



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, The Pennsylvania State University, University Park, PA, USA

Received: 4 March 2005; Revised: 29 March 2006; Accepted: 13 October 2006

Abstract. As technology scales, transient faults 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 or correct transient errors, such that the reliability of the system is improved. Two methods are proposed to insert duplicated tasks into the schedule. The improved reliability is achieved by utilizing the otherwise idle computation resources and taking advantage of the overlapping schedule for mutually exclusive tasks in the conditional task graph, such that it incurs no resource or performance penalty.

Keywords: design methodology, embedded system design, scheduling algorithm

1. Introduction

Embedded systems are of great economic importance. Embedded applications include consumer electronics, signal processing, and automobile control. The worldwide embedded systems market was estimated at \$45.9 billion in 2004 and was expected to grow at an average annual growth rate of 14% over the next several years to reach \$88 billion by 2009 [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, designers have 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 (PEs) as well as programmable CPUs.

One of the grand challenges for embedded system designers in nanometer VLSI era is guaranteeing reliability. Shrinking geometries, lower supply voltage, higher frequencies, higher power density, and higher density circuits all have a negative impact on reliability: the occurrences of transient errors increases due to these factors [19]. Transient faults are errors caused by temporary environmental condition (such as power supply noise and interconnect noise) or by special environmental condition (neutron and alpha particle, also called soft errors [12, 14]). The circuit itself is not damaged even though computational errors are introduced. Table 1 shows the Mean Time Between Failures (MTBF) due to soft errors for some typical embedded systems (taken from [12]). Consequently, detecting and correcting errors to provide reliable execution in the existence of transient faults is becoming increasingly critical for embedded systems, especially for mission-critical applications. In this paper, our main focus is on

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

	MTBF (Hours)	
	0.13 μm	90 nm
Ground-based	895	448
Civilian avionic Systems	324	162
Military avionic systems	18	9

detecting and correcting transient faults within the hardware/software co-design paradigm.

The first step in an embedded design flow is typically called hardware/software co-synthesis, which partitions the system specification into hardware and software modules to meet performance, power, and cost goals. A common model to describe an application is the task graph. The co-synthesis process normally 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 (explained in Section 3). 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 among 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 related work; Section 3 describes the problem formulation; Section 4 presents the proposed scheduling and allocation algorithm; Section 5 discusses the experimental results of our algorithm, and Section 6 concludes the paper.

2. Related Work

Hardware/software codesign moved from an emerging discipline to a mainstream technology since it was introduced about a decade ago [22]. Early co-design research focused on performance and cost tradeoffs [21] to minimize system cost under performance constraints. Starting from mid 1990s,

low-power embedded system design became a dominant theme, and prompted the co-design community into techniques for low-power embedded system synthesis. Hardware/software co-design techniques targeted at low-power embedded systems were used to explore the design space for tradeoffs among power, performance, and cost [23].

Fault tolerance has long been used as a means of improving the reliability of the VLSI chips. System reliability can be ensured by using different redundancy techniques, such as hardware redundancy, software redundancy, information redundancy, and time redundancy [15]. Various methods for reducing the transient faults in deep submicron ICs have been proposed [2, 7–10, 16]. Prior work on design automation algorithms for fault-tolerant VLSI was primarily focused on high-level synthesis (HLS) [7, 17, 20]. All of these approaches usually introduce redundancy to improve reliability and incur penalty in performance, power, die size, and design time.

Even though the reliability issues caused by transient faults 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. Reliability has not explicitly been taken into account during the design flow. System-level hardware/software co-design for reliability is not a well studied area, and the improvement of the system reliability is at the expense of performance and cost. As a result, there exist very few fault-tolerant hardware/software co-design studies. COFTA [5] 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 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 such applications is the task graph model [13]. In this model, an application is mapped to a task graph, which is a directed acyclic graph, as shown in Fig. 1. Each node in the task graph represents a task that may be of moderate to large granularity; the directed edges represent data dependencies and/or control dependencies between 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 Fig. 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 ($A \geq B$ or $A \geq C$) is activated. Each edge is associated with a scalar describing the amount of data that must be transferred between the two connected tasks. We assume that, if two tasks are allocated on the same PE, the communication time is 0. If two tasks are allocated on different PEs, the communication time depends on the data amount to be transferred and the communication link type (for example, bus bandwidth and bus congestion). An edge with an assigned condition value is a condi-

tional edge (represented with dotted lines in Fig. 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.

In our co-synthesis framework the target architecture is a heterogeneous one as shown in Fig. 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, the target architecture, and the 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.

4. Reliability-aware Co-synthesis Framework

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

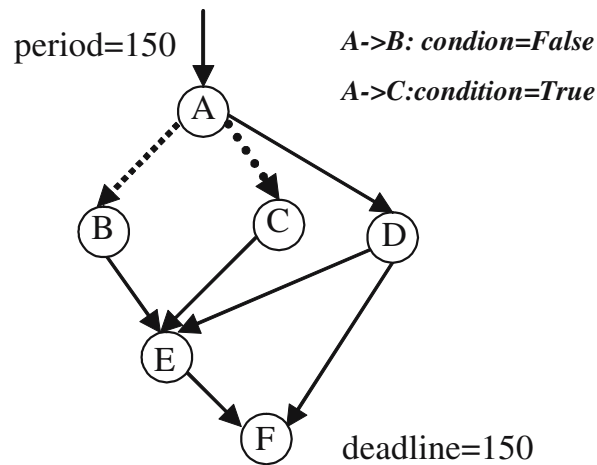


Figure 1. Conditional task graph.

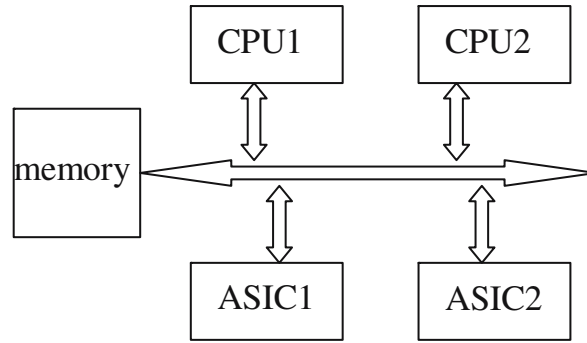


Figure 2. Target architecture.

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 transient errors. For example, in a video application, transient 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, transient errors can have a much more serious impact and lead to not only corrupt data, but also 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 closer to the source task will most likely have higher criticalities. The criticality of the tasks in Fig. 1, for example, can be ranked as $\{C_A=6, C_B=3, C_C=3, C_D=3, C_E=2, C_F=1\}$.

In this work, the improvement in 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 checker task are needed, as shown in Fig. 3 when duplicating task B in Fig. 1.

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

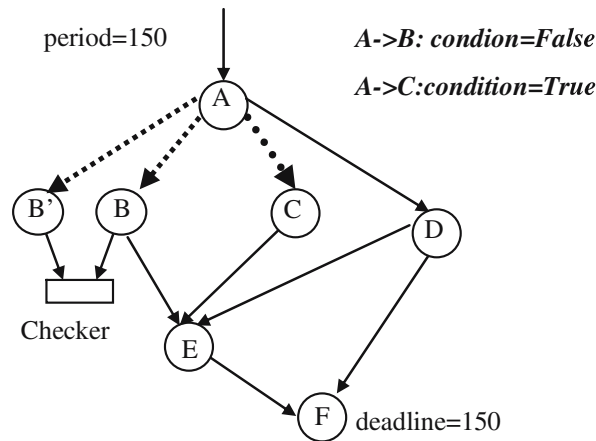


Figure 3. Task duplication.

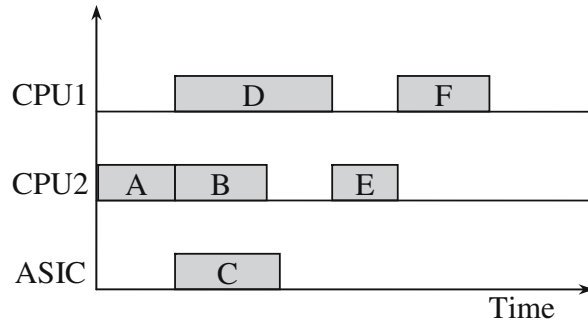


Figure 4. Scheduling without reliability consideration.

power consumption. As an example, consider the task graph in Fig. 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 Fig. 4. However, if task B is a critical task (i.e., needs to be protected against transient errors) and the reliability issue needs to be taken into account, then we may have to replicate task B (denoted B'), and another task (denoted as “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 in this case could be as illustrated in Fig. 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 mutually exclusive and will not be activated at the same time, since they belong to different conditional branches. Furthermore, there are some idle time slots for CPU1

and CPU2 that enable duplicated tasks to be filled without incurring performance and cost penalty. By taking advantage of these facts, a better synthesis result is shown in Fig. 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 are not affected.

4.2. Allocation and Scheduling Algorithm

Our allocation and scheduling algorithm is based on the co-synthesis framework ASICosyn [11]. 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 mutually exclusive tasks that belong to different branches. Figure 7 outlines the proposed allocation and scheduling procedure (ASP).

A *static urgency* 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 [11]. For the branch fork task, the longest branch path is used to

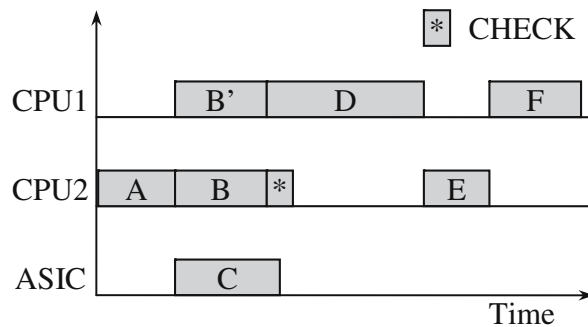


Figure 5. Reliability-aware scheduling. A duplicate of task B (B') and a checker task (*) are inserted into the schedule shown in Fig. 4, resulting in a degraded performance.

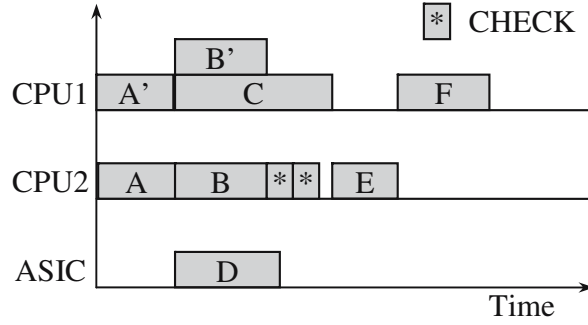


Figure 6. Reliability-aware scheduling, which takes advantage of the mutual exclusiveness of tasks B and C, and the idle time slot at CPU 1, without performance or cost penalty.

calculate the static urgency. On the other hand, a *dynamic urgency* is defined as:

$$DU(\text{task}, \text{CPU}) = SU(\text{task}) - \max \left\{ \begin{array}{l} \text{ready_time}(\text{task}), \\ \text{CPU available time} \end{array} \right\} - WCET(\text{task}, \text{CPU})$$

The dynamic urgency is actually related to the following factors:

1. 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.
2. 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.

3. The worst case execution time (WCET) of this task on the CPU.

4.3. Inserting Duplicated Tasks

In this section, we present two algorithms for inserting duplicated tasks such that the overall reliability of the system is improved. The problem of fitting duplicated tasks into empty scheduling slots is similar to the on-line bin-packing problem [24], which is an NP-hard problem. Therefore, in this work, we explore fast solutions by introducing two heuristic algorithms.

The first algorithm (S_R_INC) is an incremental scheduling approach (see Fig. 8). We first call the original allocation and scheduling procedure ASP (shown in Fig. 7), to obtain 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 again 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

```

ASP (task_graph, target_library)
1. for each task, calculate static_urgency(task)
2. if there is task i in ready list is partitioned on ASIC
3.     schedule task i; goto 9
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
8.             dynamic_urgency value
9. update ready task list and goto 2 until all tasks are
    scheduled
    
```

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

Algorithm: Schedule_with_Reliability_Inc (S_R_INC)
 Call ASP to get the task schedule graph without duplicating tasks
 Record the cost as MIN_COST
 For each task with criticality ranking from high to low
 do {
 Duplicate this task and build new task graph
 Call ASP to update allocation and schedule
 If cost > MIN_COST, delete the duplicated task
 }
 Return the allocation and schedule

Figure 8. Incremental scheduling algorithm.

exceeded because of duplicating this task. Therefore, we should not duplicate this task. The procedure is iteratively executed for all tasks.

The second scheduling algorithm (S_R_STA) is shown in Fig. 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). An idle time slot that are long enough to schedule a duplicated task without affecting other tasks is called *long-time-slot*; An 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) is called *short-time-slot*. The second step involves finding the most

suitable time slot for the target task based on the following heuristic rules:

1. *Long-time-slot* is better than *short-time-slot*
2. among all *long-time-slots*
 - a) the shorter execution time of task on that PE is better
 - b) the earlier time slot is better
3. among all *short-time-slots*
 - a) 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.
 - b) the one that 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, since it generates a new architecture with new allocation and scheduling whenever a duplicated task is inserted into the task graph,. The second algorithm (static scheduling) only calls the allocation

Algorithm: Schedule_with_Reliability_Static (S_R_STA)
 Call ASP to get the task schedule graph without duplicating tasks
 For each task with criticality from high to low
 do {
 1. Search all candidate idle time slot in the whole task schedule graph
 2. Select the best time slot for the task
 3. If time slot available, insert duplicated task and adjust task schedule
 }
 Return the allocation and schedule

Figure 9. Static scheduling algorithm.

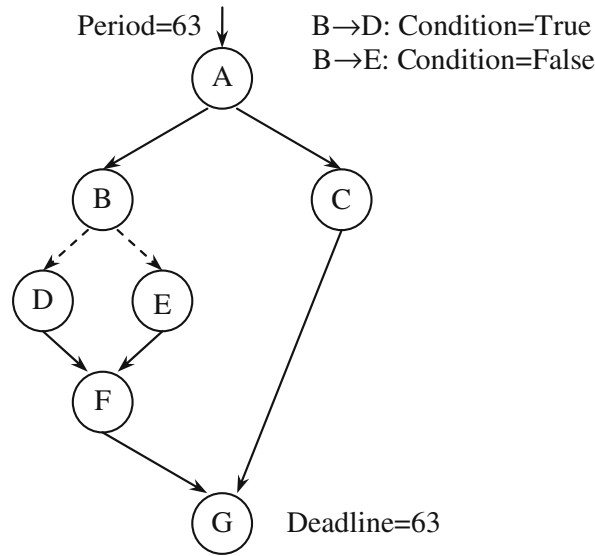


Figure 10. A simple example.

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 results at the expense of longer solution time.

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 to demonstrate how our algorithm work in practice. The input to the co-synthesis framework is the task graph shown in Fig. 10, and a technology library shown in Table 2. The technology library specifies the execution time for each task on different process elements and the cost of processing elements. Note that a CPU can be shared by tasks; but, an 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 would not be the same because of their differences in implementations.

Figure 11a 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 are duplicated. Figure 11b 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 maintaining the same system cost and performance requirement. Due to the comparison that needs to perform between the outcomes of tasks, an extra checker task is also needed for each task duplicated (usually the execution time of this checker task is small, since its only functionality is

Table 2. Technology library.

Task	CPU1	CPU2	ASIC
A	10	18	7
B	30	50	12
C	25	14	9
D	11	19	8
E	14	23	9
F	12	20	15
G	28	15	9

Cost: CPU1=50, CPU2=40, ASIC=100

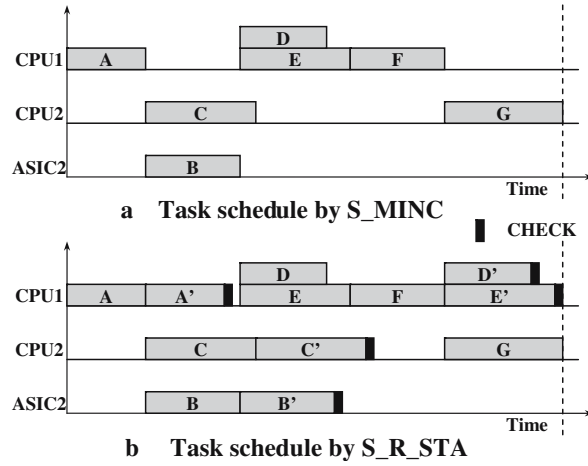


Figure 11. Task schedule by a S_MINC, b S_R_STA.

to compare the result of two tasks. In this example, for simplicity, the execution time of a checker task is assumed to be one).

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 now present results of our experiments with several benchmarks. The benchmark task graphs are either generated by the TGFF tools [6] or real examples extracted from previous literature (MPEG1 and MPEG2) [18]. Some statistics of these task graphs are listed in the first four columns of Table 3. The execution time (in seconds) of different schemes are shown in the last three columns of Table 3. We notice that the S_R_STA has almost the same performance as the original algorithm, S_MINC. This means that the static post-analysis scheme

Table 3. Characteristic of benchmarks and execution time.

	Tasks	Edges	Deadline	MINCtime	STAtime	INCtime
s01	14	16	625	0.01	0.01	0.09
s02	17	20	700	0.01	0.01	0.18
s03	22	28	625	0.01	0.01	0.63
s04	25	32	750	0.01	0.02	1.18
s05	28	32	1200	0.01	0.01	1.51
s06	31	38	600	0.03	0.03	3.52
s07	32	34	900	0.02	0.02	4.08
s08	38	53	1000	0.05	0.05	6.06
s09	39	46	800	0.09	0.09	9.28
s10	43	52	1000	0.13	0.14	21.30
s11	50	60	1000	0.26	0.26	34.38
mpeg1	15	21	40000	0.01	0.01	0.07
mpeg2	50	74	30000	0.15	0.16	20.46

Table 4. Duplicated tasks for S_R_STA and S_R_INC schemes.

	Tasks	STA_Dup	Percent	INC_Dup	Percent
S01	14	6	42.9	6	42.9
S02	17	6	35.3	6	35.3
S03	22	6	27.3	7	31.8
S04	25	6	24.0	8	32.0
S05	28	20	71.4	20	71.4
S06	31	10	32.3	10	32.3
S07	32	24	75.0	24	75.0
S08	38	10	26.3	10	26.3
S09	39	8	20.5	8	20.5
S10	43	13	30.2	13	30.2
S11	50	12	24.0	13	26.0
mpeg1	15	4	26.7	4	26.7
mpeg2	50	19	38.0	19	38.0
average			36.5		37.6

introduces very small runtime overhead. On the other hand, the execution time of S_R_INC is significantly longer than that of 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 Table 4. S_R_STA schemes duplicates an average 37.2% of the tasks and S_R_INC duplicates 38.5% of the tasks on 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 as the S_R_STA scheme.

Our basic schedule algorithms, S_R_INC and S_R_STA, only insert one copy of each original task into the new task graph. If the execution results are

Table 5. Single duplication vs. double duplication.

	Tasks	Single duplication	Percent	Double duplication	Percent
s01	14	6	42.9	2	14.3
s02	17	6	35.3	3	17.6
s03	22	7	31.8	3	13.6
s04	25	8	32.0	3	12.0
s05	28	20	71.4	8	28.6
s06	31	10	32.3	3	9.7
s07	32	24	75.0	8	25.0
s08	38	10	26.3	5	13.2
s09	39	8	20.5	4	10.3
s10	43	13	30.2	6	14.0
s11	50	13	26.0	8	16.0
mpeg1	15	4	26.7	2	13.3
mpeg2	50	19	38.0	11	22.0
Average			37.6		16.1

different between original task and duplicated task, an error can be detected but can not be corrected directly. Therefore, we extend the original error-detect-based algorithm to error-correct-based algorithm. The basic idea is, for each time, two copies of the original task are inserted into the task graph together. When the results from these three tasks are different, majority voting [15] is used to decide the correct value (this is also called Triple Module Redundancy (TMR) [15]).

Table 5 shows the number of original tasks duplicated by two versions of S_R_INC schedule algorithms, single duplication vs. double duplication. Under the same cost limitation, the number of task duplicated in double duplication algorithm is less than that using single duplication. The average number of duplicated tasks is 37.6 and 16.1% in single duplication and double duplication algorithm, respectively.

6. Conclusion

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

Reference

1. BCC Research, <http://www.bccresearch.com/editors/RG-229R.html>.
2. Todd M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," in *32nd Annual International Symposium on Microarchitecture (MICRO)*, November 1999.
3. C. Bolchini, L. Pomante, F. Salice, and D. Sciuto, "Reliability Properties Assessment at System Level: a Co-design Framework," Online Testing Workshop, 2001.
4. C. Bolchini, L. Pomante, F. Salice, and D. Sciuto, "A System Level Approach in Design Dual-Duplex Fault Tolerant Embedded Systems," Online Testing Workshop, 2002.
5. B. Dave and N. K. Jha, "COFTA: Hardware-Software co-synthesis of Heterogeneous Distributed Embedded Systems for Low Overhead Fault Tolerance," *IEEE Trans. Comput.*, vol. 48, 1999.
6. R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: Task Graphs for Free," in *Proc. Int. Workshop Hardware/Software Co-design*, 1998, pp. 97–101.
7. A. Orailoglu and R. Karri, "A Design Methodology for the High-level Synthesis of Fault-tolerant Asics," *VLSI Signal Proc. V*, 1992, pp. 417–426.
8. D. Mavis and P. Eaton, "Soft Error Rate Mitigation Techniques for Modern Microcircuits," in *Proc. Reliable Physics Symposium*, 2002, pp. 216–225.
9. K. Mohanram and N. Touba, "Cost-Effective Approach for Reducing Soft Error Failure Rate in Logic Circuits," *ITC*, 2003.
10. Shubhendu S. Mukherjee, Mike Kontz, and Steven K. Reinhardt, "Detailed Design and Implementation of Redundant Multithreading Alternatives," in *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, 2002.
11. Y. Xie and W. Wolf, "ASICosyn: Co-synthesis of Conditional Task Graphs with Custom ASICs," in *Proceedings of the International Conference on ASICs*, 2001, pp. 130–135.
12. Actel Corporation, "Effects of Neutrons on Programmable Logic," White Paper, 2002.
13. Wayne Wolf and Jorgen Staunstrup, *Hardware/Software Co-design Principles and Practice*, Kluwer, 1997.
14. J. F. Ziegler et al. "IBM Experiments in Soft Fails in Computer Electronics (1978–1994)," *IBM J. Res. Develop.*, vol. 40, no. 1, 1996, pp. 3–18.
15. B. Johnson, *Design and Analysis of Fault-tolerant Digital Systems*, Addison-Wesley, 1989.
16. H. Hollander, B. Carlson, and T. Bennett, "Synthesis of SEU-Tolerant ASICs Using Concurrent Error Correction," in *Proceedings of IEEE Great Lake VLSI Symposium*, 1995, pp. 90–93.
17. R. Karri and A. Orailoglu, "Time-constrained Scheduling during High-level Synthesis of Fault-secure," *VLSI Digital Signal Processors, IEEE Transaction on Reliability*, vol. 45, no. 3, 1996, pp. 404–413.
18. Yanbing L'i, and Jörg Henkel, "A Framework for Estimation and Minimizing Energy Dissipation of Embedded HW/SW Systems," in *Proceedings of DAC*, 1998, pp. 188–193.
19. C. Constantinescu, "Trends and Challenges in VLSI Circuits Reliability," *IEEE Micro*, July–August, 2003.
20. S. Tosun, O. Ozturk, N. Mansouri, E. Arvas, M. Kandemir, and Y. Xie, "An ILP Formulation for Reliability-oriented High-level Synthesis," in *Proceedings of the Sixth International Symposium on Quality Electronic Design (ISQED 2005)*, San Jose, CA, 22, 2005.
21. G. Micheli, R. Ernst, and W. Wolf, *Readings in Hardware/Software Co-design*, Morgan Kaufmann, 2002.
22. W. Wolf, "A Decade of Hardware/Software Co-design," *IEEE Computer*, 2003, pp. 38–43.
23. R. P. Dick, "Multiobjective Synthesis of Low-power Real-time Distributed Embedded Systems," Ph.D. thesis, Princeton University, 2002.
24. D. Coppersmith and P. Raghavan, "Multidimensional Online Bin Packing: Algorithms and Worst Case Analysis," *Oper. Res. Lett.*, vol. 8, 1989, pp. 17–20.



Yuan Xie is Assistant Professor of Computer Science Engineering Department at The Pennsylvania State University. Before joining Penn State in Fall 2003, he was with IBM Microelectronic Division's Worldwide Design Center. He received the B.S. degree in Electronic Engineering from Tsinghua University, Beijing, China, in 1997, and received the M.S. and Ph.D. degrees in Electrical Engineering from Princeton University in 1999 and 2002, respectively. He was a recipient of the SRC Inventor Recognition Award in 2002 and the NSF CAREER award in 2006. His research interests include VLSI Design, computer architecture, embedded systems design, and electronics design automation. He is a member of IEEE and ACM.



Mahmut Kandemir is an Associate Professor in the Computer Science and Engineering Department at the Pennsylvania State University. His main research interests are optimizing compilers, I/O intensive applications, and power-aware computing. He received the B.Sc. and M.Sc. degrees in Control and Computer Engineering from Istanbul Technical University, Istanbul, Turkey. He received his Ph.D. degrees in Electrical Engineering and Computer Science from Syracuse University, Syracuse, New York in 1999. He is a member of the IEEE and the ACM.



Lin Li received the B.S. and M.S. degrees in Computer Science from Peking University, Beijing, China, in 1997 and 2000, respectively, and the Ph.D. degree in Computer Science and Engineering from the Pennsylvania State University, University Park, in 2005. Then, he joined Intel Digital Enterprise Group in Hillsboro, Oregon, and worked on computer architecture, performance modeling and analysis, and low power system design.



Vijaykrishnan Narayanan received his B.E degree in Computer Science and Engineering from University of Madras, Chennai, India in 1993 and his Ph.D. degree in Computer Science and Engineering from University of South Florida, Tampa, Florida, USA in 1998. Since 1998, he has been with the Computer Science and Engineering Department at the Pennsylvania State University where he is currently an Associate Professor. His research interests are in the areas of energy-aware reliable systems, embedded systems, nano/VLSI systems and computer architecture.

Reliability-aware Co-synthesis for Embedded Systems



Mary Jane Irwin is an Evan Pugh Professor and the A. Robert Noll of Engineering in the Department of Computer Science and Engineering at Pennsylvania State University. She received her M.S. (1975) and Ph.D. (1977) degrees in Computer Science from the University of Illinois, Urbana-Champaign. Her research and teaching interests include computer architecture (power constrained, application specific) and computer arithmetic, reliable systems design, and VLSI systems design and design automation. Irwin received

an Honorary Doctorate from Chalmers University, Sweden in 1997. She is an IEEE Fellow and ACM Fellow, and was elected to the National Academy of Engineering in 2003. Irwin is currently serving as the co-Chair of ACM's Publications Board and a member of the NRC's Board of Army Science and Technology. In the past she has served as an elected member of the IEEE Computer Society's Board of Governors, as Vice President of ACM's council, and one of Board of Directors of the Computing Research Association. She was the Editor-in-Chief of ACM's Transactions on the Design Automation of Electronic Systems from 1998 to 2004 and co-Editor-in-Chief of ACM's Journal of Emerging Technologies in Computing Systems from 2005 to 2006. She was the General Chair of the 1996 Federated Computing Research Conference, the 36th Design Automation Conference, the 2002 International Symposium on Low Power Electronics and Design, and the 2004 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems.