Reducing the Energy Consumption in Fault-tolerant Distributed Embedded Systems with Time-Constraint *

Yuan Cai[1], Sudhakar M. Reddy[1], Bashir M. Al-Hashimi[2]
[1]Department of Electrical and Computer Engineering, University of Iowa
E-mail: {yucai, reddy}@engineering.uiowa.edu
[2]School of Electronics and Computer Science, University of Southampton
Email: bmah@ecs.soton.ac.uk

Abstract

In this paper we address the problem of reducing the energy consumption in distributed embedded systems associated with time-constraints and equipped with fault-tolerant techniques. A greedy heuristic is presented to reduce the energy during task mapping and fault tolerance policy assignment. Fault tolerance is achieved through task re-execution and replication. The proposed approach can obtain much lower energy consumption compared to the solution without considering energy while tolerating the same number of faults and satisfying the time-constraints. The effectiveness of the proposed approach is evaluated by using extensive experiments.

1 Introduction

Many embedded systems are required to have continuous reliability in addition to high performance and low cost. Examples include telephone systems and industrial control equipments, where the failure of systems will cause huge losses [1]. The reliability of embedded systems is affected by different kinds of faults, including transient, permanent and intermittent faults. Among these faults, the transient fault is much more common than the other two types for a system that was fault-free at deployment. Using redundant hardware has been a commonly used technique to tolerate permanent faults, as well as transient faults [2, 3, 4]. However, these approaches incur large hardware overhead. Since a transient fault has the feature that it occurs once and then disappears, running the task again, i.e., re-executing the task can achieve fault tolerance without extra hardware. In distributed systems, replicating a task on two or more processors is also an alternative to avoid using redundant hardware in processors.

An adaptive checkpoint insertion approach is proposed in [5] to avoid re-executing whole task. Instead, only the portion between two checkpoints has to be re-executed. This reduces the overhead of re-execution, however, the checkpoints themselves will involve overheads. Xie. et al [6], [7] insert task replicas into the idle time slots after tasks have been scheduled. Hence the extent of fault tolerance actually depends on the amount of idle time in the task schedule. [8] reserves re-execution slack during task scheduling for tasks to tolerate transient faults. When a fault is found during the execution of a task, the task will be re-executed in the reserved slack period. It is assumed in [8] that each processor will have at most one fault during one operation of the application. Hence several tasks which are executed consecutively on the same processor can share the reserved slack because if one task has a fault, other tasks will not have any fault. Izosimov et al.[9] find that combining task re-execution and task replication can tolerate the same number of faults while producing shorter schedule length than pure re-execution or pure replication. However, to assign a task the re-execution policy or the replication policy should be decided carefully.

The above works on fault tolerance neglect another important issue in the embedded system design: the energy consumption. In fact, the reliability and the energy consumption of the system are closely related. High power consumption (energy consumption per unit time) would cause high heat dissipation in the system, which will in turn reduce the reliability of the system. Energy reduction techniques used without evaluating their effect on reliability could decrease the reliability. Recently, researchers have begun to consider the problems of energy reduction and fault tolerance together in embedded system design. Zhang et al. introduce dynamic voltage scaling (DVS) into the embedded systems equipped with checkpoints [5, 10, 11]. In these works, after the checkpoints are inserted into tasks, if there is still slack left in the schedule, DVS will be applied to reduce energy consumption by utilizing the available offline slack. Han and Li [12] propose a similar approach to adjust the voltage by using the online slack, but the checkpoints are still determined in the offline phase. Zhu et al. [13] observe that DVS itself will reduce the reliability of the system and study the problem of obtaining the best tradeoff between the energy saving and reliability. Ejlali et al. [14] combine time and information redundancy to tolerate transient faults in DVS-enabled systems such that more slack can be used for DVS. Chen et al. [15] use idle time in the schedule to either duplicate tasks or put the processor into a low power mode. The design purpose is to minimize the product of energy, delay and the inverse of the reliability and assume the system has no time-constraint.

The above energy reduction approaches are applied after the task mapping and after the fault tolerance design has been done and assume the processors have either DVS ability or a low power mode. However, it is well known that task mapping heavily affects the final energy consumption [16] and fault tolerance by using replication will also cause large energy overhead. Hence, we should take the energy consumption into con-

*This work is supported in part by the EPSRC, U. K., under grant GR/S95770

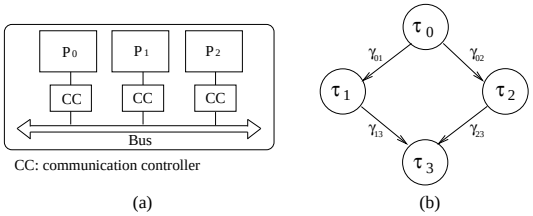


Figure 1. Architecture and application model

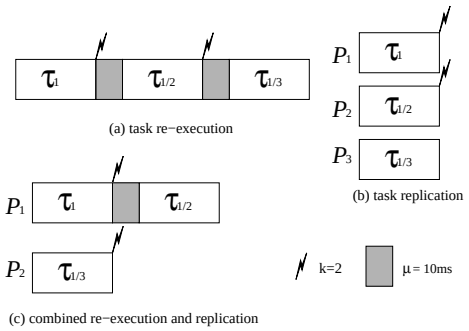


Figure 2. Fault-tolerance techniques [9]

sideration even when there is no DVS ability and low power mode in the processor.In this paper, we address the problem of energy reduction in fault-tolerant distributed embedded systems during task mapping and fault tolerance policy assignment without sacrificing the reliability level (in terms of the number of faults to be tolerated) and the time constraints on the task execution. A greedy heuristic is proposed to effectively solve this problem.

In the remainder of the paper, we introduce the system model and fault tolerance techniques in Section 2. After a motivational example in Section 3, the problem investigated is formulated in Section 4. The proposed algorithm is presented in Section 5 and experimental results are given in Section 6. Conclusions are included in Section 7.

2 Preliminaries

2.1 Architecture and application models

The embedded system considered consists of a set of heterogeneous processors $\mathcal{P}$ and a bus connects every processor $P_i \in \mathcal{P}$ together. Figure 1 (a) gives an illustration of the system architecture where three processors share the bus. Between a processor and the bus, there is a communication controller which implements the protocol services. In this paper, we consider the time-triggered protocol (TTP) [3] which allows each processor to transmit during a predetermined time interval. Details of the implementation of TTP can be found in [3].

The application to be realized on the embedded system can be modeled as a set of directed acyclic graphs $G(\mathcal{T}, \mathcal{C})$ which is usually called task graph [16]. Each node $\tau_i \in \mathcal{T}$ in the task graph represents a computational task and an edge $\gamma_{ij} \in \mathcal{C}$ between tasks τ_i and τ_j denotes the data communication between two tasks. A task can start its execution only after all its inputs have arrived from the bus and it sends out the outputs after it completes the execution. A task graph with four tasks and four edges is illustrated in Figure 1 (b). If two tasks execute on the same processor, then the communication time between them can be regarded as quite small and is neglected. We assume that every task can be executed by any processor in the system. The worst case execution time $C_{\tau_i}^{P_k}$ of task τ_i, when executed on processor P_k, is assumed to be known. Similarly, the transmission time of each data communication is also known. All tasks and communications in the task graph have a common period and a time-constraint $D_{\mathcal{G}}$ is imposed on task graph $\mathcal{G}$, by which time the last task must finish its execution.

2.2 Fault model and fault-tolerance techniques

In this paper, we consider transient faults which are much more common than intermittent and permanent faults in current embedded systems. Specifically, the k-fault model [8, 9, 10] will be used here. In this model, k transient faults will occur during one operation of a task graph. Several faults may appear on several processors at the same time or on the same processor at different moments. A fault can only affect one task and once this fault is tolerated by some technique, the successor tasks will not be affected by the same fault. The total number of faults k can be larger than the number of processors in the system. Each fault has a worst case duration μ, i.e., it takes μ time units for the processor to go back to normal operation after the fault is detected. Fault detection is not considered in this paper and we assume that each processor can detect the fault by some methods, e.g., watchdogs.

We consider two different fault tolerance techniques: task re-execution and replication [9]. Figure 2 [9] gives an example to illustrate the two techniques. In this example, the worst case execution time (WCET) of task τ_1 is 30 ms and there are 2 faults with a duration of $\mu = 10ms$. In Figure 2 (a), the first fault happens at the end of the execution of τ_1. After the fault is detected, the task is re-executed after duration of μ and the re-execution is labeled with $\tau_{1/2}$. Then in the worst case, another fault happens at the end of $\tau_{1/2}$ and the task is re-executed again. The third re-execution $\tau_{1/3}$ will execute without error since both faults have been tolerated. Task replication is illustrated in Figure 2 (b), where task τ_1 is executed on processor P_1 while the two replicas $\tau_{1/2}$ and $\tau_{1/3}$ execute on P_2 and P_3 respectively. No matter where the two faults happen, there will be one replica of the task executing correctly. In addition to pure re-execution or pure replication, by combining these two techniques, faults can also be tolerated, as illustrated in Figure 2 (c). The task is re-executed only once on P_1 in case the fault happens during the first execution of τ_1, but since there is a replica $\tau_{1/3}$ executing on P_2, the two faults can be guaranteed to be tolerated.

3 A Motivational Example

We use a motivational example to show that with the same time-constraints and the same number of faults to be tolerated, the energy consumption of the system can be quite different, hence the energy issue must be carefully considered during

system design.

In Figure 3 (a), we have a task graph with four tasks to be implemented on two processors (P_1, P_2) and $k = 1$ fault with duration of 10 ms must be tolerated. We assume the slot of the bus has a length of 10ms and it is large enough to transmit any single data communication. P_1 and P_2 will use the odd numbered and even numbered slots respectively. The power value of each processor and the WCETs of each task on both processors are shown in the table beside the task graph. The task graph is from [9] and the power values of the P_1 and P_2 correspond to that of AMD K6-2E+ 500MHz and AMD K6-2E 400MHz processors [19]. We assume the power of the bus is 5mW.

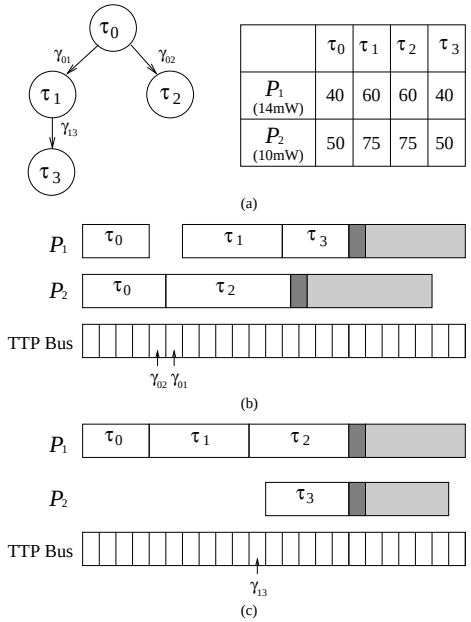
Tasks τ_0, τ_1 and τ_3 execute on P_1 while τ_3 executes on P_2 in Figure 3 (b). The shallow shaded region in the schedule is the reserved slack for re-execution.Task τ_0 is assigned replication policy and the replica will execute on P_2, the other three tasks are assigned re-execution policy. In Figure 3 (c), tasks τ_0, τ_1 and τ_2 execute on P_1 and τ_3 executes on P_2. All tasks are assigned re-execution policy.

Here we can find that that some tasks share one reserved re-execution slack, e.g., τ_1 and τ_3 in Figure 3 (b). This is because we only need to tolerate one fault and if it occurs in one task, other tasks will not be affected. Only tasks scheduled consecutively without interval on the same processor can share the reserved slack and the slack should equal the longest WCET of these tasks [8].

Based on the power value of the processors and the bus, and the WCET of each task, the energy consumption of the system can be computed. In Figure 3 (b), the system consumes 3310 μJ energy, which is the sum of the energy of all tasks including the replica of τ_0 and the two data communications. Similarly, the energy for Figure 3 (c) case can be calculated as 2790 μJ, 15.71% less than that of Figure 3 (b). Though task τ_2 finishes later in Figure 3 (c) than in Figure 3 (b), the final finish time of all tasks, i.e., the schedule length is the same (both are 230 ms) in these two cases. Hence the energy reduction is achieved without sacrificing the schedule length and the number of tolerated faults (both tolerate $k = 1$ fault). How to achieve the minimum energy is the problem considered here and it is formulated in the next section.

4 Problem Formulation

There are three issues to be considered in fault tolerant embedded systems design. 1. *Task mapping*: for a given task graph and an architecture of the embedded system, we must make a decision for each task on which processor it should execute. 2. *Task scheduling*: we need to determine the execution order and start time of all tasks based on their WCETs so that the time-constraints are satisfied. The data communications between each pair of tasks should also be scheduled on the bus. 3. *Fault tolerance policy assignment*: in order to tolerate k faults, we should decide for each task whether it should use re-execution, replication or a combination of the two to achieve fault tolerance. The fault tolerance policy assignment should be carried out together with task mapping and



	τ_0	τ_1	τ_2	τ_3
P_1 (14mW)	40	60	60	40
P_2 (10mW)	50	75	75	50

Figure 3. Task mapping and fault tolerance policy assignment

scheduling since we have to decide the processors on which the replicas will be mapped during task mapping phase and reserve re-execution slack during the task scheduling phase.

Both task mapping and fault tolerance policy assignment affect the energy consumption of the system. Since we assume the system is made up of heterogeneous processors, each processor will consume different amount of energy to execute the same task. Hence different task mappings would cause different energy consumptions. Because replicas are executed on different processors, extra energy is consumed even if no fault appears and the amount of the extra energy depends on both the number and the mapping of the replicas. Note that we compute the energy consumption for the worst case fault scenario [9]. The actual energy consumption depends on when the fault occurs, but since the number of faults is usually much less than the number of tasks, the worst case energy consumption is quite similar to the actual energy consumption.

Let $\mathcal{F}_\mathcal{R} : \mathcal{T} \rightarrow \mathcal{T}_\mathcal{R}$ be the function determining which task is to be replicated and $\mathcal{F}_\mathcal{X} : \mathcal{T} \cup \mathcal{T}_\mathcal{R} \rightarrow \mathcal{T}_\mathcal{X}$ be the function applying re-execution to the tasks, including the replicas in $\mathcal{T}_\mathcal{R}$. We denote the tuple $< \mathcal{F}_\mathcal{R}, \mathcal{F}_\mathcal{X} >$ with $\mathcal{F}$. The mapping of tasks and replicas is given by a function $\mathcal{M} : \mathcal{T} \cup \mathcal{T}_\mathcal{R} \rightarrow \mathcal{P}$. Let $\mathcal{S}$ denote task scheduling, which includes the start time for each task and the TTP slot for each data communication. Now the problem can be formulated as given the task graph $\mathcal{G}$, the set of processors $\mathcal{P}$ and the number of faults k, determine $\mathcal{F}$, $\mathcal{M}$ and $\mathcal{S}$, such that the energy consumption is minimum while the time-constraint of the task graph $D_\mathcal{G}$ is satisfied and k faults can be tolerated.

5 Design Strategy

The problem formulated above is NP complete since both task mapping and scheduling are NP complete [16]. Hence we

propose a greedy heuristic to solve this problem.

5.1 Task mapping and fault tolerance policy assignment

The basic idea of reducing the energy consumption during task mapping and fault tolerance policy assignment is to start from an initial solution and change the mapping and fault tolerance policy of each task iteratively to reduce the energy.

In the initialization phase, each task is assigned re-execution policy and tasks are mapped such that the utilization among processors are balanced. The utilization of a processor is the sum of WCETs of tasks mapped on the processor. This simple mapping procedure and re-execution only policy decides the initial solution quickly. It is not necessary for the initial solution to satisfy the time constraint because the following greedy heuristic will try to find one which satisfies the time constraint. If at the end of the heuristic, no solution can satisfy the time constaint, the design is failed and higher performance processors should be chosen. The bus configuration is also set up in the initialization phase. Each processor $P_i \in \mathcal{P}$ is given a slot in the TTP bus to transfer the outgoing data from the processor. The slot length is set to the length of the largest data communication in the task graph so that any communication can be transfered in one slot. For simplicity, the optimization of the bus configuration is not considered here.

After the initialization, we change the mapping and fault tolerance policy for each task in a greedy manner to reduce the energy. The pseudo code in Figure 4 describes the procedure. We first compute the energy consumption based on the initial mapping $\mathcal{M}^0$ and fault tolerance policy assignment $\mathcal{F}^0$ (line 01). In the outer while loop, tasks are put into a list (line 03). The task consuming maximum energy is found and we try all possible "moves" of this task. A "move" of a task is defined as changing the mapping of the task from its original mapped processor to a new processor, or adding a replica of the task into the task list, or removing a replica of the task from the task list. Though adding a replica will increase the energy consumption, it is necessary since pure re-execution may not satisfy the time constraint.When a replica is added, the incoming and outgoing data communications of task τ are also replicated and added to the task graph together with the replica. The number of replicas of the task is restricted by k since we only need to tolerate k faults. The new replica must be mapped on a processor where task τ and other replicas of τ are not mapped. For every possible move of the task, we schedule all tasks and data communications and compute the resulting energy consumption (line 10 and 11). The scheduling will be discussed in the next section. Among all possible moves, we greedily choose the one which results in the minimum energy while satisfying the time-constraint and label it as the "BestMove"(line 12-14). If this minimum energy is less than the previous energy, there is an improvement (line 16) and the previous energy is set to the minimum energy (line 17). The "BestMove" is applied to the task τ and the procedure goes into the next iteration. If after all the moves of τ have been tried and there is no energy reduction, τ is removed from the task list (line 20). The procedure ends when the task list is empty and there is no improvement.

Input: - $\mathcal{G}, \mathcal{P}, k$

```
01: PreviousEnergy = EnergyCompute(G, P, M^0, F^0);
02: while(improvement == true)
03:   Put all tasks into a task list TL;
04:   improvement = false;
05:     while(TL is not empty)
06:       find the task τ whose energy is maximum;
07:       MinimumEnergy = PreviousEnergy;
08:       for each possible "move" of τ
09:         try move(τ);
10:         ScheduleLength = Scheduling(G, P, M, F);
11:         Energy = EnergyCompute(G, P, M, F);
12:         if(Energy < CurrentEnergy
              && ScheduleLength <= D_G)
13:           MinimumEnergy = Energy;
14:           BestMove = move;
15:       if(MinimumEnergy < PreviousEnergy)
16:         improvement = true;
17:         PrevouseEnergy = MinimumEnergy;
18:         apply BestMove(τ);
19:         break;
20:       remove τ from TL;
```

Figure 4. Pseudo code: the greedy heuristic

We also implemented *tabu search* to do task mapping and fault tolerance policy assignment. Tabu search is a general search procedure designed for solving a wide range of optimization problems [17]. Tabu search does not discard a task if the moves of the task can not reduce the energy. Instead, it associates the task with a variable called *tabu* which is an integer. In each iteration, the task whose tabu is not zero will not be considered but its tabu will decrease by 1. Some search diversification techniques are also included in tabu search and the details can be found in [9]. Hence tabu search is much more complex than the proposed procedure. We will see from the experimental results that greedy heuristic can achieve energy saving quite close to tabu search, but using much less run time.

5.2 Task scheduling

For a given task mapping $\mathcal{M}$ and fault tolerance policy assignment $\mathcal{F}$, the tasks and data communications have to be scheduled. List scheduling is the most commonly used algorithm to schedule tasks with dependent relationships and the complexity of list scheduling is O(n), where n is the number of tasks. In list scheduling, whenever there is a task whose predecessor tasks and incoming data have been scheduled, the task is put into a ready list. All tasks in the ready list are investigated and the task with the highest priority is extracted from the list and scheduled on its mapped processor. Here we use the partial critical path priority presented in [18] as the task priority. After a task has been scheduled, its output data are also scheduled by using the "ScheduleMessage" function from [18] on the TTP bus.

During task scheduling, re-execution slack for tasks to be

Table 1. Experimental results of energy reduction

	20 tasks (k=3)	40 tasks (k=4)	60 tasks (k=5)	80 tasks (k=6)	100 tasks (k=7)
Min. reduction (%)	0.12	3.13	6.64	6.90	22.08
Max. reduction (%)	48.08	70.64	51.33	68.57	67.49
Ave. reduction (%)	14.39	23.35	27.88	32.84	45.98

re-executed is inserted into the schedule. The procedure of inserting the re-execution slack is introduced in [8] where the total amount of slack can be reduced through slack sharing. For example, in Figure 3 (c), tasks τ_0, τ_1 and τ_2 share one slack in the case of one fault.

We should notice that though task scheduling does not consider the energy consumption of the system, it is involved in each iteration of the greedy heuristic in Figure 4 since we must obtain the schedule length and guarantee it satisfies the time-constraint. The fault tolerance is realized in both phases: replicas are added during the task mapping and re-execution slack is inserted during the task scheduling.

6 Experimental Results

We use the task graphs generated in [9] to evaluate the effectiveness of the proposed energy reduction algorithm. The task graphs consist of 20, 40, 60, 80 and 100 tasks. These task graphs are implemented on embedded systems with 2, 3, 4, 5 and 6 processors, respectively and the number of faults k is set to 3, 4, 5, 6, 7 for each system dimension, respectively. The duration μ of the fault is set to 5 ms. The processors have different execution times for the same task and different power consumptions, and we assume the processors have no DVS ability and no low power mode here. For each task graph dimension, fifteen random examples are generated, thus totally 75 task graphs are used for the experimental evaluation. The task graphs are generated with either random structure or more regular structures like trees and groups of chains. The worst case execution time of each task is assigned randomly using both uniform and exponential distribution from 10 to 100 ms.

We first compare the energy consumption of the proposed algorithm with the procedure of [9] where energy is not taken into consideration and only solutions satisfying the time-constraint are given. For all 75 benchmarks, energy can be reduced by using the proposed algorithm while the resulting schedule length are the same as those by the procedure of [9]. That is, we can reduce the energy consumption without sacrificing the schedule length. Meanwhile, the given number of faults are guaranteed to be tolerated. Due to limited space, we do not list the results for all the 75 task graphs. However, Table 1 gives a simple statistic of the results. In the table, the minimum, maximum and average energy reductions in terms of percentage for each size of task graphs are given. It can be seen that the maximum energy saving can be up to more than 70 percent. On average, 14.39% energy can be saved in small task graphs (with 20 tasks) and 45.98% energy can be saved in large task graphs (with 100 tasks). This clearly shows the effectiveness of the proposed energy reduction procedure.

Table 2. Energy reduction (%) of different number of faults

	2 faults	4 faults	6 faults	8 faults	10 faults
Min.	4.02	6.12	7.92	11.58	11.12
Max.	32.57	40.37	45.63	56.03	67.14
Ave.	13.11	23.18	29.60	34.83	38.23

With larger number of tasks, there are more possible task mappings and fault tolerance policy assignments and the chance of finding solutions with lower energy becomes higher. Hence, we can see from the table that the average energy reduction grows with increase in the number of tasks.

It is clear that the number of faults will affect the schedule length. Specifically, to tolerate more faults, the resulting schedule length will become longer [9]. We also find that to tolerate different number of faults, the energy reduction is different. Generally, more energy can be saved when there are more faults because there are more fault tolerance policy assignments and it is possible to find solutions with lower energy consumption. Table 2 shows the energy reduction of different number of faults for task graphs with 60 tasks. On average, the energy reduction increases from 13.11% to 38.23% when the number of faults changes from 2 to 10.

The time constraint is another important parameter affecting the energy consumption. With a loose time-constraint, more tasks can be mapped on processors which execute tasks slowly but consume less energy. Hence the system energy decreases when the time-constraint is relaxed. This is shown in the graph of Figure 5, which is the energy consumption for different time-constraints for one task graph with 100 tasks. In the horizontal axis, "1" means the tightest time-constraint, which is set to the minimum schedule length found by using the procedure of [9] and "1.05" means the time-constraint is extended by 5 percent, "1.1" means the time-constraint is extended by 10 percent, and so on. From the graph, it can be seen that energy consumption drops from more than 1400 mJ to around 950 mJ when the time-constraint is increased by 30 percent. Hence there is a tradeoff between energy and the schedule length, i.e., if more energy has to be saved, we should allow the application to finish in longer time.

Finally, we compare the energy reduction of greedy heuristic with tabu search and the results are shown in Table 3. It can be seen that the average energy reduction of tabu search is more than that of the greedy heuristic, but the difference is quite small. What's more, greedy heuristic takes much less time than tabu search. It takes no more than 10 minutes for greedy heuristic to find the solution for task graphs with 100 tasks, while tabu search needs over 30 hours. This demonstrates that the proposed greedy heuristic is not only effective

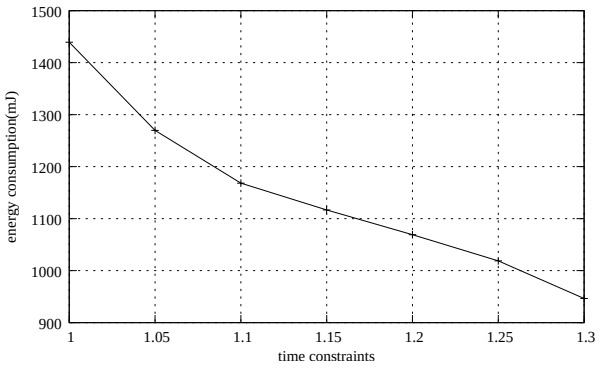


Figure 5. Energy consumptions with different time-constraints

to reduce energy consumption, but also is quite efficient.

7 Conclusions

In this paper, we formulated the problem of reducing the energy consumption in distributed embedded systems with time-constraints and fault tolerance ability. Task replication and re-execution are used to realize fault tolerance. The energy is saved through a proposed greedy heuristic during task mapping and fault tolerance policy assignment. List scheduling is used to schedule tasks and re-execution slack is inserted during the scheduling. The experimental results have shown that taking energy into account during the system design, considerable amount of energy can be saved while satisfying the time-constraint and tolerating a given number of transient faults.

8 Acknowledgments

The authors acknowledge the help of V. Izosimov, P. Pop, P. Eles, and Z. Peng of Linkoping University, Sweden for their technical support of this work.

Table 3. Average energy reduction (%) by two methods

	20 tasks	40 tasks	60 tasks	80 tasks	100 tasks
greedy heuristic	14.39	23.35	27.88	32.84	45.98
tabu search	17.23	25.98	30.27	34.58	46.85

References

[1] R. V. White and F. M. Miles, " Principles of fault tolerance", Proc. of Applied Power Electronics Conference and Exposition (APEC), pp. 18-25, Mar.1996.

[2] V. Claeson, S. Poldena, J. Soderberg, "The XBW Model for Dependable Real-Time Systems", Proc. of Parallel and Distributed Systems, pp. 130-138, 1998.

[3] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwable, C. Senft and R. Zainliner, "Fault-tolerant real-time systems: The Mars approach", IEEE Micro, 9(1), pp. 25-40, 1989.

[4] H. Kopetz, G. Bauer, "The Time-Triggered Architecture", Proceedings of the IEEE, 91(1), pp. 112-126, 2003.

[5] Y. Zhang and K. Chakrabarty, " Energy-Aware Adaptive Checkpointing in Embedded Real-Time Systems", Proc. Design, Automation and Test in Europe (DATE), pp. 918-923, 2003.

[6] Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan and M. J. Irwin, "Reliability-Aware Co-synthesis for Embedded Systems", International Conference on Application-Specific Systems, Architectures and Processors (ASAP), 2004.

[7] J. Conner, Y. Xie, M. Kandemir, R. Dick and G. Link, " FD_HGAC: A Hybrid Heuristic/Genetic Algorithm Hardware/Software Co-synthesis Framework with Fault Detection", Proc. of Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 709-712, 2005.

[8] N. Kandasamy, J. P. Hayes and B. T. Murray, " Transparent Recovery from Intermittent Faults in Time-Triggered Distributed Systems", IEEE Transaction on Computers, Vol. 52, No. 2, Feb. 2003.

[9] V. Izosimov, P. Pop, P. Eles and Z. Peng, " Design Optimization of Time- and Cost-Constrained Fault-Tolerant Distributed Embedded Systems", Proc. of Design, Automation and Test in Europe (DATE), 2005.

[10] Y. Zhang, R. Dick and K. Chakrabarty, " Energy-aware deterministic fault tolerance in distributed real-time embedded systems", Proc. of Design Automation Conference (DAC), pp. 550-555, 2004.

[11] Y. Zhang and K. Chakrabarty, " Task feasibility analysis and dynamic Voltage scaling in fault-tolerant real-time embedded systems", Proc. of Design, Automation and Test in Europe (DATE), pp. 1170-1175, 2004.

[12] J. Han and Q. Li, "Dynamic Power-Aware Scheduling Algorithms for Real-Time Task Sets with Fault-Tolerance in Parallel and Distributed Computing Environment", Proc. of International Parallel and Distributed Processing Symposium, pp. 6-16, 2005.

[13] D. Zhu, R. Melhem and D. Mosse, " The Effects of Energy Management on Reliability in Real-Time Embedded Systems", Proc. International Conference on Computer Aided Design (ICCAD), Nov. 2004.

[14] A. Ejlali. B. M. Al-Hashimi, M. T. Schmitz, P. Rosinger and S. G. Miremadi, "Combined Time and Information Redundancy for SEU-Tolerance in Energy-Efficient Real-Time Systems", IEEE Transaction on VLSI Systems, Vol. 14, No. 4, pp. 323-335, Apr. 2006.

[15] G. Chen, M. Kandemir and E. Li, "Energy-Aware Computation Duplication for Improving Reliability in Embedded Chip Multiprocessors", Proc. of Asia and South Pacific Design Automation Conference (ASP-DAC), Jan. 2006.

[16] M. T. Schmitz, B. M. Al-Hashimi, P. Eles, *System-Level Design Techniques for Energy-Efficient Embedded Systems*, Kluwer Academic Publishers, 2004.

[17] F. Glover and M. Laguna, *Tabu Search*. Kluwer Academic Publishers, 1997.

[18] P. Eles, A. Doboli, P. Pop and Z. Peng, "Scheduling with bus access optimization for distributed embedded systems", IEEE Transactions on VLSI Systems, Vol. 8, Issue 5, pp. 472-491, 2000.

[19] E3S benchmark suit. http://www.ece.northwestern.edu/ dickrp/e3s