

Reliability-Centric Hardware/Software Co-design

S. Tosun^{*}, N. Mansouri^{*}, E. Arvas^{*}, M. Kandemir^{**}, Y. Xie^{**}, and W-L. Hung^{**}

^{*}Syracuse University ^{**}Pennsylvania State University

^{*}{stosun,namansou,earvas}@ecs.syr.edu

^{**}{kandemir,yuanxie,whung}@cse.psu.edu

Abstract

This paper proposes a reliability-centric hardware/software co-design framework. This framework operates with a component library that provides multiple alternates for a given task, each of which is potentially different from the others in terms of reliability, performance, and area metrics. The paper also presents an experimental evaluation of the proposed co-design framework using several example designs and a comparison to a conventional co-design method that does not consider reliability. Our experimental evaluation demonstrates that the proposed framework can be used to study the tradeoffs between area, performance, and reliability, and that it is important to include reliability as a first class parameter in optimization.

1. Introduction

Over the years, embedded systems have evolved from simple architectures to complex SoC (System-on-a-Chip) designs that include both programmable components and ASICs. An important issue that comes with this complexity is the "design problem". Complexity of the structure of embedded designs and exponential increase in data sizes they manipulate make it imperative to adopt an automated design methodology, instead of relying purely on designer's experience and intuition.

Hardware-software co-design has been a promising approach to address the design problem of complex embedded systems, by employing a strategy that automatically decides the hardware configuration to be employed and scheduling of the tasks that constitute the design. A design/scheduling determined by a hardware-software co-design framework can then be further refined for obtaining the final design. Most of the prior efforts on the co-design area focused on optimizing performance (latency) under area constraints or optimizing area under performance constraints. More recent work has also considered power-aware co-design by including power as one of the parameters to be optimized. There exist only a few co-design studies that consider reliability or fault tolerance as a metric to optimize [1][2][3][4].

Soft errors are fast becoming a critical problem haunting electronic equipment manufacturers [5]. They originate from the cosmic rays from space or from tiny traces of radioactive elements that occur in all materials. While the particles themselves are not dangerous, they can potentially disrupt the operation of silicon chips. They are particularly problematic in safety-critical embedded systems, where an error in control memory or logic can be catastrophic in terms of human and/or equipment loss. Therefore, it is extremely important to reduce the vulnerability of complex embedded systems to soft errors. Our belief is that a reliability-aware co-design framework can be a cure to soft error related problems since, through such a framework, the reliability measures against soft errors can be incorporated to the system early in the design process. The

work presented in this paper is a step in this direction. However, it is not limited to only soft error related reliability characterization since it can be extended to other types of transient errors, whose impact can be characterized in a component library.

Reliability optimization can be targeted at the different levels of abstraction in the design hierarchy. However, more significant gains can be achieved when it is targeted at the higher levels. The reliability of the designs can also be improved at the lower levels. However, it is more costly and one cannot achieve more reliable designs compared to the designs that include reliability as a first class metric at the system level. Motivated with these observations, we include the reliability metric at the system level through the hardware/software co-design paradigm.

This paper makes the following contributions: First, it presents a reliability-centric co-design framework. This framework operates with a component library that provides multiple alternatives for a given task, each of which is potentially different from the others in terms of reliability, performance, and area metrics. Starting with the most reliable design (i.e., using the most reliable alternate for each task), our co-design approach gradually lowers the reliability of the design until the specified performance and area bounds are met. In other words, its goal is to determine the most reliable design under the given area and performance bounds. The second contribution of this paper is an experimental evaluation of the proposed co-design framework using several example designs and a comparison to a conventional co-design method that does not consider reliability. Our experimental evaluation demonstrates that the proposed framework can be used to study the tradeoffs between area, performance, and reliability, and that it is important to consider reliability as a first class parameter in optimization.

The rest of this paper is structured as follows. The next section briefly discusses the related work. Section 3 introduces our representation, target architecture, and design reliability metric. Section 4 presents the details of our approach, and Section 5 reports results from our experimental evaluation. Finally, Section 6 concludes the paper with a summary.

2. Related work

Early work on hardware/software co-design focused on a variety of issues including partitioning [6], performance-area tradeoffs [7][8], and power estimation [9]. Besides these heuristic methods, there have been studies aiming to determine the optimal solution to the problem using well-known techniques such as genetic algorithms [10] and integer linear programming [11]. However, only a few prior studies focused on the reliability, fault-detection, or fault-tolerance in the context of hardware/software co-design [1][2][3][4]. COFTA [1] presents a hardware/software co-synthesis framework for fault-tolerant heterogeneous real-time distributed embedded systems. Its fault detection capability is based on adding assertion and duplicate-and-compare tasks,

which introduces overheads to the design area. Bolchini et al. [2][3] add the reliability concern into the system specification using an elegant approach. However, their target architecture includes only one processor for software parts and one coprocessor unit for hardware implementations. Vargas et al. [4] present an approach for partitioning the specification into hardware and software parts that are described using C and Handle-C languages. They apply a weak mutation analysis technique, which was originally proposed for software testing, to verify the system reliability after the synthesis process is complete.

Our method differs from the prior hardware/software co-design systems mentioned above by treating reliability as a first class metric. We make use of different reliability characterized versions of each task to improve the overall system reliability. To the best of our knowledge, this is the first hardware/software co-design methodology that partitions and schedules a design specification using different hardware and software components, each differing from the rest from reliability, area, and performance (latency) perspectives.

3. Design representation and design reliability

3.1. Representation and architecture

Hardware/software co-design partitions the specification of an embedded system into hardware and software parts based on the desired area/performance constraints. Task graph is one of the commonly used models to describe the system specification [12]. A task graph is a directed-acyclic graph where vertices represent the tasks and edges represent the dependencies among these tasks. An example task graph is given in Figure 1(a). The deadline is the maximum allowed time that a task should finish its execution. While each task in the graph may have its own deadline, in this work we use a single deadline for the entire task graph.

Figure 1(b) illustrates the target architecture we employ in this work. This architecture consists of three main units; namely CPUs, ASICs, and memory component. The CPUs are heterogeneous processors, that is, each processor may have different architecture, size, and speed (frequency) than the others. They execute software implementations of tasks. ASICs are the hardware implementations of tasks, and we have multiple ASIC implementations for each task. The memory component is shared by CPUs and ASICs, and it stores data and instructions of the tasks that run on CPUs and the controller code for the tasks that are implemented as ASICs. In this study, we add one unit memory area to the overall design area for each task that is implemented in software to be executed on a CPU. We neglect the area of the controller on the memory component since it is very small

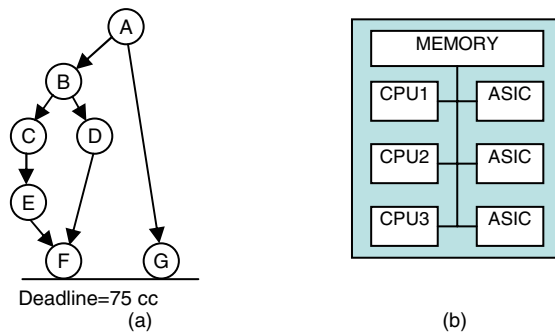


Figure 1. (a) An example task graph, (b) Target architecture.

compared to software implementations. The communication between tasks is asynchronous, which means a task starts its execution only when it receives a completion signal and its input data from all of its predecessors. We assume the communication time between tasks is one unit unless both the tasks are mapped on the same CPU.

3.2. Technology library

A technology library provides multiple processing elements (PEs) for each task, which have different area and performance characteristic and can be either ASIC or CPU. In our reliability-centric hardware/software co-design framework, we augment this library with a reliability metric for each task that can be implemented on a PE. Reliability is the probability with which a component or a system will succeed (i.e., execute in an error-free manner) for a period of time $[t_0, t_s]$, given that it has been working correctly at the time t_0 . As the previous work [13] showed, there exist various ASIC implementations with different reliability, area, and performance values. Software implementations of different tasks on various CPUs may also have different reliabilities. These values for each task running on different CPUs can be characterized using software simulation techniques [14]. A sample technology library for the task graph given in Figure 1(a) is shown in Table 1. One can observe from this table that, the reliability value of a task does not only depend on the PE on which the task is implemented; it is also a function of the software implementation used. For example, the reliability of one task may be high on a CPU, while another task's reliability may be low on the same CPU. Throughout the discussion of our reliability-centric hardware/software co-design framework, we will use the task graph given in Figure 1(a) and the technology library shown in Table 1. It should be emphasized however that our framework is very general and can work with any technology library that can contain different number of CPUs and ASICs. Note that ASIC1 and ASIC2 are the two different hardware implementations of a task (i.e., each task on Table 1 has two different ASIC implementations; namely ASIC1 and ASIC2).

3.3. Overall design reliability

After obtaining a final design within the specified area and performance bounds, we calculate the overall design reliability (R_s) by taking the products of the reliabilities of each task (R_i). In mathematical terms, we have:

$$R_s = \prod_{i=1}^n R_i$$

Table 1. Reliability, area, and performance values for each task in Figure 1(a) on different PEs.

		Tasks						
		A	B	C	D	E	F	G
CPU1	Relib.	0.985	0.978	0.996	0.991	0.987	0.983	0.956
	Area	40	40	40	40	40	40	40
	Delay	30	45	25	40	50	60	35
CPU2	Relib.	0.989	0.990	0.987	0.973	0.985	0.987	0.973
	Area	60	60	60	60	60	60	60
	Delay	25	30	35	20	55	55	30
ASIC1	Relib.	0.998	0.987	0.993	0.971	0.980	0.991	0.971
	Area	10	12	15	20	15	15	28
	Delay	8	10	10	12	8	9	7
ASIC2	Relib.	0.976	0.969	0.989	0.979	0.978	0.988	0.995
	Area	15	10	12	15	20	12	20
	Delay	6	12	12	15	6	12	9

where n is the number of tasks in the graph. Note that our task graphs do not contain any branching and iteration constructs.

The goal of our reliability-centric hardware/software co-design framework is to allocate the tasks to PEs, and schedule the task graph such that the reliability of the final design (R_s) is maximized while the specified area and performance constraints are met.

4. Allocation and scheduling algorithm

Our co-design methodology uses an iterative strategy to find the most reliable solution possible within the specified area and performance constraints by using the provided technology library. The proposed approach consists of three main steps:

1. Obtaining an initial solution by allocating the most reliable PEs from the technology library to each task.
2. Optimizing performance of the initial solution if we do not meet the performance constraint.
3. Optimizing area of the initial solution if we do not meet the area constraint.

We explain each of these steps in the following subsections in detail.

4.1. Initial Solution

Our algorithm for the initial solution is given in Figure 2. We construct the initial solution by assigning the most reliable PEs to each task. Consequently, we schedule the task graph based on this assignment. The scheduling algorithm we use is similar to the algorithms used in high-level synthesis [15]. First, we find the earliest start time (EST) and the latest start time (LST) of each task on the task graph. The EST of task i is found based on the finish times of task i 's immediate predecessors. This can be expressed as follows:

$$EST_i = \max\{EST_j + d_j + ct(j, i)\}; \forall i, j: (t_j, t_i) \in E.$$

In this expression, d_j is the delay of task j and $ct(j, i)$ is the communication time (delay) between task i and task j . The communication time between two tasks scheduled on the same CPU is assumed to be zero. (t_j, t_i) represents the dependency (the edge) between tasks i and j ; i.e., task j is the immediate predecessor of task i . The EST s of tasks that do not have any predecessors are assumed to be one. This step also returns the initial latency (IL) of the schedule, which can be found by following expression:

$$IL = \max\{EST_j + d_j\}; \forall j.$$

Note that if the given performance is higher than initial

```

Initial_solution(T(V,E),PE,A,L)
{
1.   Allocate the most reliable PEs to each task;
2.   IL=Find_EST(T);
3.   If (L>IL) {
4.     IL=L;
5.   }endif
6.   Find_LST(T,IL);
7.   Schedule tasks one by one with the highest priority first
   until all tasks are scheduled;
8.   return IL;
}

```

Figure 2. The algorithm for the initial schedule. A and L correspond to the area and latency (performance) bounds, respectively. $T(V,E)$ is the task graph (V and E are the vertex and the edge sets, respectively), and PE is the set of processing elements.

Table 2. EST , LST , and freedom (F) of each task in Figure 1(a) with the most reliable PEs from Table 1.

	A	B	C	D	E	F	G
EST	1	10	41	41	66	117	10
LST	5	14	45	80	70	121	121
F	4	4	4	39	4	6	111

latency, we use performance bound (deadline) to find the LST s of tasks.

The LST of each task can be determined by reverse application of the method used to find the EST s. That is, we can use the following expression:

$$LST_j = \min\{LST_i - d_i - ct(i, j)\}; \forall i, j: (t_j, t_i) \in E.$$

The LST s of the tasks that do not have any successors are found using the following expression:

$$LST_j = IL - d_j.$$

After having found the EST s and LST s for our tasks, we calculate their freedom (F). The freedom of a task can be defined as the time frame during which the task can be scheduled, which is the difference between the EST and LST of the task. The freedoms of the tasks that are on the critical path are zero, which means the start times of the tasks that are on the critical path are fixed. We determine the freedom of each task i using the following expression:

$$F_i = LST_i - EST_i.$$

The EST , LST , and freedom of the tasks of Figure 1(a) after assigning the most reliable PEs to them are given in Table 2. We then start scheduling the tasks one by one based on their priorities. The priority of a task is determined based on its freedom. A task with the smaller freedom gets a higher priority over a task with the larger freedom. Note that the initial latency we determine may increase when we schedule the tasks on CPUs since some CPUs may be highly loaded. In this case, the previously scheduled tasks on a CPU force their successors, which are also assigned to the same CPU, to be scheduled outside their best time frame, resulting in performance degradation.

In Figure 3, we show the initial schedule for the task graph in Figure 1(a). Let us assume that, for this task graph, we bound ourselves with a latency of 130 time units and 140 area units of total area. The area bound is found by applying the approach in [8] to the task graph, which will be explained in Section 5. As can be seen from Figure 3, our initial schedule does not meet these performance and area constraints. Consequently, we employ performance and area optimization methods, which will be explained in the following subsections, to meet the bounds. The initial schedule given in Figure 3 has an overall design reliability of 0.949101, which is the most reliable design possible with the provided technology library. In this schedule, CPU1 is heavily loaded (tasks C, E, and D are scheduled on this CPU). As a result, task D is scheduled after task E (task E is scheduled before task D due to its higher priority), even though it can be scheduled right after task B, if one considers only the task graph.

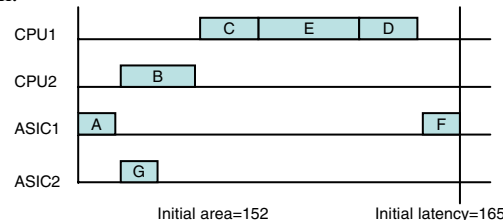


Figure 3. The initial schedule for the task graph in Figure 1(a) with the most reliable PEs.

4.2. Performance Optimization

The initial schedule we obtain is the most reliable schedule among all candidate schedules. However, we may exceed the performance and/or area constraints with this schedule (as in the case of Figure 3). Consequently, we select the faster PEs for some tasks or reallocate some of the tasks from the heavily-loaded CPUs to other PEs in an attempt to reduce the overall latency of the design (i.e., to bring it to the specified bound). However, it is to be noted that, doing so reduces the overall design reliability as well. We propose two optimization strategies in order to meet the specified deadline. These two strategies are embedded into the algorithm sketched in Figure 4.

In the first strategy, we check the tasks that are assigned to CPUs. As we mentioned earlier, the tasks that are assigned to the same CPU may increase the overall latency if their schedulable time frames overlap. Thus, we reassign the tasks that may increase the overall latency to alternate PEs (either CPU or ASIC), which have the highest reliability among the candidate PEs. If we still do not meet the performance bound, we employ our second strategy. In this strategy, we pick the slowest task on the critical path of the design, and assign it to a faster PE, which is the most reliable PE among the candidates. Note that this new assignment may increase the initial latency, even though we select a faster PE. This is because the candidate PE itself may be a heavily-loaded CPU and the assigned task may force the other tasks to delay. If this happens, we search for other candidate PEs from the technology library. This step is iterated until we meet the specified performance deadline. If we select the fastest PEs for each task on the critical path and still do not meet the specified performance bound, our approach returns no solution since it is not possible to obtain a schedule with the given technology library and performance bound.

In Figure 5, we illustrate the modified schedule for our running example after the performance optimization. As one can observe from this schedule, task D is assigned to ASIC2 to meet the performance constraint, which decreases the overall design reliability from 0.949101 to 0.937601. Note that task D is assigned to ASIC2 even though there are other possibilities. This is because the reliability of task D on ASIC2 is the highest compared to other alternate PEs.

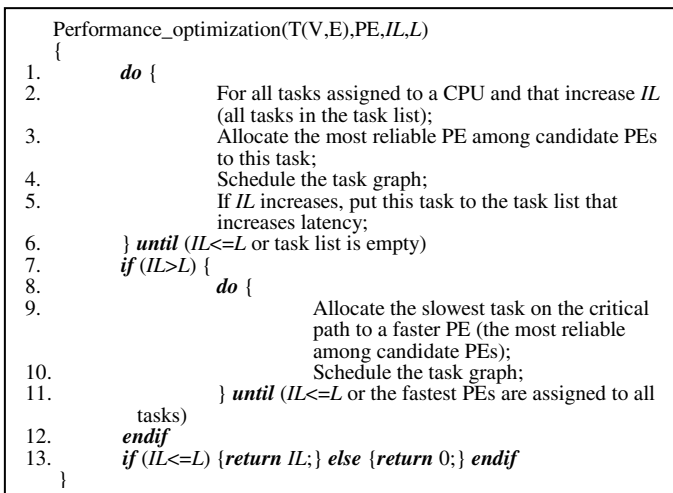


Figure 4. The algorithm for reducing latency. *IL* is the latency of the current schedule.

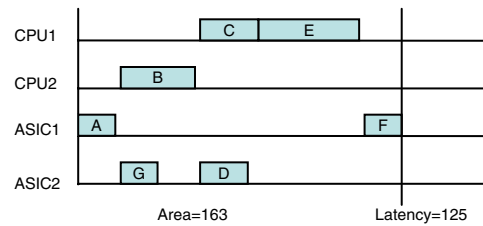


Figure 5. Scheduling of Figure 1(a) after the performance optimization.

4.3. Area Optimization

If we do not meet the area bound, we employ two optimization methods, which are embedded into the algorithm given in Figure 6.

In the first method, we move the tasks from ASICs to CPUs to reduce the overall area of the design. We attempt to reassign as many ASIC tasks as possible to CPUs. We select the tasks to be reassigned based on their priorities (i.e., the task that has the smallest freedom gets the highest priority). Note that a task that is on the critical path (i.e., a task that has zero freedom) cannot be moved to a CPU since the execution time of a task implemented as software will be typically higher than its ASIC implementation, and this will increase the overall design latency. The first candidate CPU we attempt to allocate a task will be the most reliable one among all the available candidates. After the configuration is modified, we invoke our scheduling algorithm to check if this allocation increases the latency. If it does, we assign this task to a different CPU considering the reliability values. This process continues iteratively until we schedule the selected task on a CPU. Note that if we exceed the latency bound as a result of moving a task from ASIC to CPU, we do not modify the current allocation, and we search for an alternate victim task until we move all possible tasks to CPUs. If we still cannot meet the area bound, we use our second method. In the second method, we check if the tasks that are assigned to the fast ASICs can be moved to the slower ASICs. If there are tasks that can be slowed down, we allocate these tasks to the slower ASICs to reduce the overall area requirement. If the total area of the design is still larger than the area bound, we return the schedule found with the minimum area as our final solution. That is, in our approach, if we cannot satisfy the area bound, we return the solution with an area, which is the closest to the specified bound.

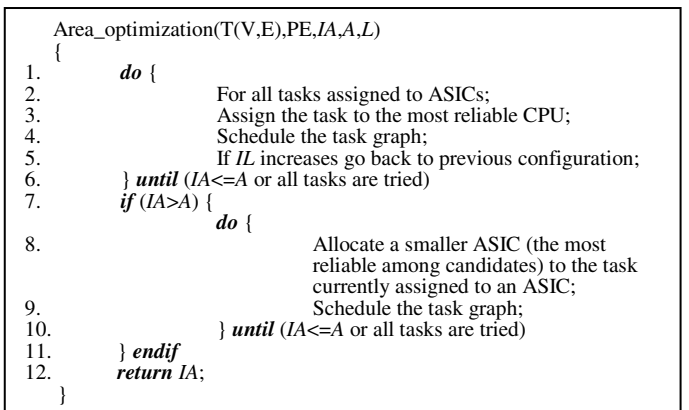


Figure 6. The algorithm for reducing area. *IA* corresponds to the initial area (the total area) of the current schedule.

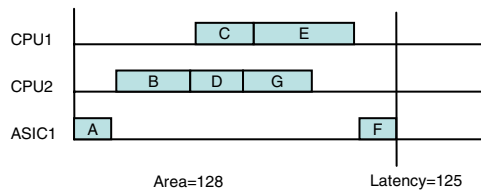


Figure 7. Scheduling of Figure 1(a) after performance and area optimizations.

The schedule in Figure 5 after the area optimization is transformed to the one shown in Figure 7. In order to meet the area bound, we move the tasks D and G from ASIC2 to CPU2, which results in a total area of 128 units. Because of this new configuration, the overall reliability of the design reduces to 0.911258. It is to be observed that, the schedule in Figure 7 satisfies both the latency and area constraints specified.

5. Experimental Results

In this section, we quantify the impact of our reliability-centric approach by scheduling several task graphs. Specifically, we compare our approach with a classical co-design strategy that focuses only on the area and performance tradeoffs, such as the method proposed in [8]. This method initially finds the fastest schedule to meet the performance constraint, and then it iteratively reduces the area of the design by moving tasks to CPUs since many of the tasks in the initial schedule would be mapped on ASICs (alternate approaches are also possible).

In our first example, we demonstrate the impact of our approach and the approach in [8] on the overall system reliability as well as on the area and performance of the design. We generate our task graphs by using the TFFG tool [16]. An example task graph generated by this tool is shown in Figure 8 (named s00). For the target architecture, we use two CPUs and several ASIC implementations. For each task, we limit ourselves to three different hardware implementations and we choose the most appropriate one that meets the specified bounds. The technology library we use to schedule s00 is given in Table 3. In this table, we provide the area, performance, and reliability values for each task when it is scheduled on different PEs. The reliability values for ASICs are determined based on the change in area and performance. The reliability of an ASIC implementation is proportional to its area and performance [13], i.e., when the area or latency of the design increases, its reliability also increases.

The area bound for our tool is the minimum design area found by the approach in [8], which is expected to be the minimum area possible under the given performance bound.

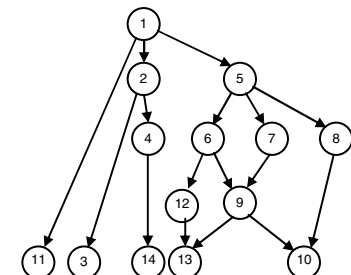


Figure 8. An example task graph.

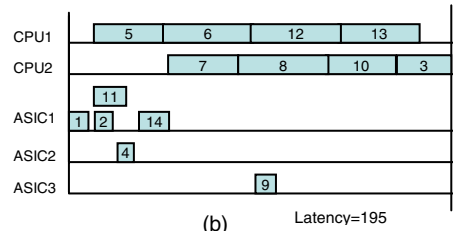
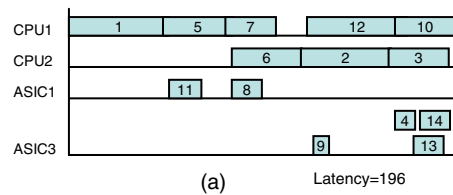


Figure 9. Two schedules for Figure 8 with 200 time units of performance and 153 units of area bounds (Schedule (a) focuses only on area-performance while schedule (b) focuses on reliability-area-performance tradeoff).

The schedule for Figure 8 without any reliability concern is shown in Figure 9(a). The minimum area of this design is 157 units under 200 time units of performance bound. Figure 9(b) illustrates the schedule determined by our method, resulting in a slight area increase. Our method finds a design with a total area of 163 units. The important point is that, our method brings a 8.76% reliability improvement over the schedule in Figure 9(a). Specifically, while the reliability of the first schedule is 0.822971, the reliability of our schedule is 0.895089.

In Table 4, we provide experimental results for various task graphs. We run our experiments on four different task graphs. Due to space concerns, we give only one technology library (Table 3), which is for the task graph s00. For each task graph given in Table 4, we first find the minimum area without reliability concern for different performance bounds (using the approach in [8]). Then, we apply our method to the graph with the given constraints. While the first two columns of Table 4 give the name of the graph and the number of tasks, the next two columns give the area and performance bounds for both the scheduling schemes we compare. The columns five and six show, respectively, the latency and design reliability of the schedule obtained by [8]. Our area, latency, and reliability values are listed in columns seven, eight, and nine, respectively. In the last three columns, we give the area, performance, and reliability improvements brought by our approach over the one in [8]. The last column of the table gives the reliability improvement brought by our approach over [8]. As one can see from the last three columns, our approach comes up with more reliable designs with a slight increase in area. Note that the values given in columns ten and eleven are with respect to the corresponding values returned by the approach in [8]. Specifically, our approach brings average 15.28% reliability improvements over the approach in [8] while sacrificing 3.98% from area on the average.

6. Concluding Remarks

Hardware-software co-design has been a promising approach to address the design problem of complex embedded systems, by employing a strategy that automatically decides the hardware configuration to be employed and scheduling of the tasks that constitute the design. While the emerging soft error problem should be

addressed in all software and hardware layers during design, our belief is that a reliability-aware co-design framework can be a cure to soft error related problems as reliability measures against soft errors can be incorporated to the design early in the process. This paper presents such a framework, experimentally evaluates it, and compares it to a conventional co-design framework. The proposed framework can be used to investigate the tradeoffs between area, performance, and reliability.

7. References

[1] B. Dave and N. K. Jha, "COFTA: Hardware-software co-synthesis of heterogeneous distributed embedded systems for low overhead fault tolerance," *IEEE Trans. on Computers*, vol. 48, Apr. 1999.

[2] C. Bolchini, L. Pomante, F. Salice, and D. Sciuto, "Reliability Properties Assessment at System Level: a Co-design Framework", *Online Testing Workshop*, 2001.

[3] 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.

[4] F. Vargas, E. Bezerra, L. Wulff, and D. Barros Jr., "Optimizing HW/SW Co-design towards Reliability for Critical-Application Systems", *Asian Test Symposium*, 1998.

[5] J. F. Ziegler et al, "IBM experiments in soft fails in computer electronics (1978-1994)", *IBM Journal of Research and Development*, 1996.

[6] A. Kalavade and E. A. Lee, "The extended partitioning problem: hardware/software mapping and implementation-bin selection",

Proceedings of the Sixth IEEE International Workshop on Rapid System Prototyping, 1995

[7] W. H. Wolf, "An Architectural Co-Synthesis Algorithm for Distributed, Embedded Computing Systems", *IEEE Trans. on VLSI*, 5(2):218-229, June 1997.

[8] Y. Xie and W. Wolf, "Co-synthesis with custom ASICs", *Proceedings of the 2000 conference on Asia South Pacific Design Automation*, 2000.

[9] W. Fornaciari, P. Gubian, D. Sciuto, and C. Silvano, "Power estimation of embedded systems: a hardware/software co-design approach", *IEEE Trans. on VLSI*, 1998.

[10] R. Dick and N. K. Jha, "MOGAC: A multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems", *Proceedings of ICCAD*, pp. 522-529, Nov. 1997.

[11] S. Prakash and A. Parker, "SOS: Synthesis of application specific heterogeneous multiprocessor systems", *Proceedings of the 34th ACM/IEEE DAC*, pp. 703-708, June 1992.

[12] W. Wolf and J. Staunstrup, *Hardware/Software Co-design Principles and Practice*, Kluwer Academic Publisher, 1997.

[13] A. Orailoglu and R. Karri, "A Design Methodology For The High-level Synthesis Of Fault-tolerant Asics", *VLSI Signal Processing V*, 1992.

[14] R. Huang, M. R. Lyu, and K. Kanoun, "Simulation Techniques for Component-Based Software Reliability Modeling with Project Application," in *Proceedings of International Symposium on Information Systems and Engineering*, June 2001.

[15] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[16] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: task graphs for free", *Proceedings of the 6th international workshop on Hardware/software co-design*, 1998.

Table 3. Technology library for the task graph given in Figure 8 (s00 in Table 4).

		1	2	3	4	5	6	7	8	9	10	11	12	13	14
CPU1	Relib.	0.985	0.985	0.986	0.987	0.980	0.995	0.995	0.993	0.972	0.989	0.985	0.995	0.995	0.990
	Area	40	40	40	40	40	40	40	40	40	40	40	40	40	40
	Delay	50	40	35	45	35	50	25	40	35	30	50	50	45	50
CPU2	Relib.	0.997	0.993	0.980	0.977	0.975	0.976	0.999	0.995	0.980	0.975	0.990	0.992	0.990	0.980
	Area	30	30	30	30	30	30	30	30	30	30	30	30	30	30
	Delay	55	45	30	65	55	35	35	50	65	35	30	65	40	65
ASIC1	Relib.	0.998	0.995	0.998	0.992	0.991	0.994	0.979	0.970	0.986	0.992	0.998	0.986	0.988	0.993
	Area	15	19	11	16	15	20	17	18	20	16	13	22	20	25
ASIC2	Relib.	0.984	0.985	0.990	0.995	0.985	0.980	0.975	0.984	0.987	0.984	0.984	0.984	0.984	0.987
	Area	13	16	16	14	12	17	15	20	15	12	25	19	18	21
	Delay	10	12	9	8	10	11	12	12	7	17	12	12	12	12
ASIC3	Relib.	0.991	0.978	0.992	0.984	0.995	0.990	0.981	0.992	0.997	0.988	0.993	0.978	0.980	0.985
	Area	17	21	10	13	10	22	13	22	13	14	20	17	16	19
	Delay	7	8	15	10	13	8	15	10	8	15	14	15	15	15

Table 4. Scheduling results of different task graphs for our method and the method without reliability concern.

Task Graph	Number of tasks	Bounds		The approach in [8]		Our method			Improvement (%)		
		Area	Deadline	Latency	Relib.	Area	Latency	Relib.	Area	Latency	Relib.
s00	14	209	125	119	0.813879	212	114	0.902377	-1.43	4.20	10.87
		204	150	150	0.812226	206	149	0.881647	-0.98	0.67	8.55
		187	175	175	0.778319	190	173	0.908789	-1.60	1.14	16.76
		157	200	196	0.822971	163	195	0.895089	-3.82	0.51	8.76
s01	17	252	125	122	0.786538	264	121	0.877468	-4.76	0.82	11.56
		232	175	174	0.81413	239	175	0.916425	-3.02	-0.57	12.56
		204	250	246	0.747634	209	248	0.913648	-2.45	-0.81	22.20
		149	300	294	0.785042	157	280	0.882602	-5.37	4.76	12.43
s02	22	230	250	245	0.730562	238	248	0.843494	-3.48	-1.22	15.46
		198	300	297	0.750714	209	292	0.86517	-5.56	1.68	15.25
		178	350	350	0.726199	187	347	0.864303	-5.06	0.86	19.02
		148	400	391	0.721729	159	383	0.840156	-7.43	2.05	16.41
s03	25	294	250	249	0.707356	305	248	0.866171	-3.74	0.40	22.45
		241	300	298	0.703067	251	297	0.850556	-4.15	0.34	20.98
		199	350	346	0.747968	209	344	0.860978	-5.03	0.58	15.11
		187	400	398	0.715289	198	399	0.830555	-5.88	-0.25	16.11