

Reliability-Aware Co-synthesis for Embedded Systems

Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin
Department of Computer Science and Engineering
Pennsylvania State University
{yuanxie, lili, kandemir, vijay, mji}@cse.psu.edu

Abstract

As technology scales, transient faults due to single event upsets have emerged as a key challenge for reliable embedded system design. This paper proposes a design methodology that incorporates reliability into hardware-software co-design paradigm for embedded systems. We introduce an allocation and scheduling algorithm that efficiently handles conditional execution in multi-rate embedded systems, and selectively duplicates critical tasks to detect soft errors, such that the reliability of the system is increased. The increased reliability is achieved by utilizing the otherwise idle computation resources and incurs no resource or performance penalty. The proposed algorithm is fast and efficient, and is suitable for use in the inner loop of our hardware/software co-synthesis framework, where the scheduling routine has to be invoked many times.

1. Introduction

Embedded systems are of great economic importance. Embedded applications include consumer electronics appliance, signal processing, automobile control, aircraft autopilot, and so on. It is estimated that the embedded system market sales are approximately \$77 billion yearly and there were nearly 2 billion embedded systems sold in 2003 [1]. Compared to high-performance computing systems, embedded systems are more cost sensitive and require shorter time-to-market. To reduce the design effort and the cost of the system and shorten the time-to-market, the designer has to come up with a heterogeneous system, which usually consists of embedded processors and ASICs. Most embedded systems are heterogeneous multiprocessors with several different types of processing elements, including customized hardware processing elements as well as programmable CPUs.

A soft error, also called single event upset (SEU), is a “glitch” in a semiconductor device [15]. These glitches are random, usually not catastrophic, and they do not normally destroy the device. Because of technology scaling, drastic shrinking in device sizes, associated with reduction in operating voltages and increase in clock frequencies, combinational logic and sequential logic are becoming increasingly susceptible to soft errors from natural ground level radiation. Table 1 shows the Mean Time Between Failures (MTBF) due to soft errors for some typical embedded systems (taken from [5]).

Consequently, detecting errors and providing reliable execution in the existence of soft errors is becoming increasingly critical for embedded systems, especially for mission-critical applications. In this paper, our main focus is on detecting soft errors within the hardware/software co-design paradigm.

Hardware/software co-synthesis is typically the first step in an embedded design procedure, which partitions the system specification into hardware and software modules to meet performance,

Table 1. MTBF for embedded systems due to soft errors.

	MTBF (Hours)	
	0.13um	90nm
Ground-based	895	448
Civilian Avionic Systems	324	162
Military Avionic Systems	18	9

power and cost goals. A common model to describe an application is the task graph. The co-synthesis process synthesizes a distributed multiprocessor architecture and allocates tasks to the target architecture, such that the allocation and scheduling of the task graph meets the deadline of the system, while the cost of the system is minimized. The algorithm presented here targets a periodic multi-rate task graph. The target architecture is a heterogeneous multiprocessor architecture that consists of multiple processing elements (PEs) of various types: general-purpose CPUs or domain-specific CPUs and ASICs. Due to the data dependency and control dependency within the application tasks, not all the computing resources are fully utilized all the time. Therefore, it is possible to selectively duplicate some tasks and utilize idle resources to detect soft errors, such that the overall reliability of the system is improved.

This paper is organized as follows. Section 2 reviews the related work. Section 3 describes the problem formulation, and Section 4 presents the proposed scheduling and allocation algorithm. In Section 5, we discuss the experimental results of our algorithm. Finally we conclude the paper in Section 6.

2. Related Work

Transient faults due to single event upsets have emerged as a key challenge that may become the new obstacle in front of technology scaling in the next few design generations. Consequently, we have seen a rapidly growing interest in reliability-related issues caused by soft errors. A variety of soft error-tolerant techniques have been proposed, ranging from special radiation-hardened circuit designs to localized error detection and correction to architectural-level redundancy [9, 8, 2, 11, 10]. However, these approaches usually introduce some sort of penalty in performance, power, die size, and design time.

Even though the reliability issues caused by soft errors have been investigated from circuit level up to microarchitecture level, most of the prior work in hardware/software co-design explore the tradeoffs among performance, power and cost, and reliability has not explicitly been taken into account during the design flow. As a result, there exist very few fault-tolerant hardware/software co-design studies. COFTA [6] was proposed as a co-synthesis framework for heterogeneous distributed embedded systems for fault tolerance. The fault detection capability is imparted to the embedded system by adding assertion and replicate-and-compare tasks to system specification prior to co-synthesis. The dependability (reliability and availability) of the architecture is evaluated during co-synthesis. Their system specification task graph considers only data dependencies. The average cost increase due to their fault-tolerant synthesis is almost 60% compared to a system without fault-tolerant features. Recent work [3, 4] has investigated the problem of reliable co-design by introducing reliability property into the specification, and provided an enhancement to the traditional co-design flow. However, their target architecture was a simplified one that had only one CPU plus one coprocessor. In this paper, we propose a co-synthesis design methodology to improve the system reliability by selectively replicating tasks and utilizing idle computing resources without

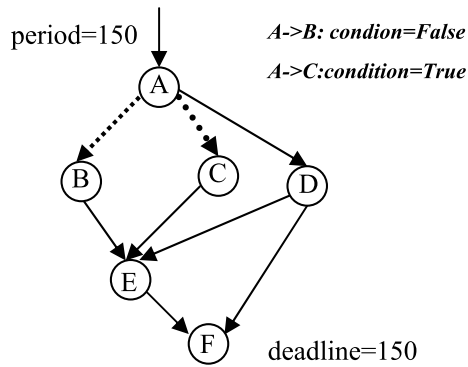


Figure 1. Conditional task graph.

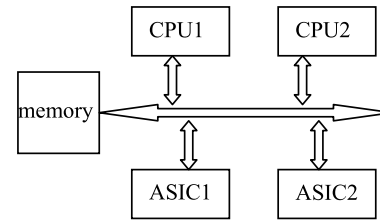


Figure 2. Target architecture.

performance or cost penalty.

3. Problem Formulation

The real-time applications running on embedded systems are periodic, running at multiple rates. A common model to describe the applications is the task graph model [12, 13]. In this model, an application is mapped to a task graph, which is a directed acyclic graph, as shown in Figure 1. Each node in the task graph represents a task that may have moderate to large granularity; the directed edges represent data dependencies and/or control dependencies between the tasks. An edge, say $A \rightarrow D$, implies that task D cannot start execution until A is finished. Data dependency edges ensure the correct order of execution. The task with an output conditional edge is a *branch fork* task. Conditional paths meet at a *branch join* task. For example, in Figure 1, A is a branch fork task, and E is a branch join task. Depending on the condition, one of the out-branches of task A ($A \rightarrow B$ or $A \rightarrow C$) is activated. Each edge is associated with a scalar describing the amount of data that must be transferred between the two connected tasks, which determines the communication time between the tasks if they are allocated on different PEs. We assume that, if two tasks are allocated on the same PE, the communication time is 0. An edge with an assigned condition value is a conditional edge (represented with dotted lines in Figure 1). The task graph is executed periodically at its specified rate. For simplicity, in this paper, we assume that the deadline, by which the task graph must complete its execution, is equal to the period. The deadline can actually be shorter or longer than the period.

The target architecture in our co-synthesis framework is a heterogeneous architecture as shown in Figure 2. This architecture has a number of processing elements (PEs), which may be CPUs or ASICs. PEs communicate with each other via communication links (e.g. a shared memory bus).

Each task can have several implementation options differing in PE type, cost and execution time. The technology library provides a number of choices of the types of CPUs and the worst case execution time (WCET) for the tasks on each type CPU.

Given the conditional task graph, target architecture and technology library, the co-synthesis algorithm chooses the number and type of the processing elements from the target library, produces an allocation of tasks on target architecture and constructs the static global schedule of the tasks on specified processing elements, with minimal system cost while meeting the deadline.

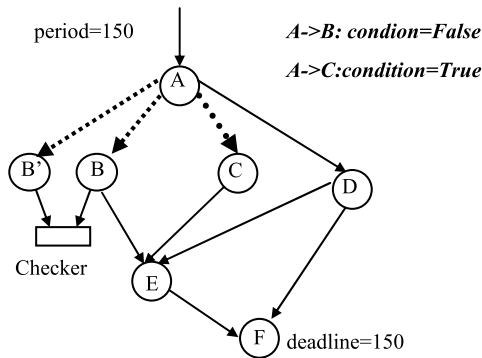


Figure 3. Task duplication.

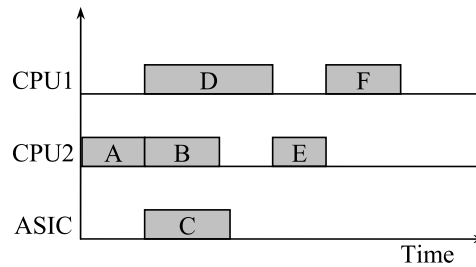


Figure 4. Scheduling without reliability consideration.

4. Reliability-aware Co-synthesis Framework

In this section, we describe the proposed reliability-aware co-synthesis framework. The objective of the proposed approach is to provide maximum reliability via task duplication, without performance or cost penalty.

4.1. Improving reliability via task duplication

To integrate reliability into this co-synthesis paradigm, the first step is to decide the criticality of each task in the task graph. Some tasks can tolerate a certain degree of soft errors. For example, in a video application, soft errors can manifest themselves as missing or wrong colored bits on a display screen. These errors may not be noticeable or important to the user. However, when memory elements are used to control the functionality of the device such as a program counter register or branch target address, soft errors can have a much more serious impact and lead not only to corrupt data, but also to a loss of functionality and critical failures.

In our co-synthesis framework, there are two ways to decide the criticality of the task. The first approach is user-defined, which is based on the knowledge of the application itself. The second one is a heuristic approach. If we do not have enough information about the application and cannot decide which task is more critical than the others, then we can assign the relative criticality to a task based on the number of tasks whose correct operation depends upon the successful execution of the task under consideration. As a result, the tasks closest to the source task will most likely have higher criticalities. The criticality of the tasks in Figure 1, for example, can be ranked as $\{C_A = 6, C_B = 3, C_C = 3, C_D = 3, C_E = 2, C_F = 1\}$.

The improvement of the system reliability is through the duplication of critical tasks. We refer to the duplicated copy of task X as X', which maintains the same characteristics as task X. X' also copies all the in-edges of X with the same conditions, but it does not need to copy the out-edges of task X. An extra checker task and the edges from X and X' to the check task are needed, as shown in Figure 3 when duplicating task B.

Note that adding reliability as a metric to the co-design flow may have a significant impact on the resulting system as well as on performance and power consumption. As an example, consider the task graph in Figure 1. Let us assume that, if we do not consider reliability, we can allocate and schedule the task graph on two CPUs and one ASIC as shown in Figure 4. However, if task B is a critical task (i.e., needs to be protected against soft errors) and the reliability issue is taken into account, then we may have to replicate task B (denoted B'), and another task (denoted as

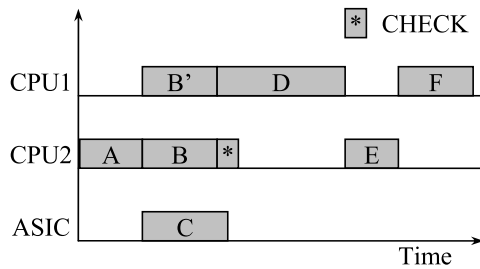


Figure 5. Reliability-aware scheduling. A duplicate of task B (B') and a checker task (*) are inserted into the schedule, resulting a degraded performance.

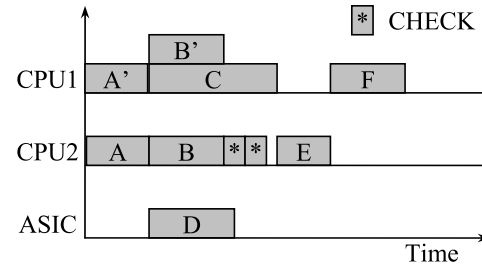


Figure 6. Reliability-aware scheduling, which takes advantage of the mutual exclusiveness of task B and C, and the idle time slot at CPU 1, without performance or cost penalty.

```

ASP (task_graph, target_library)
1. for each task, calculate static_urgency(task)
2. if there is task i in ready list and partitioned on ASIC
3.   schedule task i; goto 8
4. else
5.   for each ready task i and each CPU pe-j
6.     calculate dynamic_urgency(task-i, pe-j)
7.     schedule task-i on pe-j with maximum dynamic_urgency value
8. update ready task list and goto 2 until all tasks are scheduled

```

Figure 7. Allocation and scheduling procedure scheduled. (ASP).

“CHECK” in the figure) is also necessary to compare the results of B and B'. With the same amount of resources, one possible allocation and scheduling could be as illustrated in Figure 5, which indicates performance degradation. To satisfy both performance and reliability requirement, one may opt to use more resources, for example, adding extra CPUs or ASICs. This can potentially eliminate the performance degradation at the expense of increased overall system cost.

However, we notice that task B and C can share the same CPU resource because they are mutual exclusive and will not be activated at the same time, because they belong to different conditional branches. Furthermore, there are some idle time slots for CPU1 and CPU2 such that we may fill in those time slot with duplicated tasks without incurring performance and cost penalty. By taking advantage of these facts, a better synthesis result is shown in Figure 6, where both task A and task B are duplicated, such that the overall reliability of the system is improved while the performance and system cost is not affected.

4.2. Allocation and scheduling algorithm

Our allocation and scheduling algorithm is based on the co-synthesis framework ASICosyn [14]. It performs allocation of the tasks on CPUs and scheduling of the tasks on CPUs and ASICs simultaneously such that the algorithm can take advantage of the resource sharing among these mutual exclusive tasks that belong to different branches. Figure 7 outlines the proposed allocation and scheduling procedure (ASP).

```

Schedule_with_Reliability_Inc (S_R_INC)
1. call ASP to get the task schedule graph without duplicating tasks
2. record the cost as MIN_COST
3. for each task with criticality from high to low
4.   do {
5.     duplicate this task and build new task graph
6.     call ASP to update allocation and schedule
7.     if cost > MIN_COST
8.       delete the duplicated task
9.   }
10. return the allocation and schedule

```

Figure 8. Incremental scheduling algorithm.

The *static urgency* (SU) is calculated for each task based on the maximum distance of the task to the end task of the task graph. This is similar to the priority assignment in some list schedulers. For the branch fork task, the longest branch path is used to calculate the static urgency.

The *dynamic urgency* (DU) is defined as:

$$DU(task, CPU) = SU(task) - \max\{ready_time(task), CPU\ available\ time\} - WCET(task, CPU)$$

The dynamic urgency is actually related to the following factors:

- Static urgency (SU). If a task's SU is high, it implies that this task is a critical task and should be given a high priority.
- The earliest start time of the task on the CPU. Note that the *ready_time(task)* takes into account the communication time from its predecessor. We assume that the communication time between two tasks on the same CPU is 0. The mutually exclusive communication edges can share the same communication link with overlapping time slots.
- The worst case execution time (WCET) of this task on the CPU.

When the CPU available time is calculated, if the allocated task is mutually exclusive with the ready task, then these two tasks can share the same time slot of the processing element to share the resource. The method of detecting whether two tasks are mutual exclusive can be found in [14].

4.3. Inserting duplicated tasks

In this section, we present two algorithms to insert duplicated tasks such that the overall reliability of the system is improved.

The first algorithm (S_R_INC) is an incremental scheduling algorithm as shown in Figure 8. We first call the original allocation and scheduling procedure ASP, which is shown in Figure 7, to get an initial task schedule and resource cost (recorded as MIN_COST); then we duplicate one task each time following the criticality ranking from high to low. The algorithm then calls the ASP to get a new task schedule and new cost. If the new cost is larger than the MIN_COST, it means that the cost limitation is exceeded because of duplicating this task. Therefore, we should not duplicate this task. The procedure is iteratively executed for all tasks.

```

Schedule_with_Reliability_Static (S_R_STA)
1. call ASP to get the task schedule graph without duplicating tasks
2. for each task with criticality from high to low
3.   do {
4.     search all candidate idle time slot in the whole task schedule graph
5.     select the best time slot for the task
6.     if time slot available, insert duplicated task and adjust task schedule
7.   }
8. return the allocation and schedule

```

Figure 9. Static scheduling algorithm.

The second scheduling algorithm (S_R_STA) is shown in Figure 9. This algorithm calls the allocation and scheduling procedure (ASP) only once, and adjusts the schedule directly when inserting duplicated tasks.

For each task with the priority from high to low criticality, we try to find the best idle time slot for it. The first step is to find all candidate idle time slots in the schedule for the task, including the real idle time slots or the time slots occupied by other tasks which are mutual exclusive with current task (due to different execution conditions). For the idle time slot that are long enough to schedule a duplicated task without affecting other tasks, we call it *long-time-slot*; for the idle time slot that are too short (i.e., we have to postpone the execution of other tasks if a duplicated task is put into that slot), we call it *short-time-slot*.

The second step is to find the most suitable time slot for the target task based on the following heuristic rules:

- Long-time-slot is better than short-time-slot
- among all long-time-slots
 - the shorter execution time of task on that PE is better
 - the earlier time slot is better
- among all short-time-slots
 - calculate the new completion time due to the postponement of all adjacent tasks in the same PE and all following tasks in the other PEs.
 - the one results in the earliest completion time is better.

If the new completion time is later than the deadline, the time slot is not considered anymore. The delay may not affect the performance concern (i.e., the new completion time is still earlier than deadline), when the delay is absorbed by another time slot. We also notice that, if a task is postponed, all adjacent tasks in the same PE and all adjacent following tasks in the task graph need to be postponed.

The first algorithm (incremental scheduling) can achieve a better reliability-aware schedule result, because it calls the allocation and scheduling procedure ASP whenever a duplicated task is inserted into the task graph, such that a new architecture with new allocation and scheduling is generated for the enhanced task graph. The second algorithm (static scheduling) only calls the allocation and scheduling procedure ASP once, and does not modify the architecture and the allocation. It only inserts duplicated tasks and adjust the schedules on each PE. Therefore, the first algorithm is expected to generate better result at the expense of much longer runtime.

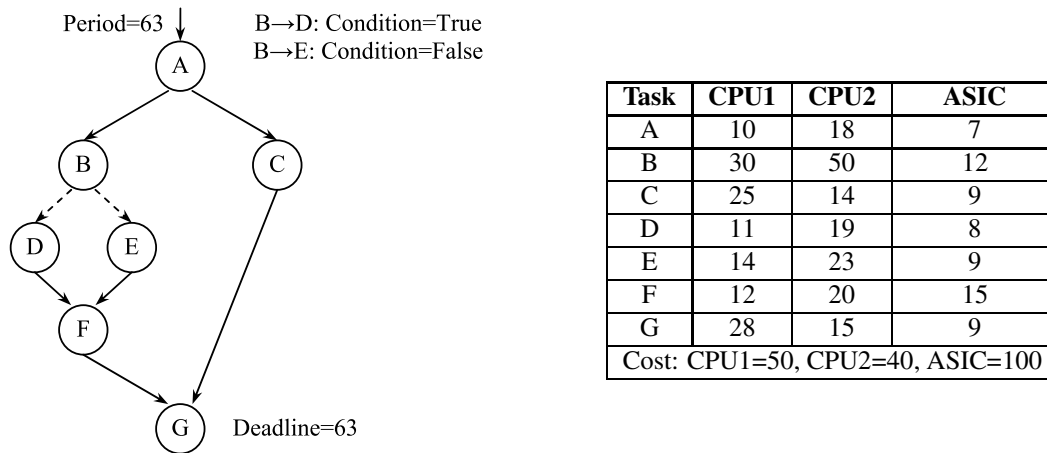


Figure 10. A simple example.

5. Experimental Results

The co-synthesis framework is implemented in C++ and the experiments are run on a Linux machine with an Intel Pentium 4 processor (2.8 GHz / 512M RAM). We denote the first reliability-aware algorithm (incremental scheduling) as S_R_INC and the second reliability-aware algorithm (static scheduling) as S_R_STA. We also compare the reliability-aware scheme with the scheme without duplicating tasks (S_MINC) based on the original allocation and scheduling procedure (ASP).

First we show a simple example. The input to the co-synthesis framework is the task graph shown in the left part of Figure 10, and a technology library shown in the right part of Figure 10. The technology library specifies the execution time for each task on different process elements and the cost of processing elements. Note that CPU resource can be shared by tasks, but ASIC is application specific and can not be shared by two different tasks. For simplicity, we assume all ASIC implementation costs for different tasks have the same cost here, even though usually they are not the same due to difference in implementation.

Figure 11(a) shows the result of task allocation and scheduling from scheme S_MINC, which employ minimum cost to achieve the deadline requirement. None of the tasks is duplicated. Figure 11(b) shows the result of scheme S_R_STA (for this example, the result of S_R_INC is the same). Reliability-aware schemes S_R_STA and S_R_INC try to duplicate as many tasks as possible to increase the reliability while maintain the same system cost and performance requirement. Due to the comparison between tasks, an extra checker task is also needed for each extra duplicated task (usually the execution time of this checker task is small, since its only functionality is to compare the result of two tasks. In this example, we assume that the execution time of a checker task is 1).

We notice that the degree of duplication without extra cost is dependent on the intrinsic feature of the input task graph. In this example, due to the tight deadline, the tasks are allocated to the processing elements with shorter execution time. Therefore, tasks F and G cannot be duplicated to achieve the full duplication. Our schemes duplicate 71% of the tasks without incurring performance or cost penalty.

We use the TGFF tool [7] to generate benchmark task graphs and apply our schemes to evaluate the efficiency of our algorithms. Some statistics of these task graphs are listed in the first four columns of Table 2. The execution time (in seconds) of different schemes are shown in the middle three columns of Table 2. We notice that the S_R_STA has almost the same performance as the

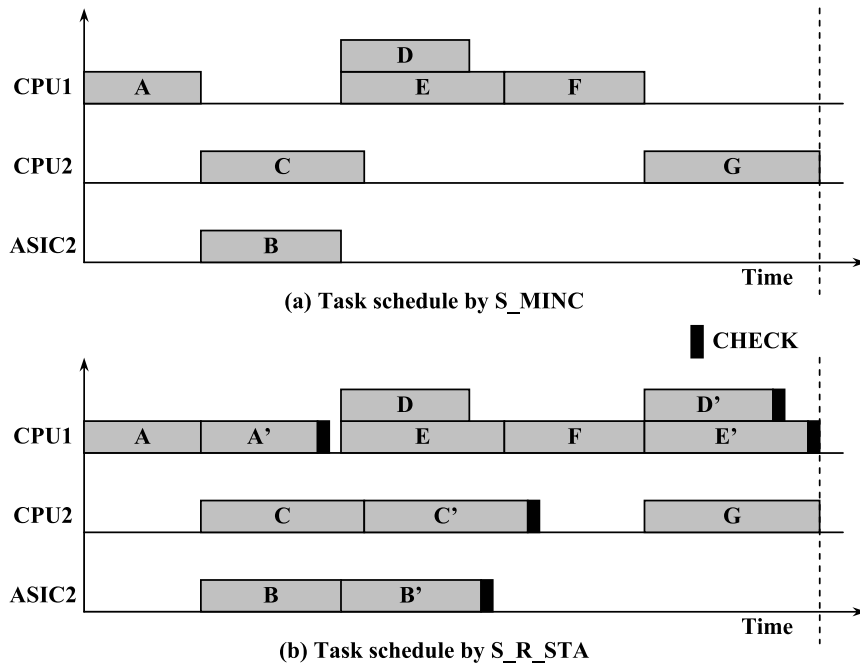


Figure 11. Task schedule by (a) S_MINC, (b) S_R_STA.

original algorithm, S_MINC. This means that the static post-analysis scheme introduce very small runtime overhead. On the other hand, the execution time of S_R_INC is significantly longer than the original and S_R_STA schemes.

The number of original tasks and duplicated tasks by S_R_STA and S_R_INC are shown in the first and last four columns of Table 2. S_R_STA schemes can duplicate average 37.2% tasks and S_R_INC can duplicate 38.5% tasks in average. In most cases, S_R_STA scheme can achieve the same duplication rate as the S_R_INC scheme. Therefore, we conclude that S_R_STA algorithm is a preferable choice than S_R_INC, since its runtime is much shorter while the reliability improvement (in terms of how many tasks are duplicated) is almost the same.

Table 2. Characteristic of benchmarks and experimental results.

	Tasks	Edges	Deadline	Execution Time			Duplicated Tasks			
				MINC	R_STA	R_INC	R_STA	%	R_INC	%
s01	14	16	625	0.01	0.01	0.09	6	42.9%	6	42.9%
s02	17	20	700	0.01	0.01	0.18	6	35.3%	6	35.3%
s03	22	28	625	0.01	0.01	0.63	6	27.3%	7	31.8%
s04	25	32	750	0.01	0.02	1.18	6	24.0%	8	32.0%
s05	28	32	1200	0.01	0.01	1.51	20	71.4%	20	71.4%
s06	31	38	600	0.03	0.03	3.52	10	32.3%	10	32.3%
s07	32	34	900	0.02	0.02	4.08	24	75.0%	24	75.0%
s08	38	53	1000	0.05	0.05	6.06	10	26.3%	10	26.3%
s09	39	46	800	0.09	0.09	9.28	8	20.5%	8	20.5%
s10	43	52	1000	0.13	0.14	21.30	13	30.2%	13	30.2%
s11	50	60	1000	0.26	0.26	34.38	12	24.0%	13	26.0%

6. Conclusion and Future Work

As technology scales, soft errors become a big concern for reliable embedded system design. We propose a design methodology that selectively duplicates critical tasks to detect soft errors, such that the reliability of the system is increased. The increased reliability (via selectively duplicate tasks) utilizes the idle computation resource and takes advantage of the mutual exclusiveness among those tasks that have different execution conditions. The improvement of reliability incurs no resource or performance penalty.

Currently, the reliability is improved by adding error detection through duplicating tasks. Our future work includes introducing error correction features during system co-synthesis.

7. Acknowledgments

This work was supported in part by grants from MARCO/DARPA GSRC-PAS, NSF Grants 0103583, 0082064, 0130143, 0202007 and CAREER Awards 0093082, 0093085.

We would like to thank Dr. Sanjay Rajopadhye and the anonymous referees for providing helpful comments.

References

- [1] <http://www.embedded.com.au/market.html>.
- [2] Todd M. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO)*, pages 196–207, 1999.
- [3] C. Bolchini, L. Pomante, F. Salice, and D. Sciuto. Reliability properties assessment at system level: a co-design framework. In *Proceedings of the Seventh International On-Line Testing Workshop*, pages 165–171, 2001.
- [4] C. Bolchini, L. Pomante, F. Salice, and D. Sciuto. A system level approach in designing dual-duplex fault tolerant embedded systems. In *Proceedings of the Eighth International On-Line Testing Workshop*, pages 32–36, 2002.
- [5] Actel Corporation. Effects of neutrons on programmable logic. *white paper*, December 2002.
- [6] B. P. Dave and N. K. Jha. COFTA: Hardware-software co-synthesis of heterogeneous distributed embedded systems for low overhead fault tolerance. *IEEE Transactions on Computers*, 48(4):417–441, 1999.
- [7] Robert P. Dick, David L. Rhodes, and Wayne Wolf. TGFF: task graphs for free. In *Proceedings of the 6th International Workshop on Hardware/Software Codesign*, pages 97–101, 1998.
- [8] D. Mavis and P. Eaton. Soft error rate mitigation techniques for modern microcircuits. In *Proceedings of Reliable Physics Symposium*, pages 216–225, 2002.
- [9] K. Mohanram and N. Touba. Cost-effective approach for reducing soft error failure rate in logic circuits. In *Proceedings of International Test Conference*, pages 893–901, 2003.
- [10] S. Mukherjee, M. Kontz, and S. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 99–110, 2002.
- [11] A. Orailoglu and R. Karri. A design methodology for the high-level synthesis of fault-tolerant ASICs. In *VLSI Signal Processing V*, pages 417–426, 1992.
- [12] W. Wolf and J. Staunstrup. *Hardware/Software co-design Principles and Practice*. Kluwer Academic Publishers, 1997.
- [13] Y. Xie and W. Wolf. Allocation and scheduling of conditional task graph in hardware/software co-synthesis. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE)*, pages 620–625, 2001.
- [14] Y. Xie and W. Wolf. ASICosyn: Co-synthesis of conditional task graphs with custom ASICs. In *Proceedings of the International Conference on ASICs*, pages 130–135, 2001.
- [15] J. F. Ziegler. IBM experiments in soft fails in computer electronics (1978-1994). *IBM Journal of Research and Development*, 40(1):3–18, 1996.