xi'an, in 1996, his M.S. degree in circuits and systems from the University of Science and Technology of China, Hefei, in 1999, and his Ph.D. degree in electrical engineering from Polytechnic University (now Polytechnic Institute of New York University), Brooklyn, in 2004. He was with the University of Illinois, as an assistant professor. In 2013, he joined as a professor at the College of Computer Science, Chongqing University. His current research interests include computer aided design of radiation hardened VLSI system, countermeasures for side-channel cryptanalysis for crypto devices, and robust and fault-tolerant nanotechnology designs. Dr. Wu is the recipient of the 2004 EDAA Outstanding Dissertation Award for "New Directions in Circuit and System Test".

Tong-Quan Wei received his Ph.D. degree in electrical engineering from Michigan Technological University, Minnesota, in 2009. He is currently an associate professor in the Department of Computer Science and Technology at the East China Normal University, Shanghai. His research interests are in the areas of real-time systems, green and reliable computing, and parallel and distributed systems. He has served as a regional editor for Journal of Circuits, Systems, and Computers (World Scientific) since 2012. He also served as the guest editor of the IEEE Transactions on Industrial Informatics Special Section on Building Automation, Smart Homes, and Communities, and the ACM Transactions on Embedded Computing Systems Special Issue on Embedded Systems for Energy-Efficient, Reliable, and Secure Smart Homes. He is a member of CCF and IEEE.

Edwin Hsing-Mean Sha received his Ph.D. degree in computer science from the Department of Computer Science, Princeton University, USA, in 1992. From August 1992 to August 2000, he was with the Department of Computer Science and Engineering at University of Notre Dame, USA. Since 2000, he has been a tenured full professor at the University of Texas at Dallas. Since 2012, he has served as the dean of the College of Computer Science at Chongqing University, Chongqing. He has published more than 300 research papers in refereed conferences and journals. His work has been cited over 2 200 times. He received Teaching Award, Microsoft Trustworthy Computing Curriculum Award, NSF CAREER Award, NSFC Overseas Distinguished Young Scholar Award, Chang Jiang Scholar Honorary Chair Professorship, and China Thousand Talents Program.