Journal of Circuits, Systems, and Computers

ADAPTIVE FAULT-TOLERANCE TASK SCHEDULING FOR REAL-TIME ENERGY HARVESTING SYSTEMS *

LINJIE ZHU, TONGQUAN WEI†

Department of Computer Science and Technology, East China Normal University, Shanghai 200241, China
†*tqwei@cs.ecnu.edu.cn*

XIAODAO CHEN, YONGHE GUO, SHIYAN HU

Department of Electrical and Computer Engineering, Michigan Technological University, Houghton, MI 49931, USA

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Fault-tolerance and energy have become important design issues in multiprocessor system-on-chips (SoCs) with the technology scaling and the proliferation of battery-powered multiprocessor SoCs. This paper proposed an energy efficient fault tolerance task allocation scheme for multiprocessor SoCs in real-time energy harvesting systems. The proposed fault-tolerance scheme is based on the principle of the primiary/backup task scheduling, and can tolerate at most one single transient fault. Extensive simulated experiment shows that the proposed scheme can save up to 30% energy consumption and reduce the miss ratio to about 8% in the presence of faults.

Keywords: Allocation and scheduling; energy harvesting; fault tolerance; multiprocessor SoCs.

1. Introduction

The number of transient faults in hardware has been rising continuously due to the increasing complexity of design and aggressive technology scaling. Meanwhile, a growing number of complex safety critical applications operate under extreme conditions; thus, demand ultra-reliability and high performance. As a result, fault-tolerance has become an important design constraint for an embedded core based multiprocessor system-on-a-chip (SoC).

Since multiprocessor systems are more amenable to fault-tolerance techniques due to their inherent redundancy, several techniques have been developed with varying levels of granularity: 1) Triple Modular Redundancy (TMR), 2) Primary

*The preliminary version of this manuscript appeared in International Conference on Intelligent Control and Information Processing, 2010.

Backup (PB), and 3) checkpointing scheme. Of these, the PB approach is adopted in this paper to tolerate transient faults. In the PB approach, two copies of a task are executed serially on two different processors and an acceptance test is performed to check the result. The backup copy is executed only if the output of the primary version fails the acceptance test.

Due to the fast-evolving application of real-time systems in battery-powered portable devices, energy has emerged as an important design constraint. Recent research on battery-powered embedded system design has focused on either reducing system energy consumption using dynamic power management technique or increasing system energy storage by harvesting energy from ambient environment.

Dynamic power management has been an active area of research for decades and several techniques, such as Dynamic Voltage Scaling (DVS), have been proposed to reduce processor energy consumption. For system with multiple processing elements, such as multiprocessor SoCs, the system energy consumption depends significantly on the task-to-processor allocation strategies [1]. Recently, energy harvesting technologies have been actively explored [2,3,4]. This is due to the fact that many real-time applications are being deployed in extreme environmental conditions where replacing batteries of devices in the field is not practical. Energy harvesting is a process of deriving energy from external sources, such as ambient vibration, heat, or light, and is particularly significant for autonomous low-power embedded systems.

Energy efficient task allocation and scheduling for multiprocessor systems has been extensively explored. Gururaj et al. [5] described a mathematical programming formulation-based technique for scheduling tasks in DVS-capable multiprocessor systems. The proposed task scheduling scheme takes into account precedence constraints and variation in task execution times to produce an energy efficient task schedule in polynomial time. An energy-aware scheduling scheme for real-time multiprocessor systems with uncertain task execution time was proposed in [6]. The authors considered the probabilistic distribution in task execution time and balanced the application workload among processors using the worst-fit decreasing bin-packing heuristic for energy savings. Watanabe et al. [7] presented a pipelined task scheduling method for minimizing the energy consumption of Globally Asynchronous Locally Synchronous (GALS) multiprocessor SoCs under latency and throughput constraints. In [8], an energy-aware scheduling scheme for periodic real-time tasks in the DVS-capable multiprocessor systems was proposed by considering practical constraints such as discrete speed, idle power, and application-specific power characteristics. However, all the above work focuses on reduction of energy consumption of battery-powered devices assuming a constant energy budget without considering fault-tolerance.

In this paper, an energy efficient fault-tolerance task-to-processor assignment scheme for multiprocessor SoC energy harvesting systems is proposed by extending the multiprocessor energy harvesting scheme presented in [9]. The proposed offline

task allocation scheme greedily assigns real-time tasks to individual processing elements to generate an energy efficient task schedule in the presence of transient fault. It is assumed that fault tolerance is achieved by re-executing the faulty task and at most one single transient fault can be tolerated in the entire system.

The rest of the paper is organized as follows. Section II introduces system architecture. Section III describes the proposed task allocation scheme for multiprocessor SoCs in energy harvesting systems. Section IV presents the experimental results, and Section V concludes the paper.

2. System Architecture

The target fault-tolerance energy harvesting system consists of multiple processing elements on a single chip, energy source module, and energy storage module.

2.1. *Energy Source and Storage*

The system includes energy harvesting and storage modules. Let $P_h(t)$ denote the power generated by the energy source module at time instant t, and $E_h(t_1, t_2)$ denote the harvested energy during time interval $[t_1, t_2]$, the harvested energy fed into energy storage module can be calculated as

$$E_h(t_1, t_2) = \int_{t_1}^{t_2} P_h(t)dt.$$

It is assumed that the power output $P_h(t)$ of the energy source module is a function of time instant t. The power output can be predicted by tracing the energy source profile [10]. It is also assumed that the harvested energy is fed into energy storage without wastage and the energy demand of the multiprocessor SoC only comes from the energy storage. Let E_C denote the energy storage capacity, $E_c(t)$ denote the stored energy at time instant t, and $E_d(t_1, t_2)$ denote the processor energy dissipation, then

$$0 \leq E_c(t) \leq E_C \ \forall t$$

and

$$E_d(t_1, t_2) \leq E_c(t_1) + E_h(t_1, t_2) \ \forall t$$

hold. The later inequality indicates that a processing element has to stop the execution of a task when the available energy is not enough to finish the task execution. The stored energy at a time instant is the current available energy minus the processor energy dissipation, that is

$$E_c(t_2) \leq E_c(t_1) + E_h(t_1, t_2) - E_d(t_1, t_2) \ \forall t_1 < t_2$$

2.2. *Processor Model*

Before the processor model is described, two terminologies, **busy time** and **idle time**, are described. If a processing element executes tasks at a certain period of time, the period of time is the busy time of the processing element. During the period of time, the processing element is also called busy or being used. If a processing element is idle or does not execute tasks in a certain period of time, the period of time is called the idle time of the processing element.

It is assumed that there are R identical processing elements in a multiprocessor SoC, which are denoted by $P = \{p_1, p_2, ..., p_R\}$. A processing element is defined as a quadruple $p = (pid, f, v, btimelist)$, where pid identifies the processing element, f is the fixed clock frequency, v is the supply voltage of the processing element, and $btimelist$ is the busy time chain of the processing element.

It is assumed that the energy consumption of the processor is dominated by dynamic switching energy, which is given by [11]

$$P(f) = C_{eff} \times V_{DD}^2 \times f,$$

where C_{eff} is the effective capacitance, f is the operating frequency, and V_{DD} is the supply voltage. It is also assumed that each processing element supports only one operating frequency and the processing element with smaller index has lower operating frequency. Let f_r denote the operating frequency of the r^{th} processing element for $1 \leq r \leq R$, then $f_{min} = f_1 \leq f_2 \leq ... \leq f_R = f_{max}$ holds. It is assumed that processing elements of the SoC are tightly coupled and share L2 cache such that the communication cost is small enough to be incorporated in task execution time

2.3. *Task Model*

Real-time tasks with precedence constraints are considered in the proposed task allocation scheme. For the sake of easy presentation, this paper only deals with precedence constraints of tasks in the form of a chain. The proposed task allocation scheme can be extended to handle tasks with various forms of precedence constraints.

It is assumed that there are N task chains in a given task set, which is denoted by $\Gamma = \{\Gamma_1, \Gamma_2, ..., \Gamma_N\}$, and there are M_i periodic tasks in each chain, which is denoted by $\Gamma_i = \{\tau_1, \tau_2, ..., \tau_{Mi}, \prec\}$. In each task chain, a partial-order relation $\prec$, called a precedence relation, is utilized to specify the precedence constraints among tasks. The notation $\tau_i \prec \tau_j \prec \tau_k$, where $i = j - 1$ and $k = j + 1$, indicates that τ_i is an immediate predecessor of task τ_j, which is in turn a predecessor of task τ_k. On the contrary, τ_k is an immediate successor of task τ_j, which is in turn a successor of task τ_i. In other words, task τ_j is ready for execution when task τ_i is completed, and task τ_k is ready for execution when both task τ_i and task τ_j are completed. The timing characteristics of the task in a chain are defined as $\tau_m = (T_m, D_m, wcet_m, pe, st_m, ft_m, pre, next)$, where T_m is the period, D_m is the

deadline, $wcet_m$ is the worst case execution cycles, and pe denotes the processing elements on which the task is to be executed. The task is also characterized by the start time st_m, the expected finish time ft_m, the immediate predecessor pre of the task, and the immediate successor $next$ of the task in the chain. It is assumed that the period of a task equals its deadline and all tasks in a task set share a common deadline, that is $T_m = D_m = T = D$. It is also assumed that each task is ready to execute at the very beginning and can be assigned to one and only one processing element. The execution of a task is assumed to be non-preemptive.

2.4. *Fault Model*

Although all digital systems are susceptible to single event upset (SEU)-induced transient faults, the rate of transient fault arrival remains low in the foreseeable future and fault-free condition continues to dominate [12,13,14]. Based on this observation, the primary/backup technique that is best suited for low failure rate and for applications of large laxity [15], is adopted as the fault-tolerance technique in this paper. It is assumed that at most a singe fault is to be tolerated in the target system. Since real-time tasks allocated to multiple processing elements share a common deadline and the system is designed to tolerate at most one fault, the budget for only one backup copy is reserved on each processing element. The reserved slack is equal to the execution time of the longest task on the processing element.

3. Energy Efficient Fault-Tolerance Task Allocation Algorithm

This section describes fault tolerant strategy firstly. Then it describes the proposed task-to-processing element assignment scheme for multiprocessor SoCs in real-time embedded energy harvesting systems. It is assumed that tasks have precedence constraints and all tasks are ready at time instant 0. The input to the proposed algorithm is a set of tasks sharing a common deadline D and the output of the algorithm is an energy efficient fault-tolerance task schedule. The proposed scheme can tolerate a single fault of any processor at any time.

3.1. *Fault-Tolerance Strategy*

The proposed fault-tolerance task allocation scheme reserves enough slack for fault recovery and re-executes the faulty task when a single fault occurs. For a task set Γ, the reserved slack cycle is the worst case execution cycle of the longest task in the task set. In other words, the reserved slack for fault recovery is

$$wect_{max} = \max_{i=1}^{M} wcet_i,$$

where M is the number of tasks in the task set. Since a fault could occur on any processing element, this amount of slack is reserved on each processing element. The slack reserved on a processing element also can be written as

$$wcet_{max}/f_{min},$$

where $wcet_{max}$ is the longest worst case execution cycle of tasks in the task set and f_{min} is the lowest operating frequency of processing elements.

Assume that a fault occur during the execution of task i on processing element m, and the system recover from the fault by re-executing the faulty task, then the required slack is given by

$$wect_i/f_m.$$

As is discussed above, the reserved slack is given by

$$wcet_{max}/f_{min}.$$

Since $wect_i < wcet_{max}$ and $f_{min} < f_m$, the reserved slack is long enough for task i to recover from the fault.

3.2. *Static Task-to-Processor Assignment Algorithm*

In the proposed task allocation scheme, tasks in a given task set are assigned to individual processing elements in a way that timing constraints and precedence constraints of all tasks are satisfied, meanwhile, energy consumption of a task set is greedily minimized.

It has been shown that, for a given task, the processing element operating at the single frequency f_{low} consumes less energy than the processing element operating at the single frequency f_{high}, where $f_{low} < f_{high}$ [16]. Therefore, a task schedule is generated by assigning tasks in the given task set to the processing element with lower operating frequency such that the energy consumption of the output task schedule is greedily minimized. The proposed task allocation scheme selects a task chain and assigns tasks in the task chain to processing elements such that the precedence constraints of tasks in the task chain are preserved. Since precedence graph of tasks is assumed to be independent chains, the precedence constraint of a task can be satisfied by executing the task after the finish time of its immediate predecessor and before the deadline. For a task without its immediate predecessor, the task can start at time 0. For a task τ_m in a chain assigned to the processing element PE_r,

$$st_m \geq ft_{m-1}, ft_m \leq D, 1 < m < M \tag{1}$$

$$t_m = \frac{wcet_m}{f_r} \tag{2}$$

where t_m is the worst case execution time of the task τ_m, f_r is the fixed frequency of the processing element PE_r and $wcet_m$ is the worst case execution cycles. Combining the inequality (1) and equation (2) gives the below condition for task τ_m to be feasibly scheduled on processing element PE_r without compromising the precedence constraint of the task. That is,

$$st_m \geq ft_{m-1}, st_m + \frac{wcet_m}{f_r} \leq D. \tag{3}$$

Procedure($Task\ set\ \Gamma$)

```
1.  feasible = 1
2.  for Γ_i, 1 ≤ i ≤ N do
3.    for PE_r, 1 ≤ r ≤ R do
4.      for τ_m ∈ chain Γ_i do
5.        for (j == r; j <= R; j++) do
6.          if inequality (3) holds then
7.            update PE_j, st_m, ft_m acc. to inequality(1)
8.          end if
9.        end for
10.       if j == R + 1 then
11.         for τ_pre of task τ_m ∈ chain Γ_i do
12.           update PE, st_pre, ft_pre acc. to inequality(1)
13.         end for
14.       end if
15.     end for
16.   end for
17.   if n == R + 1 then
18.     feasible = 0; print "unfeasible schedule";
        return feasible
19.   end if
20. end for
21. return feasible
```

Fig. 1. Generate a static task schedule.

Since the operating frequency of each processing element is constant, the worst execution time t_m of task τ_m satisfies condition given as

$$\frac{wcet_m}{f_{max}} \leq t_m \leq \frac{wcet_m}{f_{min}}, \tag{4}$$

where f_{min}/f_{max} is the speed of the processing element with lowest/highest operating frequency in the multiprocessor SoC system.

The resultant task schedule tries to assign task τ_m to processing element with lower operating frequency at which the task satisfies inequality (3). If the task can not be assigned to any processing element, the scheduler backtracks to the immediate predecessor τ_{m-1} of task τ_m and re-assign τ_{m-1} to the processing element with higher frequency. This strategy of task scheduling generate extra slack for task τ_m. The scheduler then re-allocates task τ_m to the processing element with lowest operating frequency. Repeat this process until τ_m is schedulable, or the head of the chain is reached and the task is still unschedulable. If this scenario happens, the schedule is deemed infeasible.

Figure 1 shows the algorithm that generates a static task schedule. In the algo-

rithm, a flag $feasible$ is introduced to indicate the feasibility of a task allocation. The $feasible$ flag is initialized to 1 in line 1. N denotes the number of chains and R denotes the number of PEs. If one task can not be feasibly allocated to any of R processing elements, the tasks in the given task set can not be feasibly scheduled, and the algorithm exits. This process continues until all tasks are feasibly assigned to processing elements, or a task can not be feasibly scheduled on any processing element, whichever comes first. If the inequality (3) holds in line 6, then the task can be feasibly allocated in processing element r as its timing and precedence constraints are satisfied. The PE_r, st_m, ft_m in inequality (1) are updated in turn, that is

$$st_m = \max(ft_{m-1}, PE_r.btimelist)$$

$$ft_m = st_m + \frac{wcet_m}{f_r} \tag{5}$$

$$insert(st_m, ft_m, PE_r.btimelist).$$

When the algorithm updates st_m in equation (5), if the processing element r is busy at the instant ft_{m-1}, the algorithm will search for an idle time interval of processing element r after the instant ft_{m-1} and ensure that the time interval is long enough for the task to complete in the worst case. Then the algorithm updates ft_m and inserts the calculated task execution time as a time interval into the busy time chain of PE_r.

If the inequality (3) in lines 5 to 9 does not hold, the algorithm tries to select the processing element with higher frequency. If the task is assigned to the processing element with the highest frequency and it is still unschedulable, the algorithm reschedules the chain to the processing element with higher frequency. Before rescheduling, it is important that the algorithm should cancel the updates to the predecessors of the task in line 10 to 14. For a predecessor τ_{pre} of the task τ_m, the function

$$delete(st_{pre}, ft_{pre}, PE.btimelist)$$

is utilized to delete the busy time interval of the processing element. If the resultant task schedule is still infeasible after rescheduling the chain to the PE with highest frequency, the task set can not be feasibly scheduled. Conversely, if each task is schedulable in the system, the schedule of the task set is feasible, and the energy consumption of the schedule is greedily minimized.

Figure 2 gives an example of a static task schedule. The target multiprocessor SoC has three processing elements, which are PE_1, PE_2, PE_3, and the application is divided into three task chains. The tasks in a chain are shown using the same color and the precedence constraint is indicated by task index. In a task chain, the task with smaller index is the predecessor of the task with larger index. t_{pr} is the reserved time for fault-tolerance.

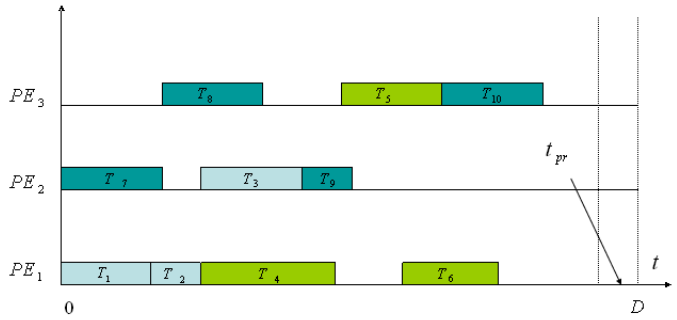

Fig. 2. The example of a static task schedule

3.3. *Task Migration Algorithm*

It has been shown that executing a task as late as possible increases the chance of the task to meet its deadline in energy harvesting systems [1]. This section proposes a Task Movement Algorithm (TMA) to balance system workload among multiple processing elements and further minimizes energy consumption of the static task schedule.

For presentation simplicity, the concept of energy profit is introduced. The energy profit of a task movement is defined to be the energy consumption of the task on the processing element of the initial task schedule minus the energy consumption of the task on the processing element to be moved to. As a result, for a system with R processing elements, each task has R energy profits. To achieve an energy efficient task schedule, tasks in the initial schedule are to be migrated among the processing elements.

For a system with M real-time tasks and R processing elements, a single task has R possible destination processing elements, which include the processing element assigned in the initial schedule. So there are $R \times M$ task movement options in total. A task movement vector (TMV) is proposed to describe a task movement option. The $i^{th}(1 \leq i \leq R \times M)$ task movement option is denoted by TMV_i. TMV_i is defined as $\{tid, st_{tid}, ft_{tid}, PE_{n1}, PE_{n2}, ep, flg\}$, where tid is the index of the task τ_{tid} in the task movement option, st_{tid} and ft_{tid} denote the start time and the finish time of the task, the source and destination processing element are PE_{n1} and PE_{n2}, ep is the energy profit of the task movement option and flg indicates that if the task movement has happened. Let $TMV_i.tid$ indicates the index of the task, and other members can be represented in the same way.

It is assumed that a task τ_{tid} in a chain Γ_i is to be migrated from the processing element n_1 to the processing element n_2. If the inequality

$$st_{tid} \geq ft_{tid-1} \geq 0, st_{tid} + \frac{wcet_{tid}}{PE_{n2}.f} \leq st_{tid+1} \leq D$$

$$1 \leq tid \leq M, 1 \leq n_2 \leq R \tag{6}$$

Procedure (Task initial schedule, PE)

1. **for** $\Gamma_i, 1 \leq i \leq N$ **do**
2. **for** each task τ_m in chain Γ_i **do**
3. **for** $PE_r, 1 \leq r \leq R$ **do**
4. generate & initialize task movement vectors($TMVs$)
5. **end for**
6. **end for**
7. **end for**
8. sort $M \times R$ $TMVs$ in the order of energy profit from high to low
9. $change = 0$
10. **while** *true* **do**
11. **for** $i = 1$ to $M \times R$ **do**
12. **if** $TMV_i.flg$ equals 1 **then**
13. {task $TMV_i.tid$ has been moved once}
14. **end if**
15. **if** inequality (7) holds **then**
16. update $PE_{n1}, PE_{n2}, st_{tid}, ft_{tid}$
17. $change = 1$
18. **end if**
19. **end for**
20. **if** *change* equals 0 **then**
21. **for** $\Gamma_i, 1 \leq i \leq N$ **do**
22. $search(pre, \Gamma_i)$ {search the first task that has not been moved from the tail of the chain}
23. $push2D(pre, PE, deadline)$ {push the task towards the deadline}
24. $change = 1$
25. **end for**
26. **end if**
27. **if** *change* equals 0 **then**
28. break
29. **end if**
30. **end while**
31. **return** *mid*

Fig. 3. Task Movement Algorithm(TMA).

holds, task τ_{tid} can be moved to the processing element n_2. The PE_1 is then updated using the function $delete(st_{tid}, ft_{tid}, PE_{n1}.btimelist)$ to remove the busy interval of task τ_{tid} and the PE_2 is updated using the function $insert(st_{tid}, ft_{tid}, PE_{n2}.btimelist)$ to insert the busy interval of task τ_{tid}. st_{tid} is updated to the larger one of the ft_{tid-1} and $PE_{n2}.btimelist$. This is because the processing element PE_{n2} may be busy at the instant ft_{tid-1} and the algorithm

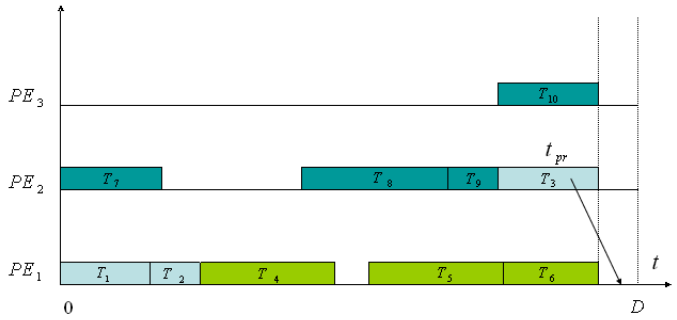

Fig. 4. The example of an optimized schedule

needs to search for the next idle interval after the instant ft_{tid-1} for task τ_{tid}. ft_{tid} is updated to $st_{tid} + \frac{wcet_{tid}}{f_{n2}}$.

Fig. 3 shows the Task Movement Algorithm (TMA). It takes the static initial task schedule as input. Line 1 to 7 generate task movement vectors. For TMV_i, the members of tid, st_{tid}, ft_{tid} are initialized by the initial schedule, energy profit ep is calculated, and flg is initialized to 0, which means that the task movement option does not happen. This algorithm sorts task movement vectors by the order of energy profit from high to low in line 8. This arrangement of task movement vectors is to ensure that the energy consumption of the resultant schedule is less than the initial schedule. A flag *change* is introduced to indicate that if task movement happens in the loop. In line 9, *change* is initialized to 0, which means that there is no task movement in the loop. It is assumed that a single task can be moved only once. The movement can happen among processing elements including the one in the initial schedule. In line 11 to 19, if there is no task movement among processing elements, that is, there is no energy profit, *change* is still 0. Conversely, *change* is set to 1.

In order to minimize energy consumption of task schedule, from line 20 to 26, the algorithm searches a task from the tail of each chain until finding a task that has not been moved and reschedules the task. This strategy can reserve some idle time for some tasks by migrating among processing elements and therefore generates energy profit. In addition, pushing tasks towards deadlines is in favor of harvesting energy in the system. If one task on a processing element has been moved, *change* is set to 1. Otherwise, *change* remains to be 0. If *change* is set to 1, at least one task movement has been performed and the algorithm repeats from line 11 to line 26. If *change* is still 0, the while loop breaks and the algorithm exits. The output of the algorithm is an energy efficient offline task schedule. Fig. 4 gives an example of an optimized static task schedule using the schedule shown in Fig. 2 as input. t_{pr} in Fig. 4 indicates the reserved time for fault-tolerance.

3.4. *Dynamic Adaptation of the Offline Task Schedule to Energy Availability*

The available energy of a energy harvesting system fluctuates with time and is limited by the energy storage capacity. When the offline task schedule is generated, energy limitation is not taken into consideration. If the available energy of the system is not enough for a task to finish in the runtime, the processing element has to stop the execution of the task. As a result, this task and the successors of the tasks will surpass the deadline and the offline task schedule fails. Therefore, the offline task schedule needs to be tailored in the runtime based on the energy information of the energy harvesting system.

The proposed scheme improves the percentage of schedulable tasks in the energy harvesting systems. To achieve this, the offline schedule is generated to minimize the energy consumption of the offline task schedule. In fact, when the available energy in the system varies, minimizing the energy consumption of the offline schedule increases the chances to complete the execution of real-time tasks on time in the runtime. In addition, when the current available energy $E_c(t)$ is small, pushing tasks towards the deadline avoids spending scarce energy early.

In the runtime, only few tasks execute up to their worst case execution time. This characteristic of real-time tasks helps maintain the feasibility of task schedules. The variation of task execution time reduces the energy consumed by real-time tasks, which in turn saves the scarce energy resource.

Assume that the task τ_m is assigned to the processing element PE_n. Let slk_m denote the accumulated slack time before executing task τ_m. The task τ_m has enough energy to finish its execution if the inequality

$$E_c(st_m - slk_m) + E_h(st_m - slk_m, ft_m) \geq E_d(st_m, ft_m) \tag{7}$$

holds, where $E_c(st_m - slk_m)$ is the energy available at the time instant $st_m - slk_m$, $E_h(st_m - slk_m, ft_m)$ is the energy harvested during the period from instant $st_m - slk_m$ to ft_m, and $E_d(st_m, ft_m)$ is the energy demand of executing task τ_m. If inequality (7) does not hold, task τ_m is abandoned. This saves significant energy for remaining tasks to execute successfully. Note that a task is not rescheduled to execute early even if its processing element is idle before its start time. This policy enables the algorithm avoid spending energy on the task too early when system available energy storage is small.

When no faults occur in the system, the execution of real-time tasks follows the static schedule and adapts to energy availability in the runtime. If a task incurs a fault, the task is to be re-executed on the processing elements where it is assigned, and all successors of the task are to be delayed by t_{pr} on their respective processing elements, where t_{pr} is the slack reserved for fault recovery. This approach to fault recovery does not lead to violation of the offline task schedule. In other words, the precedence and timing constraints of real-time tasks remain unchanged in the process of fault recovery.

Table 1. Compare the initial schedule with the optimized schedule in energy efficiency.

# of tasks in a set	deadline(ms)	energy savings
$10-19$	56	10.67%
$20-29$	88	18.98%
$30-39$	134	26.32%
$40-49$	170	29.81%
$50-59$	203	28.47%

4. Experimental Results

Extensive simulation experiments were performed to validate the proposed task allocation scheme for energy efficiency, fault-tolerance, and deadline miss ratio. The proposed task allocation algorithm was implemented in C, and tested on Thinkpad machine with core duo processor of 2.66 GHz and DDR memory of 4 GB. The proposed scheme was compared with the scheme to randomly assign tasks to processing element in energy consumption and deadline miss ration.

Simulations were carried out over 1000 task sets of varing sizes to account for stochastic anomalies. The number of tasks in a task set ranges from 10 to 60 and the tasks were generated by assuming a common deadline. The number of chains in a task set ranges from 3 to 5. Task execution cycles were generated such that each task chain can be scheduled in the processing element with the highest frequency. Precedence constraint is applied to randomly selected tasks in a task set such that the task set consists of several independent task chains.

Solar energy was selected as the energy source in the simulation experiments. The power $P_h(t)$ of the solar source is given by [17]

$$P_h(t) = |F \cdot N(t) cos(\frac{t}{70\pi}) cos(\frac{t}{120\pi})|, \tag{8}$$

where F is a constant scaling unit, and $N(t)$ is a random variable that is normally distributed with mean 0 and variance 1.

Table 1 compares the randomly generated initial task schedule with the optimized task schedule generated using the proposed scheme. It can be derived from Table 1 that the optimized task schedule generated by the proposed allocation scheme achieves energy savings of up to 30% when compared to the randomly generated task schedule. This is because the proposed task allocation scheme greedily assigns tasks to processing elements with lower operating frequency, which effectively reduces system energy consumption.

Table 2 compares the initial task schedule with the optimized task schedule generated using the proposed allocation scheme in task deadline miss ratio and energy efficiency. It is assumed that each task set contains different number of independent task chains. Table 2 shows that the proposed task allocation scheme achieves deadline miss ratio of as low as 8% and energy savings of up to 40%. It can be derived from the table that when the number of chains in a task set increases, the deadline miss ration decreases and the energy savings increases. This

Table 2. Compare the initial schedule with the optimized schedule in deadline miss ratio and energy efficiency.

# of chains in a set	deadline miss ratio	energy savings
3	18%	22.29%
4	16%	20.16%
5	8%	39.56%

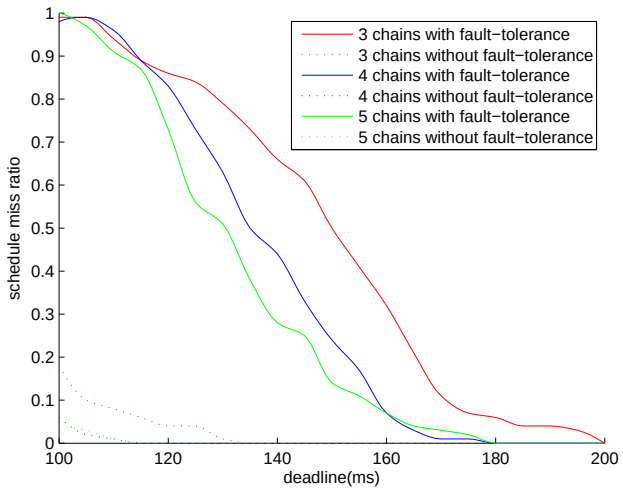

Fig. 5. Compare deadline miss ratio with fault-tolerance consideration

is because the proposed task allocation algorithm takes advantage of parallelism of multiprocessors, and are suitable for task set with multiple chains.

Figure 5 compares deadline miss ratio of real-time task sets with fault-tolerance considered. Three task sets are considered, each of which contains 3, 4, and 5 tasks, respectively. It can be derived from the figure that when the number of chains in a task set increases, the deadline miss ration decreases in the presence of faults. It also can be seen from the figure that when the deadlines lengthen, the deadline miss ratio decreases. This observation remains the same for the scenario in the presence and in the absence of fault occurrences. Overall, it can be concluded that the deadline miss ratio can be reduced by executing tasks in parallel or extending the deadlines of real-time tasks, if allowed.

5. Conclusion

In this paper, an energy efficient fault-tolerance task allocation scheme for multiprocessor SoCs is proposed for real-time embedded energy harvesting systems. The proposed task allocation scheme achieves fault-tolerance by reserving enough slack

on each processing elements and re-executing the faulty task when the fault occurs. Extensive simulation experiments show that the proposed scheme can obtain energy savings of up to 30% and reduce task deadline miss ratio to about 8%.

Acknowledgments

This work was supported in part by the Fundamental Research Funds for the Central Universities of China under the grant No. 78220021.

References

1. T. Wei, P. Mishra, K. Wu, and H. Liang, "Fixed-priority allocation and scheduling for energy-efficient fault-tolerance in hard real-time multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, 2008.
2. S. Roundy, D. Steingart, L. Frechette, P. Wright, and J. Rabaey, "Power sources for wireless sensor networks," *Proceedings of Wireless Sensor Networks*, pp. 1–17, 2004.
3. V. Raghunathan, A. Kansal, J. Hsu, J. Friedman, and M. Srivastava, "Design considerations for solar energy harvesting wireless embedded systems," *International Symposium on Information Processing in Sensor Networks*, pp. 457– 462, 2005.
4. X. Jiang, J. Polastre, and D. Culler, "Perpetual environmentally powered sensor networks," *International Symposium on Information Processing in Sensor Networks*, pp. 463– 468, 2005.
5. K. Gururaj and J. Cong, "Energy efficient multiprocessor task scheduling under input-dependent variation," *Proceedings of the DATE*, 2009.
6. C. Xian, Y. Lu, and Z. Li, "Energy-aware scheduling for real-time multiprocessor systems with uncertain task execution time," *Proceedings of the DAC*, 2007.
7. R. Watanabe, M. Kondo, M. Imai, H. Nakamura, and T. Nanya, "Task scheduling under performance constraints for reducing the energy consumption of the GALS multi-processor soc," *Proceedings of the DATE*, 2007.
8. G. Zeng, T. Yokoyama, H. Tomiyama, and H. Takada, "Practical energy-aware scheduling for real-time multiprocessor systems," *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2009.
9. T.Wei, Y. Guo, X. Chen, and S.Hu, "Adaptive task allocation for multiprocessor socs in real-time energy harvesting systems," *International Symposium on Quality Electronic Design*, 2010.
10. A. Kansal, J. Hsu, S. Zahedi, and M. Srivastava, "Power management in energy harvesting sensor networks," *ACM Transactions on Embedded Computing Systems* (in revision).
11. S. Kang and Y.Leblebici, *CMOS Digital Integrated Circuits Analysis and Design*. McGraw-Hill, 2002.
12. R. Reed, J. Kinnison, J. Pickel, S. Buchner, P. Marshall, S. Kniffin, and K. LaBel, "Single-event effects ground testing and on-orbit rate prediction methods: the past, present, and future," *IEEE Transactions on Nuclear Science*, vol. 50, no. 3, pp. 622–634, 2006.
13. C. Weulersse, G. Hubert, G. Forget, N. Buard, T. Carriere, P. Heins, J. Palau, F. Saigne, and R. Gaillard, "Dasie analytical version: A predictive tool for neutrons, protons and heavy ions induced SEU cross section," *IEEE Transactions on Nuclear Science*, vol. 53, no. 4, pp. 1876–1882, 2006.
14. T. Langley, R. Koga, and T. Morris, "Single-event effects test results of 512MB SDRAMs," *IEEE Radiation Effects Data Workshop*, pp. 98–101, Jul. 2003.

15. D. Mosse, R. Melhem, and S. Ghosh, "Analysis of a fault-tolerant multiprocessor scheduling algorithm," *The 24th International Symposium on Fault Tolerant Computing*, 1994.
16. T. Ishihara and H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," *Proceedings of the ISLPED*, 1998.
17. C. Moser, D. Brunelli, L. Thiele, and L. Benini, "Lazy scheduling for energy harvesting sensor nodes," *Working Conference on Distributed and Parallel Embedded Systems*, 2006.