Hardware/Software Optimization of Error Detection Implementation for Real-Time Embedded Systems

Adrian Lifa Petru Eles Zebo Peng
Linköping University, Linköping, Sweden
{adrian.alin.lifa, petru.eles, zebo.peng}@liu.se

Viacheslav Izosimov
Embedded Intelligent Solutions, Linköping, Sweden
viacheslav.izosimov@eis.semcon.com

ABSTRACT

This paper presents an approach to system-level optimization of error detection implementation in the context of fault-tolerant real-time distributed embedded systems used for safety-critical applications. An application is modeled as a set of processes communicating by messages. Processes are mapped on computation nodes connected to the communication infrastructure. To provide resiliency against transient faults, efficient error detection and recovery techniques have to be employed. Our main focus in this paper is on the efficient implementation of the error detection mechanisms. We have developed techniques to optimize the hardware/software implementation of error detection, in order to minimize the global worst-case schedule length, while meeting the imposed hardware cost constraints and tolerating multiple transient faults. We present two design optimization algorithms which are able to find feasible solutions given a limited amount of resources: the first one assumes that, when implemented in hardware, error detection is deployed on static reconfigurable FPGAs, while the second one considers partial dynamic reconfiguration capabilities of the FPGAs.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance; C.4 [**Performance of Systems**]: Fault tolerance.

General Terms

Reliability, Design, Performance, Algorithms.

1. INTRODUCTION

Safety-critical applications must function correctly even in the presence of faults. Such faults might be transient, intermittent or permanent. Factors like high complexity, smaller transistor sizes, higher operational frequencies and lower voltage levels have contributed to the increase in the rate of transient and intermittent faults in modern electronic systems [5]. From the fault tolerance point of view, transient and intermittent faults manifest themselves very similarly: they have a short duration and then disappear without causing permanent damage. Considering this, we will further refer to both types as transient faults. Permanent faults are not addressed in this paper.

Error detection is crucial for meeting the required reliability of the system. Unfortunately, it is also a major source of time overhead. In order to reduce this overhead, one possible approach is to implement the error detection mechanisms in hardware, which, however, increases the overall cost of the system. Because error detection incurs high overheads, optimizing it early in the design phase of a system can result in a big gain, and often makes the difference between a feasible solution and an unfeasible one.

Previous work mainly focused on optimizing different fault tolerance techniques, or integrating fault tolerance concerns into scheduling algorithms, while considering error detection as a black box [10, 11, 21]. Various techniques for error detection have been proposed, both software and hardware-based [3, 4, 9, 18].

The main contribution of this paper is an approach to the optimization of error detection implementation (EDI) in the context of fault-tolerant embedded systems. We propose two optimization algorithms: one considering that, when implemented in hardware, error detection is deployed on static reconfigurable FPGAs and the other one assuming 1-dimensional (1D) or 2-dimensional (2D) partial dynamic reconfiguration (PDR) capabilities of the FPGAs. To our knowledge, this is the first approach considering the optimization of error detection in a system-level design context.

2. PRELIMINARIES

2.1 Error Detection Technique

In recent years, the concept of application-aware reliability [15] has been introduced as an alternative to the traditional one-size-fits-all approach. Application-aware techniques make use of the knowledge about the application's characteristics. As a result, customized solutions are created, tuned to better suit each application's needs. As shown in [15] and [14] the main idea of this application-aware technique is to identify, based on specific metrics [16], *critical variables* in a program. A critical variable is defined as "a program variable that exhibits high sensitivity to random data errors in the application" [15]. Then the backward program slice for each acyclic control path is extracted for the identified critical variables. The backward program slice is defined as "the set of all program statements/instructions that can affect the value of the variable at a program location" [15]. Next, each slice is aggressively optimized at compile time, resulting in a series of checking expressions. These will be inserted in the original code, before the use of a critical variable. Finally, the original program is instrumented with instructions to keep track of the control paths followed at runtime and with checking instructions that would choose the corresponding checking expression, and then compare the results obtained.

Let us present an illustrative example of how the above error detection technique works. We use a simplified example of an if-then-else statement in Figure 1 (adapted from [15]). The original program code is presented on the left (no shading), the checking code added by the technique is presented on the right (light shading) and the path tracking instrumentation is shown with dark shading. We assume that x was identified as a critical variable and, thus, it needs to be checked before its use.

We can identify two paths in the program slice of x, corresponding to the two branches. The instructions on each path are optimized, resulting in a concise expression that checks the correctness of the variable's value along that path. For the first path, the expression reduces to $x' = w$, while for the second one, the checking expression reduces to $x' = s - 2t$ (the values are assigned to the temporary variable x'). At runtime, when control reaches a point that uses variable x, one of the two checking expressions is chosen based on

CODES/ISSS'10, October 24–29, 2010, Scottsdale, Arizona, USA.

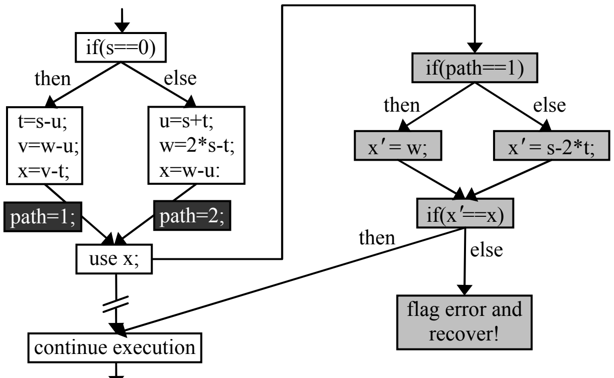


Figure 1. Code fragment with detectors

the value of the path variable (updated via the instrumentation code added). Then, the value of x (computed by the original program) is compared with the value of x' (recomputed by the checking expression). In case of a mismatch, an error flag is raised and a recovery action should be taken.

The above technique has two main sources of performance overhead: path tracking and variable checking. In the context of transient faults, both of them can be implemented either in software, potentially incurring high performance overheads, or in hardware, which can lead to costs sometimes exceeding the amount of resources.

Pattabiraman et al. have proposed a software-only, straightforward approach, in which both the path tracking and the variable checking are implemented in software and executed together with the application. In this case, the path tracking alone incurs a time overhead of up to 400%, while the overhead due to variable checking is up to 80% [15]. Complete hardware implementations of path tracking and expression checking are proposed in [15] and [14]. In fact, between the extreme solutions of implementing all error detection in software, on the one side, and performing it in hardware, on the other side, there is a wide range of possible alternatives characterized by the particular implementation decision taken for each process in the application. This decision depends on various factors such as time criticality, the amount and cost of available hardware resources and their nature, such as FPGAs with static or partial dynamic reconfiguration. The focus of this paper is on efficiently implementing error detection in the context mentioned above.

The error detection technique described above detects any transient errors that result in corruption of the architectural state (e.g. instruction fetch and decode errors, execute and memory unit errors and cache/memory/register file errors) provided that they corrupt one or more variables in the backward slice of a critical variable. In order to achieve maximal error coverage, we assume that some complementary, generic error detection techniques are used in conjunction with the application-aware one presented above (like a watchdog processor and/or error-correcting codes in memory, for example). Some hardware redundancy techniques might also be used to deal with the remaining, not covered, faults [4]. In this paper we concentrate on the optimization of the application-aware error detection component.

2.2 Synthesis of Fault-Tolerant Schedules

In order to provide resiliency against transient faults one possible fault tolerance technique to use is re-execution. In such a context, schedule synthesis should account for the possible process re-executions in case of faults [11].

The authors of [10] proposed an approach to the generation of fault-tolerant schedules such that multiple transient faults are tolerated in the context of hard real-time systems. The algorithm takes as an input an application modeled as a process graph, the worst-case execution time (WCET) of processes, the worst-case transmission time (WCTT) of messages, as well as the error detection and recovery overheads for each process, the architecture on which this application is mapped and the maximum number of faults that could affect the system during one period. As an output it produces schedule tables that capture the alternative execution scenarios corresponding to possible fault occurrences.

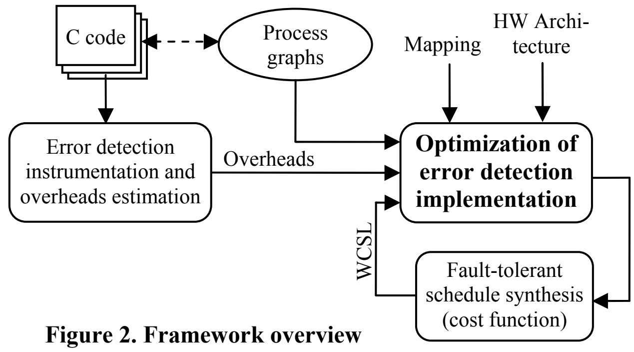


Figure 2. Framework overview

Among all fault scenarios there exists one which corresponds to the worst-case in terms of schedule length. In the rest of the paper, we are interested in this worst-case schedule length (WCSL), which has to satisfy the imposed application deadline.

In this context, our fault model assumes that a maximum number k of transient faults can affect the system during one period. To provide resiliency against these faults re-execution is used. Once a fault is detected by the error detection technique, the initial state of the process is restored and the process is re-executed.

The above mentioned scheduling technique considers error detection as a black box. In this paper, we will try to minimize the WCSL of the application, by accelerating error detection in reconfigurable hardware in an intelligent manner, so that we meet the time and cost constraints imposed to our system.

2.3 Optimization Framework

In Figure 2 we present an overview of our framework. The initial applications, available as C code, are represented as a set of process graphs. The code is processed through the error detection instrumentation framework [14]. This framework outputs the initial application code with the embedded error detectors, as well as the VHDL code needed to synthesize error detector modules on FPGA. The instrumented code is used to estimate the time overheads and hardware costs implied by different implementations of the error detection technique.

This information, together with the system architecture and the mapping of processes to computation nodes, is used by the optimization tool, which tries to find a close to optimal error detection implementation. The cost function for optimization is represented by the WCSL generated by the fault-tolerant schedule synthesis tool [10] (see Section 2.2). Our goal is to minimize the WCSL of the application, while also meeting the HW cost constraints.

3. SYSTEM MODEL

We consider a set of real-time applications A_i, modeled as acyclic directed graphs $G_i(V_i, E_i)$, executed with period T_i. The graphs G_i are merged into a single graph $G(V, E)$, having the period T equal with the least common multiple of all T_i. This graph corresponds to a virtual application A. Each vertex $P_j \in V$ represents a process, and each edge $e_{jk} \in E$, from P_j to P_k, indicates that the output of P_j is an input for P_k. Processes are non-preemptable and all data dependencies have to be satisfied before a process can start executing. We consider a global deadline D, representing the time interval during which the application A has to finish.

The application runs on a distributed architecture composed of a set of computation nodes, connected to a bus (Figure 3c and e). The processes are mapped to these nodes and the mapping is given (illustrated with shading in Figure 3a, b and d). The bus is assumed

Table 1. WCET and overheads

Proc.	$WCET^U$	Error Detection Implementation								
		SW-only			Mixed HW/SW			HW-only		
		$WCET_i$	h_i	ρ_i	$WCET_i$	h_i	ρ_i	$WCET_i$	h_i	ρ_i
P_1	60	240	0	0	100	15	20	80	40	45
P_2	50	140	0	0	80	15	20	60	40	45
P_3	40	150	0	0	60	10	15	50	30	35
P_4	30	100	0	0	60	15	20	40	40	45

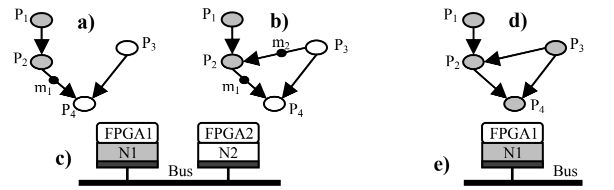


Figure 3. System model

to be fault-tolerant (i.e. we use a communication protocol such as TTP [12]). Each node is composed of a central processing unit, a communication controller, a memory subsystem, and also includes a reconfigurable device (FPGA). Knowing that SRAM-based FPGAs are susceptible to single event upsets [23], we assume that suitable mitigation techniques are employed (e.g. [13]) in order to provide sufficient reliability of the hardware used for error detection.

For each process we consider three alternative implementations of error detection (EDIs): SW-only, mixed HW/SW and HW-only. For the SW-only alternative, the checking code (illustrated with light shading in Figure 1) and the path tracking instrumentation (illustrated with dark shading in Figure 1) are implemented in software and interleaved with the actual code of the application. Since the time overhead of path tracking is significant, a natural refinement of the technique is to place the path tracking instrumentation in hardware, and, thus, drastically reduce its overhead. This second alternative represents the mixed HW/SW solution, in which the path tracking is moved to hardware and done concurrently with the execution of the application, while the checking expressions remain in software, interleaved with the initial code. In order to further reduce the time overhead, the execution of the checking expressions can also be moved to hardware (referred as the HW-only implementation).

We assume that for each process its worst-case execution time ($WCET_i$) [22] is known, for each of the three possible implementations of error detection (SW-only, mixed HW/SW and HW-only). Also, the corresponding HW cost/area (h_i) and the reconfiguration time (ρ_i) needed to implement error detection are known.

For all the messages sent over the bus (between processes mapped on different computation nodes), their worst-case transmission time (WCTT) is given. Such a transmission is modeled as a communication process inserted on the edge connecting the sender and the receiver process (Figure 3a and b). For processes mapped on the same node, the communication time is considered to be part of the process' WCET and is not modeled explicitly.

4. MOTIVATIONAL EXAMPLES

In Figure 3a we present an application A, modeled as a process graph, with four processes: P_1 to P_4. We have an architecture with two computation nodes: N_1 and N_2, connected by a bus (Figure 3c). Processes P_1 and P_2 are mapped on N_1, while P_3 and P_4 are mapped on N_2. The WCET of processes for each of the three alternative implementations of error detection are listed in Table 1. The WCET of the un-instrumented process ($WCET^U$) is also given in Table 1, only to emphasize the error detection time overheads, which can be calculated, for each of the three alternatives, by subtracting the WCET of the un-instrumented process from the WCET of the process instrumented with a particular implementation of error detection. For example, the error detection overhead incurred by the SW-only EDI for process P_1 is 240 - 60 = 180 time units. The HW costs (h_i) and the reconfiguration times (ρ_i) incurred by each alternative EDI, for each process, are also presented in Table 1. The WCTT of messages is considered to be 20. The recovery overhead for all processes (see Section 2.2) is 10. Our application has to tolerate a number of k = 1 faults, within its execution period. We represent process execution with white boxes, recovery overheads with dark shading and FPGA reconfiguration overheads with a checkerboard pattern.

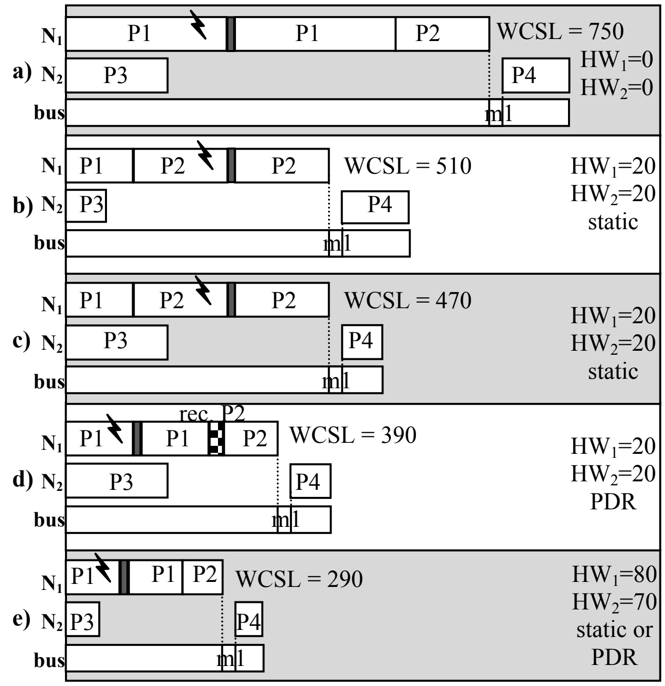


Figure 4. Motivational example 1

In Figure 4a we present the worst-case execution scenario for the SW-only solution. This means that we implement error detection in software for all processes and then we generate fault-tolerant schedules as described in Section 2.2. In this case, we obtain a worst-case schedule length (WCSL) of 750 time units, corresponding to the scenario in which P_1 experiences a fault, but we do not use any additional hardware. Considering that we have no cost constraint (i.e. unlimited reconfigurable hardware), the shortest possible WCSL is obtained by using the HW-only solution (Figure 4e): 290 time units. This means that we use the HW-only implementation of error detection for all processes. In this case, the sizes of $FPGA_1$ and $FPGA_2$ should be at least 80 and 70 area units, respectively. These are the two extreme cases corresponding to the longest WCSL, but no additional hardware cost, and to the shortest WCSL, with maximal hardware cost. In our approach we will try to obtain the minimal WCSL, subject to the available hardware area.

Considering that we have static reconfigurable FPGAs of size 20 on each node, we can afford to place only the mixed HW/SW error detection module for only one process per node into hardware. The question is how should we choose the modules to place on FPGA? If we do not consider the characteristics of the application, a naive approach would be to place, for each of the two computation nodes, the error detection module corresponding to the biggest process on FPGA, and thus reduce its total WCET. This case is shown in Figure 4b: by placing error detection for P_1 and P_3 into hardware, we have reduced the WCET from 240 to 100 for P_1 and from 150 to 60 for P_3. Thus, we obtain a WCSL of 510 time units. Nevertheless, in node N_2 it is actually better to place error detection for process P_4

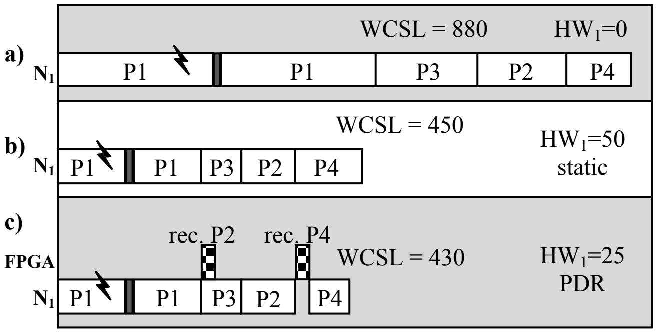


Figure 5. Motivational example 2

into HW (Figure 4c). Even though this means we only shorten its WCET with 40 time units (compared to 90 in the case of P_3), we finally obtain a shorter WCSL, of only 470 time units. Compared to the SW-only solution, we got an improvement of 37%. Because of the slack following P_3, shortening its WCET does not impact the end-to-end delay of our application, and the FPGA can be used more efficiently with P_4. In general, assigning different error detection implementations to processes is very much dependent on the actual application, and a good strategy should take into account the specific characteristics of processes and their interdependencies.

Let us now assume that the FPGAs of size 20 we use have partial dynamic reconfiguration (PDR) capabilities (this means that parts of the device may be reconfigured at runtime, while other parts remain functional). In Figure 4d we present the shortest WCSL we could obtain in this case: 390 time units. We initially place the mixed HW/SW EDI for P_1 on the FPGA, and then, after P_1 finishes execution, using partial dynamic reconfiguration, we place the mixed HW/SW error detection module for P_2, reusing the FPGA area. In Figure 4c it was impossible to reuse the FPGA area, since we assumed only static reconfiguration. Comparing the shortest WCSL from Figure 4d (i.e. 390 time units) with the minimal WCSL we got using static FPGAs of the same size (i.e. 470 time units), we can see that by exploiting PDR we were able to improve the WCSL even further, shortening it with an extra 17%. Remember that we used FPGAs with a size of only a quarter of the maximum needed to implement the HW-only solution.

Let us now consider the application from Figure 3d mapped on only one computation node (Figure 3e). In Figure 5a we present the SW-only solution, with a WCSL of 880 time units. Assuming that we have an FPGA of size 50, without PDR capabilities, the shortest worst-case schedule we can get is illustrated in Figure 5b (450 time units). We manage to obtain this by assigning the mixed HW/SW EDI for P_1, P_2 and P_3. Let us now consider an FPGA of only 25 area units, but having PDR capabilities. In this case (Figure 5c), we can initially place the mixed HW/SW implementations for processes P_1 and P_3 on the FPGA. Then, as soon as P_1 finishes, we can reuse the FPGA area corresponding to its detector module and reconfigure in advance the mixed HW/SW EDI for P_2. This reconfiguration is done in parallel with the execution of P_3, so all the reconfiguration overhead can be masked. As a consequence, P_2 can be scheduled as soon as P_3 finishes. Unfortunately, for P_4 we cannot reconfigure in parallel with P_2's execution, since we only have 10 area units available. So, we are forced to wait until P_2 ends, then reconfigure the FPGA with P_4's mixed HW/SW detector module and only after that schedule P_4. Note that, even if the reconfiguration time for P_4 could not be masked, we still prefer this solution compared to running the SW-only alternative of P_4, because we gain 20 time units = $WCET_{SW\text{-}only}$ - ($\rho_{mixed\ HW/SW}$ + $WCET_{mixed\ HW/SW}$). Comparing Figure 5c (WCSL = 430 time units) with Figure 5b (WCSL = 450 time units), we see that, by exploiting PDR capabilities, we can get even better performance than using static FPGAs of double the size. Note that the improvement relative to the SW-only solution (Figure 5a: WCSL = 880 time units) is 51%.

5. PROBLEM FORMULATION

As an input we have an application A modeled as a process graph $G(V, E)$ (see Section 3). The WCETs of each alternative error detection implementation (EDI), for each process, are given by a function $W : V \times H \rightarrow \mathbf{Z}^+$, where H is the ordered set of available error detection implementations (i.e. H = *{SW_only, mixed_HW/SW, HW_only}*). Similarly, the hardware costs implied by each error detection implementation, for each process, are given by a function $C : V \times H \rightarrow \mathbf{Z}^+ \times \mathbf{Z}^+$, i.e. we know the size (rows × columns) of the rectangle needed on FPGA. Note that function C covers both the 1D and 2D reconfiguration scenarios, since for the 1D case the number of rows for all detector modules is 1. For each message m_{ij}, sent from process P_i to process P_j, its worst-case transmission time ($WCTT_{ij}$) on the bus is known.

The application is implemented on a system consisting of a set of computation nodes N connected by a bus B. For each node $N_j \in N$, the available hardware area ($HW_j = Rows_j \times Columns_j$) which we can use to implement the error detection is known. The mapping of processes to computation nodes is given by a function $M : V \rightarrow N$.

The parameter k, which denotes the number of transient faults to be tolerated during one period of execution, T, is given. k is used for the generation of fault-tolerant schedules (see Section 2.2).

We are interested in finding an error detection implementation assignment $S : V \rightarrow H$, such that the k transient faults are tolerated and the worst-case schedule length (WCSL) is minimal, while the hardware cost constraints are met.

Depending on the nature of the available hardware resources (i.e. FPGAs with static reconfiguration or with PDR capabilities), in the following we propose two solutions to the above problem.

6. EDI WITH STATIC CONFIGURATION

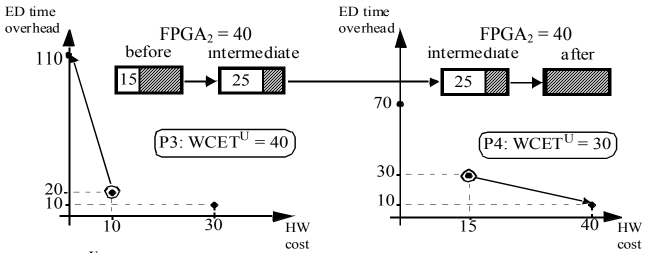
The problem defined in Section 5 is a combined mapping and scheduling problem, which is NP-complete [8]. Thus, while for small problem sizes, we can solve it by, for example, exhaustive search, for bigger problem sizes finding an exact solution is unfeasible. Therefore, the solution proposed in this paper is based on a Tabu Search heuristic [17].

Figure 6 presents the pseudocode for our EDI assignment optimization algorithm. The input consists of the merged application graph G, the architecture N, the mapping M, the WCETs of each process, for each EDI, as well as the EDI cost overheads and the number k of faults to be tolerated within one system period. The algorithm outputs an assignment S of EDIs to processes, so that the WCSL is minimized and the HW cost constraints are met.

The exploration of the solution space starts from a random initial solution (line 1). In the following, based on a neighborhood search, successive moves are performed with the goal to come as close as possible to the solution with the shortest WCSL. The transition from one solution to another is the result of the selection (line 5) and

```
EDI_Optimization(G, N, M, W, C, k)
1   best_Sol = current_Sol = Random_Initial_Solution();
2   best_WCSL = current_WCSL = WCSL(current_Sol);
3   Tabu = Ø;
4   while (iteration_count < max_iterations) {
5       best_Move = Select_Best_Move(current_Sol, current_WCSL);
6       Tabu = Tabu U {best_Move};
7       current_Sol = Apply(best_Move, current_Sol);
8       current_WCSL = WCSL(current_Sol); Update(best_Sol);
9       if (no_improvement_count > diversification_count)
10          Restart_Diversification();
11  }
12  return best_Sol;
end EDI_Optimization
```

Figure 6. EDI optimization algorithm



WCET^U indicates the WCET of the un-instrumented process. The WCET of P_3, e.g. for the SW-only EDI, will be 40 + 110 = 150.

Figure 7. Swap move

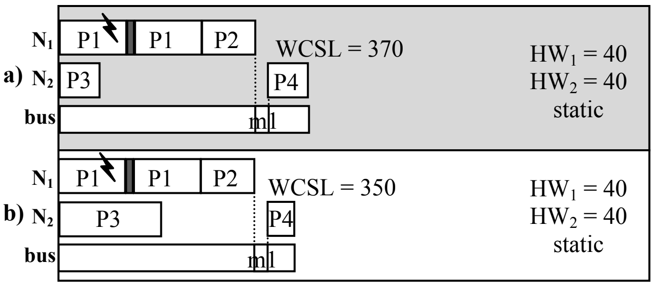


Figure 8. Schedules before and after swap move

application (line 7) of an appropriate move. At each iteration, in order to evaluate our cost function (WCSL), the processes and messages are scheduled using the fault-tolerant scheduling technique (function WCSL in Figure 6) presented in [10] (see Section 2.2). To assure the proper breadth of the search process, diversification is employed (lines 9-10). The whole search process is based on a recency memory (Tabu list) and a frequency memory (Wait counters). We will describe these aspects in the following paragraphs.

6.1 Moves

We have considered two types of moves: *simple* ones and *swaps*. A simple move applied to a process P_i is defined as the transition from one error detection implementation to any of the adjacent ones from the ordered set H = {*SW_only, mixed_HW/SW, HW_only*}. Intuitively, EDI of a process is moved more into hardware (for example, from the SW-only alternative to the mixed HW/SW EDI, or from the mixed HW/SW to the HW-only EDI), or more into software (for example from the HW-only to the mixed HW/SW EDI, or from the mixed HW/SW to the SW-only EDI), but direct transitions between SW-only and HW-only EDIs are not allowed. The motivation behind restricting transitions only to adjacent EDIs was to limit the size of our neighborhood (defined as the set of solutions that can be directly reached from the current one, by applying a single move). A swap consists of two "opposite" simple moves, concerning two processes mapped on the same computation node. The idea is that, in order to move the EDI of a process more into hardware, in the case we would not have enough resources, it is first needed to move the EDI of another process mapped on the same computation node, more into software, to make room on the FPGA device. The advantage of performing a swap is that, if possible, we manage to find a more efficient use of the currently occupied HW resources.

In Figure 7 we consider the case of processes P_3 and P_4 from the motivational example 1 (Figure 3a and c), which are mapped on the same computation node (N_2). The two processes have the HW cost – time overhead trade-off points shown in Figure 7 (see also Table 1). The point on the vertical axis represents the SW-only EDI, the middle point represents the mixed HW/SW EDI and the third point illustrates the HW-only EDI, for a particular process. At a certain step during the design space exploration, P_3 and P_4 both have mixed HW/SW EDI assigned, which implies a total $FPGA_2$ area of 10 + 15 = 25 units. The worst-case schedule in this case is illustrated in Figure 8a (note that also P_1 and P_2 have their mixed HW/SW EDI assigned). Assuming that $FPGA_2$ has a total size of 40 area units, we have 15 units free in the above scenario. In order to be able to move P_4 to the HW-only EDI, we need 25 extra units of area. Since we only have 15 units available, the solution is to move P_3 to its SW-only EDI, thus freeing 10 extra area units. After this simple move, we can apply the second simple move, occupying the 25 available area units, by moving P_4 to the HW-only solution. Please note that the two simple moves mentioned above are performed in the same iteration, thus forming a swap move. The swap in our example had a beneficial impact and we reduced the WCSL from 370 to 350 time units (Figure 8b), thus getting closer to the minimum.

An important feature of Tabu Search is its capability to escape from local minima by allowing the selection of non-improving moves. After selecting such a move, it is important to avoid the cycling caused by selection of the reverse (improving) move leading back to the local optimum. This is solved by the use of tabus. Whenever we perform a simple move we declare tabu the move that would reverse its effect, i.e. assigning P_i its previous EDI (line 6 in Figure 6). The move is forbidden for a number of iterations equal to its *tabu tenure* (determined empirically). When performing a swap move, we declare tabu each of its constituent simple moves (and record them individually in the Tabu list).

6.2 Neighborhood Restriction

In theory, the best move is selected (line 5 in Figure 6) by considering all possible moves (simple or swap) and evaluating the cost function for each one. This, however, is inefficient from the computational point of view. Therefore, in each iteration, the selection of the best move is done by exploring only a subset of the possible moves, namely the ones affecting the processes on the critical path (CP) of the worst-case schedule for the current solution.

In Figure 9 we consider the application from Figure 3b mapped on the architecture in Figure 3c, with static FPGAs of size 20. We show how we move from the SW-only solution (Figure 9a: WCSL = 750) to the solution in Figure 9c (WCSL = 510), passing through two iterations. At each iteration, only the processes on the CP (illustrated with dotted rectangles) are considered for new EDI assignment. From (a) to (b), the best possible choice is to move P_1 to its mixed HW/SW EDI. As a result, the CP changes (see Figure 9b) and, thus, in the next iteration P_3 will also be considered (while P_1 is now excluded). The best choice is to move P_3 to its mixed HW/SW EDI. The result is shown in Figure 9c (WCSL = 510).

6.3 Move Selection

Figure 10 presents our approach to selecting the best move in each iteration (line 5 in Figure 6). As explained earlier, we first determine the set of processes on the critical path (CP) of the current solution (line 1). Next, based on this set, we proceed and search for the best move in a hierarchical manner (in order to reduce the

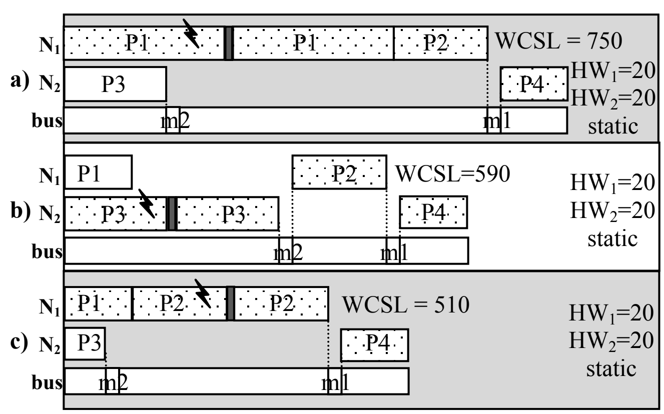


Figure 9. Restricting the neighborhood

number of evaluations of the cost function, done in each iteration). We first explore the simple moves into HW (line 2). If at least one such move exists and it is not tabu, then we select the move that generates the best improvement and we stop further exploring the rest of candidate moves (lines 3-4). Otherwise, we try to improve the current WCSL by searching for the best swap move (line 6). If we get closer to a minimum (line 9), we accept the move (line 10). Otherwise, we diversify the search.

6.4 Diversification

In order to assure the proper breadth of the search process, we decided to employ a *continuous* diversification strategy (lines 12-13 in Figure 10), complemented by a *restart* diversification strategy (lines 9-10 in Figure 6). The continuous diversification is based on an intermediate-term frequency memory (the *Wait* memory), in which we record how many iterations a process has waited so far, since the last time it has been involved in a move. Every time we reach a solution better than all the previous ones, encountered so far, we reset the Wait counters.

Whenever we need to escape local minimum points, the Wait memory is used to selectively filter candidate moves (and generate the set WCP - line 12 in Figure 10). So, if the waiting counter of a process is greater than a threshold (*waiting_count*), we consider that the process has waited a long time and should be selected for diversification (it is included in WCP). Our neighborhood exploration continues by selecting the non-improving move (simple or swap) that leads to the solution with the lowest cost, giving priority to diversifying moves (line 13 in Figure 10).

We complemented our continuous diversification strategy with a restart diversification. Whenever we do not get any improvement of the best known solution for more than a certain number, *diversification_count*, of iterations, we restart the search process (lines 9-10 in Figure 6). The search is restarted from a state corresponding to an EDI assignment to processes that has not been visited, or has been visited rarely, during the previous searches.

The search process is stopped after a specified maximum number of iterations (line 4 in Figure 6). This value (*max_iterations*), as well as the counters used for diversification purposes (*waiting_count* and *diversification_count*), were determined empirically for each application size.

7. OPTIMIZING EDI WITH PDR FPGAS

Latest generations of commercially available FPGA families provide support for partial dynamic reconfiguration (PDR) [24]. This means that parts of the device may be reconfigured at runtime, while other parts remain functional[1]. In the last years, a large amount of research has been carried out with regard to PDR and more details on this subject can be found in, e.g., [6, 19, 20].

PDR enables the possibility of reusing FPGA area that is no longer needed (corresponding to the error detector modules of processes that finished executing). Also, it is possible to overlap process execution with reconfiguration of other error detector modules, in order to mask the reconfiguration time overhead, and thus reduce the latency of the application. Under the PDR assumptions, our problem formulation (Section 5) becomes more complex. Besides generating the EDI assignment to processes ($S : V \rightarrow H$) we also need to generate a placement and a schedule for EDI reconfiguration on FPGA. Formally, this means that, for all processes that have

```
Select_Best_Move(current_Sol, current_WCSL)
1   CP = Select_CP_Processes(current_Sol);
2   trial_Move = Try_Simple_Moves_into_HW(CP);
3   if (trial_Move exists)
4       return trial_Move;
5   else {
6       trial_Move = Try_Swap_Moves(CP);
7       trial_Sol = Apply(trial_Move, current_Sol);
8       trial_WCSL = WCSL(trial_Sol);
9       if (trial_WCSL < current_WCSL)
10          return trial_Move;
11      else {
12          WCP = {p є CP | Wait(p) > waiting_count };
13          trial_Move = Diversifying_Non-improving_Moves(WCP);
14          return trial_Move;
15      }
16  }
end Select_Best_Move
```

Figure 10. Hierarchical neighborhood exploration

error detection implemented in HW (either mixed HW/SW or HW-only), we have to find the function $R : \{p \in V \mid S(p) \neq SW\_only\} \rightarrow \boldsymbol{Z}^+$, which specifies the reconfiguration start time for the EDI module of each process, and the placement function $P : \{p \in V \mid S(p) \neq SW\_only\} \rightarrow \boldsymbol{Z}^+ \times \boldsymbol{Z}^+$, which specifies the position of the upper left corner of each EDI module on the FPGA.

We model our FPGA supporting PDR, as a rectangular matrix of configurable logic blocks (CLBs). Each EDI to be scheduled in this architecture occupies a contiguous rectangular area of this matrix. The model allows choosing a 2D or a 1D placement and reconfiguration scenario. A 1D reconfiguration constraint implies that we can only reconfigure whole columns of CLBs. The 2D scenario allows modules of any rectangular shape and size.

As stated above, the execution of an EDI can proceed in parallel with the reconfiguration of another EDI, but only one reconfiguration may be done at a certain time. However, it is not always possible to completely hide the reconfiguration overhead; in such a case, process execution is scheduled as soon as the reconfiguration of its EDI ends. We have illustrated this in Figure 5c. As we can see, for P_2 it was possible to mask the entire reconfiguration overhead by overlapping it with P_3's execution. Due to the limited resources, this was not possible in the case of P_4. As a consequence, P_4 had to wait for the reconfiguration of its EDI module to finish.

In our approach, we do simultaneous scheduling of processes on the processor and placement of the corresponding EDIs on the FPGA: once a process is selected for scheduling from the list of ready processes, its EDI is placed onto the FPGA, as soon as enough space is available for it. In this way, all the generated schedules are correct by construction, and we cannot end up with an unfeasible schedule (that would violate placement constraints).

In order to be able to take into account the issues presented above, we extended the fault-tolerant schedule synthesis tool described in [10] (that we used in Section 6). This scheduler is based on a list scheduling approach that uses a modified partial critical path (PCP) priority function [7] to decide the order of process execution. However, this priority function does not capture the particular issues related to PDR mentioned above. So, in order to adapt the scheduler according to our needs, we changed the priority function with another one, similar to that proposed by Banerjee et al. in [2]. Key parameters of this function are EST (earliest execution start time of a process), WCET of a process, EDI area, and the partial critical path length. The PCP captures the particular characteristics of the application (like the process interdependencies), the WCET and EDI area characterize each error detection implementation of each process, while the EST captures physical issues related to placement and reconfiguration of EDI modules on the FPGA. Thus, the priority function for our scheduler can be described as:

$$f(EST, WCET, area, PCP) = x \times EST + y \times WCET + z \times area + w \times PCP$$

---

1 As opposed to PDR, for simple dynamic reconfiguration the entire configuration memory has to be configured at a time, and no FPGA computation can proceed in parallel with this process (i.e. reconfiguration overhead cannot be masked). Our algorithm can also be used for such type of FPGAs, since they are a subset of the ones supporting PDR.

When computing the EST, we find the earliest time slot when a process can be scheduled, subject to the various constraints. We first search for the earliest time instant when a feasible EDI placement on FPGA is available. In order to decide the position of an EDI on the FPGA we use the first-fit policy (i.e. we search for the first empty area big enough to accommodate the EDI module). If the reconfiguration controller is available at this time, then the reconfiguration of the EDI can start immediately. Otherwise, it has to wait until the reconfiguration controller becomes free. Once the reconfiguration component (corresponding to the EDI of a process) is scheduled, we check to see if the execution of the process could be scheduled immediately after that, subject to dependency constraints.

Figure 11 illustrates some of the above issues. Let us consider the application from Figure 11a, having to tolerate a number of k = 1 faults, mapped on an architecture with two computation nodes (see Figure 3c), with $FPGA_1 = FPGA_2$ = 25 area units. We assume that at a certain step during the optimization, each process has a particular EDI assigned. The WCET of processes, as well as the EDI area (h_i) and reconfiguration overheads (ρ_i) corresponding to this current EDI are presented in Table 2. WCTT of all messages and the recovery overheads for processes are 10.

Figure 11b shows the corresponding schedule obtained by using only the partial critical path (PCP_i - listed in Table 2) for priority assignment, and ignoring the physical issues related to FPGA placement and reconfiguration of EDIs. In this case WCSL = 525.

Let us now assume that the set of weights (*x, y, z, w*) take values (0.5, -0.5, 0, 1), which implies the following priority function: $f = 0.5 \times EST - 0.5 \times WCET + PCP$. This means that we give higher priority to processes that have a big PCP and EST value (since they have positive weights), while the WCET should be small. The result is that P_2 is scheduled before P_1, which in turn makes it possible for P_3-P_5-P_6 to run earlier, thus reducing the WCSL (which is equal to 475 in Figure 11c).

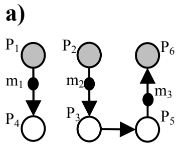
Although from (b) to (c) we reduced the WCSL from 525 to 475, it is possible to choose an even better set of weights, adapted to the characteristics of the application. It is preferable to give higher priority to processes with a big PCP value, but a small WCET and EDI area, and that can start earlier (smaller EST). By setting (*x, y, z, w*) to (-0.5, -0.5, -1, 1) we manage to obtain the schedule illustrated in Figure 11d, with WCSL = 360. In this case, P_5 is scheduled before P_4 and this has two advantages: one is that now P_6 can run earlier, in parallel with P_4, and the second one is that P_3 and P_5 can fit together on the FPGA from the beginning, and thus the runtime reconfiguration for the EDI of P_5 is eliminated. As can be seen from the above example, it is important to take into account the issues related to the FPGA by choosing proper weights for the scheduler.

The weights x, y, z and w are dynamically tuned for each particular application, during our optimization. We kept the same Tabu Search core as the one used for the static reconfiguration approach (see Section 6), with two modifications: (1) we use the modified list scheduler described above, as a cost function in our optimization loop, instead of the scheduler used before (line 8 in Figure 6 and line 8 in Figure 10); (2) we added a new type of possible moves: those that concern the weights x, y, z and w used in the priority function of the scheduler. We extended our search as follows: in each iteration, before exploring different EDI assignments to processes, we explore different values for the weights. We consider that each of them can take a value between -1 and 1, with a step of 0.25. So, at each iteration, we explore if modifying these weights would result in a better priority function, and consequently in a smaller WCSL. If this is not possible, we search for a better EDI assignment, exactly as we did before (Figure 10). Thus, the priority function of the list scheduler is tuned for each particular application.

Table 2. Process characteristics

	P_1	P_2	P_3	P_4	P_5	P_6
$WCET_i$	70	40	30	100	40	50
PCP_i	170	160	120	100	90	50
EDI area (h_i)	20	10	10	25	15	15
rec. time (ρ_i)	25	15	15	30	20	20

a)



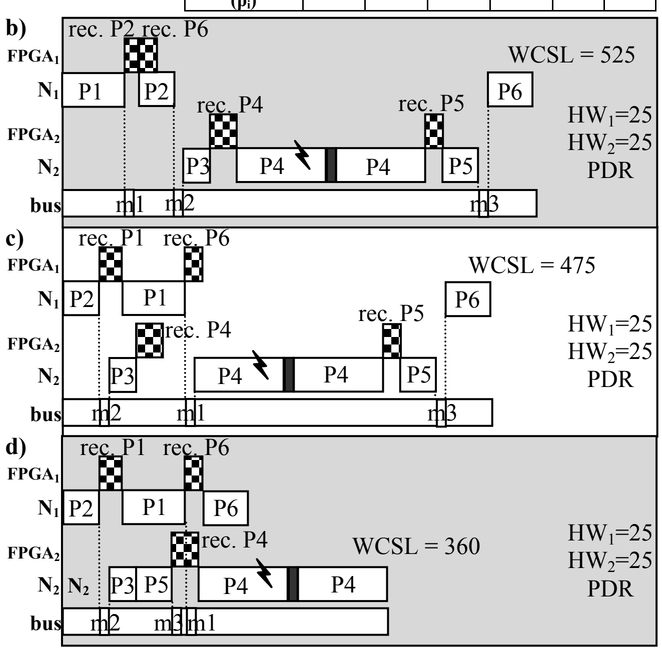


Figure 11. Modified scheduler

8. EXPERIMENTAL RESULTS

We first performed experiments on synthetic examples. We generated process graphs with 20, 40, 60, 80, 100 and 120 processes each, mapped on architectures consisting of 3, 4, 5, 6, 7 and 8 nodes respectively. We generated 15 graphs for each application size, out of which 8 have a random structure and 7 have a tree-like structure. Worst-case execution times for processes were assigned randomly within the 10 to 250 time units range. All messages were assumed to have equal worst-case transmission time. We have considered that our system has to tolerate a number of k = 2 faults[2]. The results are presented below.

In order to generate time and hardware cost overheads for each EDI, we proceeded as follows: we generated one class of experiments (testcase1), based on the estimation of overheads done by Pattabiraman et al. in [15] and by Lyle et al in [14]. We also generated a second class of experiments (testcase2), for which we assumed slower hardware (in other words, in order to get the same time overheads as in testcase1, we need to use more hardware). In Figure 12 we show the ranges used for randomly generating the overheads. The point corresponding to 100% HW cost overhead represents the maximum HW area that the EDI for this process might occupy if mapped to FPGA. We assumed that this value is proportional to the process size.

Figure 12a depicts the ranges for testcase1: for the SW-only EDI, we considered a time overhead as big as 300% and as low as 80%, related to the worst-case execution time of the corresponding process; obviously, the HW cost overhead in this case is 0. For the

[2] We also conducted experiments with $k \in [3, 8]$ and we concluded that the impact of different k is not significant for the quality of results produced by our heuristic.

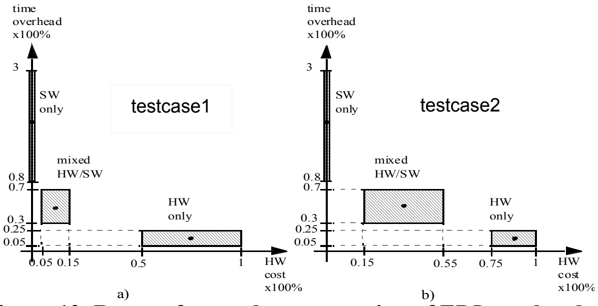


Figure 12. Ranges for random generation of EDI overheads

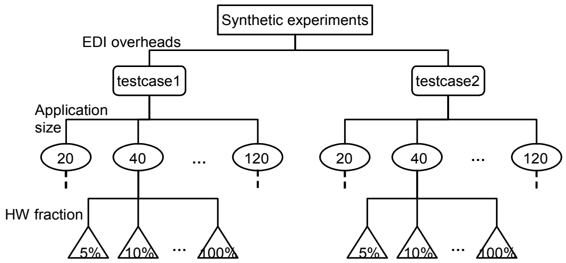


Figure 13. Synthetic experiment space

mixed HW/SW implementation, the time overhead range is between 30% and 70%, and the HW cost overhead range is between 5% and 15%. Finally, the HW-only implementation would incur a time overhead between 5% and 25% and a HW cost overhead between 50% and 100%. Figure 12b depicts the ranges for testcase2: the time overhead ranges are the same, but we pushed the HW cost ranges more to the right. Also note that for testcase2, the centers of gravity of the considered areas are more uniformly distributed. The execution time overheads and the HW cost overheads for the processes in our synthetic examples are distributed uniformly in the intervals depicted in Figure 12a (testcase1) and Figure 12b (testcase2).

We also varied the size of every FPGA available for placement of error detection. We proceeded as follows: we sum up all the HW cost overheads corresponding to the HW-only implementation, for all processes of a certain application:

$$MAX\_HW = \sum_{i=1}^{card(V)} C(P_i, HW\_only)$$

Then we generated problem instances by considering the size of each FPGA corresponding to different fractions of MAX_HW: 5%, 10%, 15%, 20%, 25%, 30%, 35%, 40%, 60%, 80%, 90% and 100%, distributed evenly among computation nodes. The 100% is an extreme case, and represents the situation in which we have available all the HW that we need, so the optimization problem actually disappears. Figure 13 shows the resulting space of experiments. A total of 2×6×15×12 = 2160 different settings were used for experimental evaluation.

8.1 Static Reconfiguration Approach

As a first step we considered for all our generated process graphs the SW-only EDI. The worst-case schedule length for this case, $WCSL_{baseline}$, was used as a baseline. For the same process graphs, we then considered the various HW fractions assigned and for each case we calculated the worst-case schedule length, $WCSL_{static}$, after applying our heuristic. The performance improvement (PI) obtained for each individual case is:

$$PI = \left(\frac{WCSL_{baseline} - WCSL_{static}}{WCSL_{baseline}} \times 100 \right)\%$$

In order to evaluate the proposed heuristic we first were interested to compare our results with the theoretical optimum. We implemented a Branch and Bound (BB) based search and calculated the performance improvement similar to that above, but using the theoretical optimum worst-case schedule length, $WCSL_{opt}$, instead of $WCSL_{static}$. Of course, it was possible to obtain the optimal solution only for the application size of 20 and examples with up to 40% HW fraction.

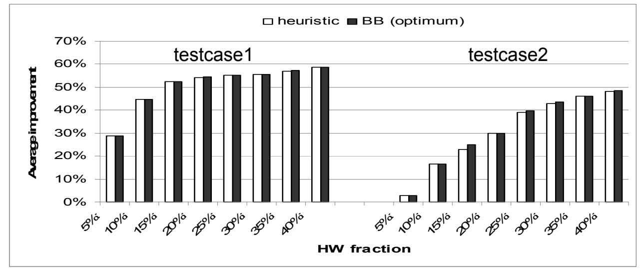


Figure 14. Comparison with theoretical optimum

Figure 14 shows the average improvement over all test cases for our heuristic and for the optimal solution. Considering all the cases, the differences between our heuristic and the optimum were up to 1% for testcase1, and up to 2.5% for testcase2, which shows the effectiveness of our approach.

Next, we were interested to evaluate the impact of the HW fraction assigned to each FPGA, on the WCSL improvement. Figure 15 shows the average improvement we obtained when running our heuristic. It can be seen that we shortened the WCSL with up to 64% (compared to the baseline – SW-only solution). As expected, assigning more HW to FPGAs increases the improvement. We can also observe that this happens up to a saturation point: beyond that point, assigning more HW area does not help. The reason is that, at the saturation point, all processes having an impact on the schedule length already have their best EDI assigned, while moving the EDI for other processes into HW does not impact the WCSL.

We would also like to point out that, with only 15% HW fraction, we can reduce the WCSL by more than half (i.e. get an improvement >50%), for testcase1. For testcase2, in order to reduce the WCSL by half, we need ~40% HW fraction. This difference is due to the assumptions we made when generating testcase2 examples (see Figure 12), namely that the hardware is slower and, thus, we need more HW in order to get the same performance as for testcase1. As we can see from Figure 15, this difference also influences the saturation point for testcase2 (~90% HW fraction, compared to ~60% for testcase1).

8.2 PDR Approach

In order to evaluate the efficiency of implementing error detection on FPGAs with partial dynamic reconfiguration we kept the same experimental setup as for the static case (Figure 13).

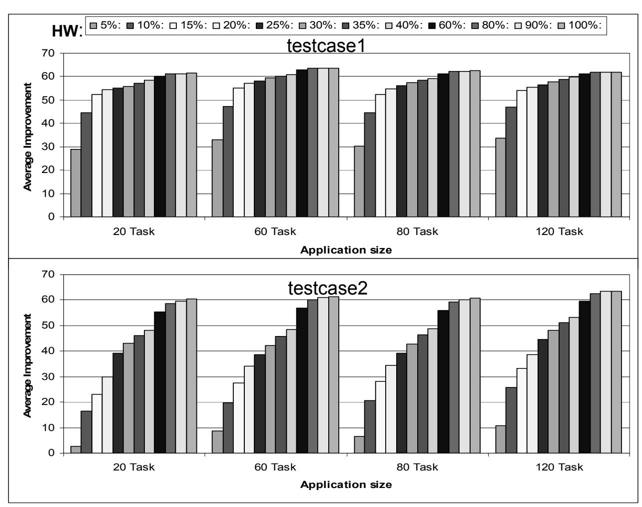


Figure 15. Impact of varying HW fraction

For these experiments we considered the worst-case schedule length, $WCSL_{static}$, obtained considering static reconfiguration, as a baseline. The performance improvement obtained with PDR over the static reconfiguration approach is:

$$PI^{PDR} = \left(\frac{WCSL_{static} - WCSL_{PDR}}{WCSL_{static}} \times 100 \right) \%$$

In Figure 16 we show the average performance improvement for the PDR approach over the static. By employing PDR of the FPGA, execution of some processes was parallelized with the reconfiguration of the error detectors for others (masking reconfiguration overheads). Reusing FPGA area also enabled placing of EDIs more into HW and, consequently, we were able to shorten the schedule length with up to 36% (with a HW fraction of only 5%) for testcase1 and with up to 34% (with a HW fraction of 25%) for testcase2.

One interesting aspect related to testcase1 is that, after the peak of improvement gain, the improvement drops with increasing HW fraction, then it slightly increases and finally drops again (for large amounts of HW the improvement gain is zero, which means that using PDR produces the same WCSL as the static reconfiguration approach). The explanation resides in the ranges we have chosen for EDI overheads in testcase1 (see Figure 12a). The initial peak of improvement gain is the result of the fact that the PDR algorithm is able to move more EDIs to the mixed HW/SW implementation. Then, assigning more HW does not help proportionally much, since the heuristic is not yet able to move EDIs even more into HW, to their HW-only implementation. This is due to the large gap between the mixed HW/SW and HW-only EDIs generated for testcase1. So, actually, we cannot take full advantage of the partial dynamic reconfiguration capabilities, since some EDIs (HW-only implementation) might have their size even bigger than the entire size of the FPGA. Further, as we assign more HW area, the FPGA is able to accommodate the HW-only implementation of error detectors, so another increase in improvement happens. Finally, for big FPGA sizes, employing PDR does not make a difference anymore, since we have enough space from the beginning, so the static strategy can readily place all the needed EDIs.

With regard to testcase2, changing the ranges of EDI overheads, as compared to testcase1, impacts the results of the heuristic. The trend observed for testcase1 is not visible anymore, since now, the transition from SW-only, to mixed HW/SW and then to HW-only implementation of error detectors is smoother and more uniform (see Figure 12b). In other words, the gap (concerning HW cost) between mixed HW/SW and HW-only implementation is smaller in testcase2. As expected, the maximum improvement (34%) in this second case corresponds to a HW fraction of ~25% (compared with 5% for testcase1).

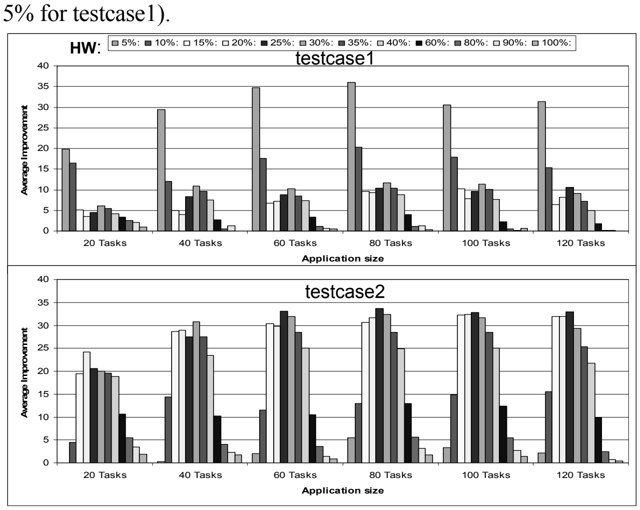


Figure 16. Improvement - PDR over static approach

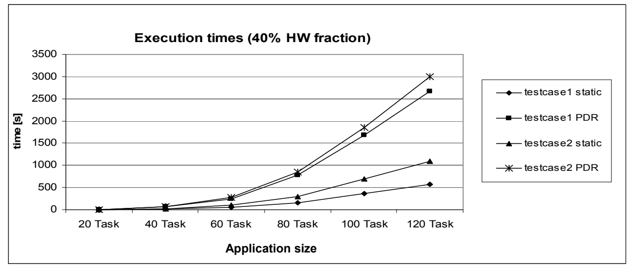


Figure 17. Execution times

Figure 17 presents the execution times of our optimization heuristics for different application sizes. All experiments were run on a PC with CPU frequency 2.83 GHz, 8 GB of RAM, and running Windows Vista. The values correspond to the setting with 40% HW fraction, which is usually the case producing the longest execution times. For settings with tight cost constraints (i.e. small HW fraction) the algorithm does not have many alternative solutions to choose from, so a result is reached faster (e.g, for a 5% HW fraction and 80 processes application size, the execution time is roughly 28% shorter for testcase1 static, and 80% shorter for testcase2 static, while for testcase1 PDR is 18% shorter, and for testcase2 PDR is 67% shorter than the corresponding values in Figure 17). For settings with big HW fractions, the algorithms converge to good solutions relatively fast, since there is more freedom to place EDIs on FPGA. For example, for a 90% HW fraction and 60 processes application size, for both approaches, the execution times were around 19% shorter for testcase1 and for testcase2 around 25% shorter than the corresponding values in Figure 17. Another aspect worth mentioning is that the PDR approach takes considerably more time, because in this case we also have to adjust the weights for the scheduler (see Section 7).

8.3 The Adaptive Cruise Controller

We also tested our approach on a real-life example, an adaptive cruise controller (ACC), similar to the one described in [1]. The process graph is composed of 13 processes, depicted in Figure 18. The adaptive cruise controller helps the driver keep a desired speed and a safe distance to the preceding vehicle. It also has the possibility of autonomous changes of the maximum speed depending on the speed-limit regulations and helps the driver with the braking procedure in extreme situations. The functionality of the adaptive cruise controller is as follows: based on the driver specification and on the speed-limit regulations, the SpeedLimit process computes the actual speed limit allowed in a certain situation. The Object Recognition process calculates the relative speed to the vehicle in front. This component is also used to trigger ModeSwitch in case there is a need to use the brake assist functionality. ModeSwitch is used to trigger the execution of the ACC or of the BrakeAssist component. The ACC assembly (P_9 and P_{10}) controls the throttle lever, while the BrakeAssist process is used to slam the brakes if there is an obstacle in front of the vehicle that might cause a collision.

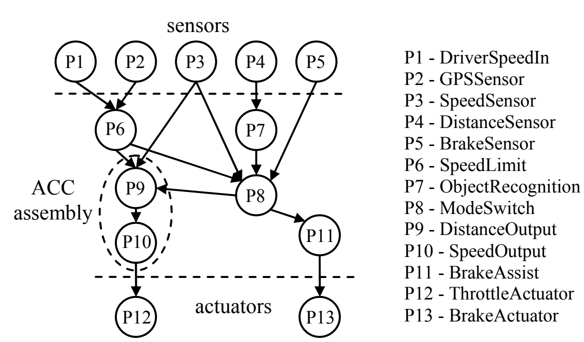


Figure 18. Adaptive cruise controller

Table 3. Time and area overheads

Task	Error Detection						
	SW-only	Mixed HW/SW			HW-only		
	WCET (μs)	WCET (μs)	area (slices)	rec. time (μs)	WCET (μs)	area (slices)	rec. time (μs)
P1	8.12	6.02	2	1.23	4.9	15	9.23
P2	8.12	6.02	2	1.23	4.9	15	9.23
P3	8.12	6.02	2	1.23	4.9	15	9.23
P4	8.17	6.07	2	1.23	4.95	15	9.23
P5	7.32	5.4	2	1.23	4.27	13	8
P6	8.37	6.27	2	1.23	4.92	13	8
P7	10.35	7.87	2	1.23	6.75	42	25.83
P8	12.17	9.6	3	1.85	8.45	15	9.23
P9	46.22	33.7	11	6.77	27.25	43	26.45
P10	20.3	14.45	4	2.46	11.57	28	17.22
P11	37.27	29.47	7	4.31	21.7	52	31.99
P12	8.12	6.02	2	1.23	4.9	15	9.23
P13	8.12	6.02	2	1.23	4.9	15	9.23

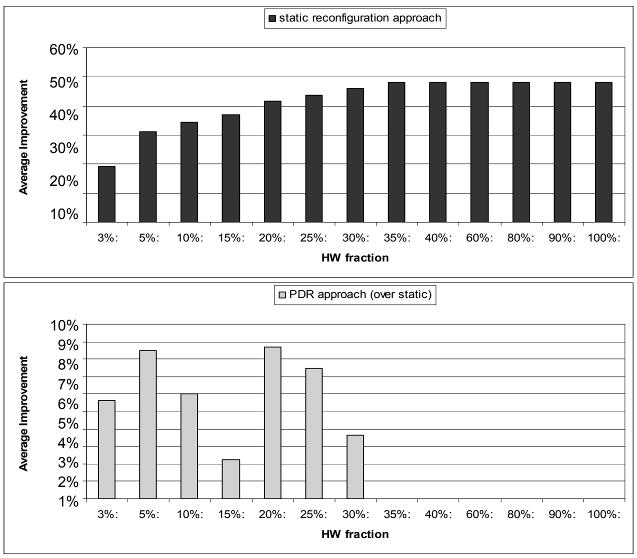


Figure 19. ACC example results

We instrumented every process of this application with error detectors, according to the technique described in Section 2.1. The execution times were derived considering an ARM processor with an operational frequency of 40 MHz. The path trackers and checking modules where synthesized on XC5VLX50 Virtex-5 device, using the Xilinx ISE WebPack. The reconfiguration times where computed considering a 60 MHz configuration clock frequency and the ICAP 32-bit width configuration interface. In order to reduce the reconfiguration granularity we used a methodology similar to the one presented in [20]. The execution times, as well as the hardware overheads obtained are given in Table 3.

We mapped this application on an architecture with two computation nodes. Considering a number of $k = 2$ faults, the results obtained are presented in Figure 19. As we can see, using the static reconfiguration approach, we can get up to 47% reduction in schedule length (over the SW-only implementation), while assuming PDR for FPGAs we manage to get an extra 9% reduction (on top of the static reconfiguration approach).

9. CONCLUSIONS

In this paper we have presented an approach to optimization of error detection implementation in the context of fault-tolerant real-time embedded systems. We proposed heuristic approaches for solving the problem, and ran extensive experiments to prove the effectiveness of our algorithms.

We have revealed a big potential for performance improvement through optimization of error detection implementation. We obtained significant reductions of worst-case schedule length with relatively small HW resources available. Assuming that the FPGAs support partial dynamic reconfiguration, the experimental results show an important reduction in worst-case schedule length, compared to the static reconfiguration approach.

REFERENCES

[1] Akerholm, M., Moller, A., Hansson, H. and Nolin, M., “Towards a Dependable Component Technology for Embedded System Applications”, *Intl. Workshop on Object-Oriented Real-Time Dependable Systems,* 2005, 320-328.

[2] Banerjee, S., Bozorgzadeh, E. and Dutt, N., “Physically-Aware HW-SW Partitioning for Reconfigurable Architectures with Partial Dynamic Reconfiguration”, *DAC*, 2005, 335-340.

[3] Blome, J.A., Gupta, S., Feng, S., and Mahlke, S., “Cost-Efficient Soft Error Protection for Embedded Microprocessors”, *CASES*, 2006, 421-431.

[4] Bolchini, C., Miele, A., Rebaudengo, M., Salice, F., Sciuto, D., Sterpone, L. and Violante, M., “Software and Hardware Techniques for SEU Detection in IP Processors”, *J. Electron. Test.*, 24 (1-3), 2008, 35-44.

[5] Constantinescu, C., “Trends and Challenges in VLSI Circuit Reliability”, *IEEE Micro*, 23 (4), 2003, 14-19.

[6] Cordone, R., Redaelli, F., Redaelli, M.A., Santambrogio, M.D. and Sciuto, D., “Partitioning and Scheduling of Task Graphs on Partially Dynamically Reconfigurable FPGAs”, *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 28 (5), 2009, 662-675.

[7] Eles, P., Doboli, A., Pop, P. and Peng, Z., “Scheduling with Bus Access Optimization for Distributed Embedded Systems”, *IEEE Trans. Very Large Scale Integr. Syst.*, 8 (5), 2000, 472-491.

[8] Garey, M.R. and Johnson, D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. H. Freeman & Co., New York, NY, USA, 1990.

[9] Hu, J., Li, F., Degalahal, V., Kandemir, M., Vijaykrishnan, N. and Irwin, M.J., “Compiler-Assisted Soft Error Detection under Performance and Energy Constraints in Embedded System”, *ACM Trans. Embed. Comput. Syst.*, 8 (4), 2009, 1-30.

[10] Izosimov, V., Pop, P., Eles, P. and Peng, Z., “Synthesis of Fault-Tolerant Schedules with Transparency/Performance Trade-offs for Distributed Embedded Systems”, *DATE*, 2006, 1-6.

[11] Izosimov, V., Pop, P., Eles, P. and Peng, Z., “Design Optimization of Time- and Cost-Constrained Fault-Tolerant Distributed Embedded Systems”, *DATE*, 2005, 864-869, Vol. 2.

[12] Kopetz, H. and Bauer, G., “The Time-Triggered Architecture”, *Proceedings of the IEEE*, 91 (1), 2003, 112-126.

[13] Lima, F., Carro, L. and Reis, R., “Designing Fault-Tolerant Systems into SRAM-Based FPGAs”, *DAC*, 2003, 650-655.

[14] Lyle, G., Chen, S., Pattabiraman, K., Kalbarczyk, Z. and Iyer, R.K., “An End-to-End Approach for the Automatic Derivation of Application-Aware Error Detectors”, *DSN*, 2009, 584-589.

[15] Pattabiraman, K., Kalbarczyk, Z. and Iyer, R.K., “Automated Derivation of Application-Aware Error Detectors using Static Analysis: the Trusted Illiac Approach”, *IEEE Trans. Dependable Secure Comput.*, 99, 2009.

[16] Pattabiraman, K., Kalbarczyk, Z. and Iyer, R.K., “Application-Based Metrics for Strategic Placement of Detectors”, *PRDC*, 2005, 75-82.

[17] Reeves, C.R., *Modern Heuristic Techniques for Combinatorial Problems*, John Wiley & Sons, Inc., New York, NY, USA, 1993.

[18] Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I. and Mukherjee, S.S., “Software-Controlled Fault Tolerance”, *ACM Trans. Archit. Code Optim.*, 2 (4), 2005, 366-396.

[19] Schuck, C., Kühnle, M., Hübner, M. and Becker, J., “A Framework for Dynamic 2D Placement on FPGAs”, *Intl. Symp. on Parallel and Distributed Processing*, 2008, 1-7.

[20] Sedcole, P., Blodget, B., Anderson, J., Lysaght, P. and Becker, T., “Modular Partial Reconfiguration in Virtex FPGAs”, *Intl. Conf. on Field Programmable Logic and Applications*, 2005, 211-216.

[21] Wei, T., Mishra, P., Wu, K. and Liang, H., “Online Task-Scheduling for Fault-Tolerant Low-Energy Real-Time Systems”, *Intl. Conf. on Computer-Aided Design*, 2006, 522-527.

[22] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J. and Stenström, P., “The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools”, *ACM Trans. Embed. Comput. Syst.*, 7 (3), 2008, 1-53.

[23] Wirthlin, M., Johnson, E., Rollins, N., Caffrey, M. and Graham, P., “The Reliability of FPGA Circuit Designs in the Presence of Radiation Induced Configuration Upsets”, *Symp. on Field-Programmable Custom Computing Machines*, 2003, 133-142.

[24] Xilinx Inc., “Early Access Partial Reconfiguration User Guide”, Xilinx UG208 (v1.1), March 6, 2006.