Performance Optimization of Error Detection Based on Speculative Reconfiguration

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

ABSTRACT

This paper presents an approach to minimize the average program execution time by optimizing the hardware/software implementation of error detection. We leverage the advantages of partial dynamic reconfiguration of FPGAs in order to speculatively place in hardware those error detection components that will provide the highest reduction of execution time. Our optimization algorithm uses frequency information from a counter-based execution profile of the program. Starting from a control flow graph representation, we build the interval structure and the control dependence graph, which we then use to guide our error detection optimization algorithm.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance, Design, Reliability.

Keywords

Error detection implementation, reconfigurable systems, fault tolerance, FPGA, system-level optimization.

1. INTRODUCTION

In the context of electronic systems implemented with modern semiconductor technologies faults have become more and more frequent. 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 [3]. Our work focuses on optimizing the error detection for these types of faults, while permanent ones are not addressed here.

As error detection is needed, no matter what tolerance strategy is applied, the detection mechanisms are always present and active. Unfortunately, they are 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 cost of the system. Thus, careful optimization of error detection early in the design phase of a system is important.

Previous work proposed various techniques for error detection, both software and hardware-based [2, 6, 15], or focused on efficiently implementing the error detection at a system level [10]. There has been much research done in the area of speculative execution and supporting compiler techniques, but there has been less work addressing the problem of performance optimization using speculative reconfiguration of FPGAs. Jean et al. describe in [7] a dynamically reconfigurable system that can support multiple applications running concurrently and implement a strategy to preload FPGA configurations in order to reduce the execution time. Several other papers present different configuration prefetching techniques in order to minimize the reconfiguration overhead [4, 9, 13, 19], but error detection or fault tolerance are not tackled in any of them. Many of the previous papers address the performance optimization problem at a task level. For a large class of applications (e.g. those that consist of a single sequential task) using such a task-level coarse granularity is not appropriate and, instead of considering the tasks as black boxes, it is necessary to analyze their internal structure and properties.

We propose a speculative reconfiguration strategy to optimize the hardware/software implementation of error detection, in order to minimize the average execution time, while meeting the imposed hardware area constraints. The frequency information used to guide our decision process is obtained from a counter-based execution profile of the program.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC'11, June 5-10, 2011, San Diego, California, USA
Copyright © 2011 ACM 978-1-4503-0636-2/11/06...$10.00

2. PRELIMINARIES

2.1 Error Detection Technique

We focus on application-aware error detection techniques, which make use of the knowledge about the application's characteristics in order to create customized solutions, tuned to better suit each application's needs. The main idea of the application-aware technique proposed in [12] is to identify, based on specific metrics [14], *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" [12]. 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" [12]. Next, each slice is aggressively optimized at compile time, resulting in a series of checking expressions (checkers). These will be inserted in the original code after the computation of a critical variable. The checkers can be seen as customized assertions, specialized for each acyclic control path in the backward program slice of a 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 checker, and then compare the results obtained.

The above technique has two main sources of performance overhead: path tracking and variable checking. Both can be implemented either in software, potentially incurring high performance overheads, or in hardware, which can lead to costs sometimes exceeding the amount of available resources. In [12], complete hardware implementations of the path tracker and checkers have been proposed. The modules are part of a framework tightly interconnected with the host processor (the pipeline signals are exposed through an interface, some special purpose registers provide fast communication and the framework also has a direct memory access module for improved effectiveness). The path tracker is efficient and low-overhead. Unfortunately, implementing each checker to its own dedicated hardware incurs excessive costs. In order to overcome this wasteful use of hardware resources, we propose to leverage the advantages of partial dynamic reconfiguration (PDR) of FPGAs in order to place in hardware only those checkers that have the potential to provide the highest performance improvement.

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.

The authors of [12] report that it is possible to achieve 75%-80% error coverage for crashes by checking the five most critical variables in each function on a set of representative benchmarks. Fault injection experiments conducted in [14] showed that the above technique can reach a coverage of 99% by considering 25 critical variables for the *perl* interpreter. 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). Some hardware redundancy techniques might also be used to deal with the remaining, not covered, faults [2]. In this paper we concentrate on the optimization of the application-aware error detection component.

2.2 Basic Concepts

Definition 1: A *control flow graph (CFG)* of a program is a directed graph $G_{cf} = (N_{cf}, E_{cf})$, where each node in N_{cf} corresponds to a straight-line sequence of operations and the set of edges E_{cf} corresponds to the possible flow of control within the program. G_{cf} captures potential execution paths and contains two distinguished nodes, *root* and *sink*, corresponding to the entry and the exit of the program.

Definition 2: A node $n \in N_{cf}$ is *post-dominated* by a node $m \in N_{cf}$ in the control flow graph $G_{cf} = (N_{cf}, E_{cf})$ if every directed path from n to *sink* (excluding n) contains m.

Definition 3: Given a control flow graph $G_{cf} = (N_{cf}, E_{cf})$, a node $m \in N_{cf}$ is *control dependent* upon a node $n \in N_{cf}$ via a control flow edge $e \in E_{cf}$ if the below conditions hold:

- there exists a directed path P from n to m in G_{cf}, starting with e, with all nodes in P (except m and n) post-dominated by m;
- m does not post-dominate n in G_{cf}.

In other words, there is some control edge from n that definitely causes m to execute, and there is some path from n to *sink* that avoids executing m.

Definition 4: A *control dependence graph (CDG)* $G_{cd} = (N_{cd}, E_{cd})$ corresponding to a control flow graph $G_{cf} = (N_{cf}, E_{cf})$ is defined as: $N_{cd} = N_{cf}$ and $E_{cd} = \{((n, m), e) \mid m$ is control dependent upon n via edge $e\}$. A *forward control dependence graph (FCDG)* is obtained by ignoring all back edges in CDG.

Definition 5: An *interval I(h)* in a control flow graph $G_{cf} = (N_{cf}, E_{cf})$, with header node $h \in N_{cf}$, is a strongly connected region of G_{cf} that contains h and has the following properties:

- $I(h)$ can be entered only through its header node h;
- all nodes in $I(h)$ can be reached from h along a path contained in $I(h)$;
- h can be reached from any node in $I(h)$ along a path contained in $I(h)$.

The interval structure represents the looping constructs of a program.

Definition 6: An *extended control flow graph (ECFG)* is obtained by augmenting the initial control flow graph with a pseudo control flow edge (*root, sink*) and with explicit interval entry and exit nodes as follows:

- for each interval $I(h)$, a preheader node, ph, is added and all interval entries are redirected to ph (i.e. every edge (n, h), with n not in $I(h)$ is replaced with the edge (n, ph) and an unconditional edge (ph, h) is added);
- a postexit node, pe, is added and every edge (n, m) with n in $I(h)$ and m not in $I(h)$ is replaced by edges (n, pe) and (pe, m);
- a pseudo control flow edge (ph, pe) is added.

The pseudo control flow edges provide a convenient structure to the control dependence graph (causing all nodes in an interval to be directly or indirectly control dependent on the preheader node).

Definition 7: Given an edge $(n, m) \in E_{fcd}$ in the forward control dependence graph (FCDG) $G_{fcd} = (N_{fcd}, E_{fcd})$, the execution frequency of the edge [16] is defined as follows:

- if n is a preheader and m is a header, $FREQ(n, m) \geq 0$ is the average loop frequency for interval $I(m)$;
- otherwise, $0 \leq FREQ(n, m) \leq 1$ is the branch probability.

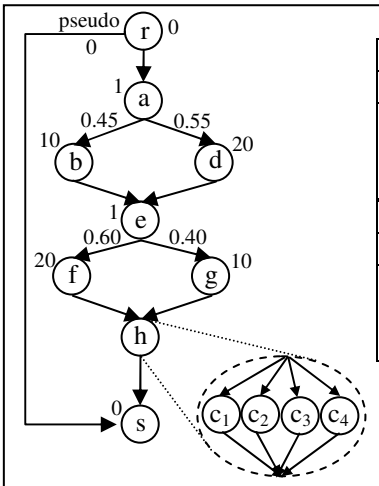

Table 1. Overheads

		c_1	c_2
checking expr.			
SW - overhead		22	18
HW	overhead	4	3
	area	25	10
	recon. time	25	10
checking expr.		c_3	c_4
SW - overhead		15	29
HW	overhead	2	3
	area	20	40
	recon. time	20	40

$PATH(c_1) = \{(a, b), (e, f)\}$
$PATH(c_2) = \{(a, b), (e, g)\}$
$PATH(c_3) = \{(a, d), (e, f)\}$
$PATH(c_4) = \{(a, d), (e, g)\}$

Table 2. Conditional Reconfiguration Schedule Tables

Condition:	(a, b)	$(a, b) \Lambda (e, g)$	(a, d)	$(a, d) \Lambda (e, g)$
S_1 action:	rec. c_1	rec. c_2	rec. c_3	–
Condition:	(a, b)	$(a, b) \Lambda (e, g)$	(a, d)	$(a, d) \Lambda (e, f)$
S_2 action:	rec. c_1	rec. c_2	rec. c_4	rec. c_3

Figure 1. Motivational Example

3. MOTIVATIONAL EXAMPLE

Let us consider the ECFG in Figure 1. The graph shows two conditional branches. The execution times of the nodes are: $a = 1$, $b = 10$, $d = 20$, $e = 1$, $f = 20$, $g = 10$, while the root (r) and sink (s) have 0 execution times. The conditional edges have their associated probabilities presented on the graph.

Let us assume that inside node h we have a critical variable that needs to be checked and, by applying the technique described in Section 2.1, we derived four checkers for it: c_1 to c_4. Each of them corresponds to a different acyclic control path in the ECFG, as shown in Figure 1. In Table 1 we present the time overheads incurred by the software implementation of each checker, as well as the overheads, area and reconfiguration time incurred by the hardware implementation. Since node h is composed of the four checkers, its execution time depends on the actual path followed at runtime, which will determine which of the four, mutually exclusive, checkers will get executed.

For the given example, if we assume that all the checkers are implemented in software, the resulting average execution time of the ECFG is 54.01. On the other hand, if we assume that we have enough FPGA area so that we can place all the checkers in hardware from the beginning (i.e. 95 area units), the average execution time will be 36.44.

Let us now consider an FPGA supporting partial dynamic reconfiguration and having an area of only 40 units. For the reconfiguration decision on each of the outgoing edges of node a, one intuitive solution would be the following: if edge (a, b) is followed, we would start reconfiguring the checker that has the highest probability to be reached on this branch (i.e. c_1). At runtime, when we reach the second conditional node, if the edge (e, f) is followed, then our previous speculation was right, and the reconfiguration of c_1 will have been finished by the time we reach it; otherwise, if the edge (e, g) is taken, then we can drop the reconfiguration of c_1 (because we will not execute it for sure) and we can start the reconfiguration of c_2. Let us now consider the situation when edge (a, d) is followed. Applying the same strategy as above, we would start reconfiguring c_3 (because its probability is 60% compared to 40% for c_4). In this case, when we reach node e, if the prediction was right, we will have c_3 on FPGA by the time we reach it. But if the prediction was wrong, it is not profitable to reconfigure c_4 (on edge (e, g)) because reconfiguring it and then executing it in hardware would take 40 + 3 = 43 time units (note that node g is executed in parallel with the reconfigura-

tion), while leaving c_4 in software will result in a shorter execution time (10 + 29 = 39 time units). The result is that only c_1, c_2 and c_3 will possibly be placed in hardware, while c_4 is kept in software. In Table 2 we capture the discussion above in the form of a conditional reconfiguration schedule (the alternative denoted with S_1). This schedule table contains a set of conditions and the corresponding reconfigurations that should be started when the condition is true. These conditions are given as a conjunction of edges, meaning that in order for the condition to evaluate to true, the edges have to be followed in the specified order at runtime. The schedule alternative S_1 will lead to an average execution time of 42.16.

However, investigating carefully the example, one can observe that a better scheduling alternative than S_1 can be generated. We can see that, in case edge *(a, d)* is taken, it is better to start the reconfiguration of c_4 (although it might seem counterintuitive since it has a smaller probability to be reached than c_3). Doing this way, if the prediction is wrong, it is still profitable to later place c_3 in hardware (schedule S_2 from Table 2). For c_3 it is possible to postpone its reconfiguration until a later point, since its execution is further away in time and its reconfiguration overhead is smaller than that of c_4. Even if we start the reconfiguration of c_4 first, we still introduce a waiting time because the reconfiguration is not ready by the time we reach node *h* (we start reconfiguring c_4 at the end of node *a*, and the path to *h* is only 20 + 1 + 10 = 31 time units, while the reconfiguration takes 40). Nevertheless, waiting 9 time units and then executing the checker in hardware for 3 units is better than executing it in software for 29 time units. So, by applying schedule S_2, the resulting average execution time is 38.42, with an FPGA of just 40 area units.

When generating the more efficient scheduling alternative S_2 we did not base our decisions exclusively on execution probabilities (as is the case with S_1). Instead, we took into account the time gain resulted from each checker, as well as its reconfiguration overhead in conjunction with the length of the path from the current decision point, up to the location where the checker will be executed.

4. SYSTEM MODEL

We consider a *structured* [1] program[1], modeled as a control flow graph $G_{cf} = (N_{cf}, E_{cf})$. The program runs on an architecture composed of a central processing unit, a memory subsystem and a reconfigurable device (FPGA). Knowing that SRAM-based FPGAs are susceptible to single event upsets [20], we assume that suitable mitigation techniques are employed (e.g. [11]) in order to provide sufficient reliability of the hardware used for error detection.

We model our FPGA, supporting partial dynamic reconfiguration, as a rectangular matrix of configurable logic blocks (CLBs). Each checker occupies a contiguous rectangular area of this matrix. The model allows modules of any rectangular shape and size. The execution of a checker can proceed in parallel with the reconfiguration of another checker, but only one reconfiguration may be done at a certain time.

5. PROBLEM FORMULATION

Input: a structured program modeled as a CFG $G_{cf} = (N_{cf}, E_{cf})$. Each node has a corresponding execution time: $TIME : N_{cf} \rightarrow \mathbf{Z}^+$. Each edge has associated the probability to be taken: $PROB : E_{cf} \rightarrow [0; 1]$. We assume that this information is obtained by profiling the program and it is given.

The set of checkers (checking expressions), *CE*, corresponding to the program is also given. Each checker $c \in CE$ has associated the following:

1. the node where it will be executed ($HOST : CE \rightarrow N_{cf}$);
2. the acyclic path for which the checker has been derived (denoted with *PATH(c)* = sequence of edges that lead to *HOST(c)* in the CFG);
3. the execution time ($SW : CE \rightarrow \mathbf{Z}^+$) if implemented in software;
4. the execution time ($HW : CE \rightarrow \mathbf{Z}^+$), necessary FPGA area ($AREA : CE \rightarrow \mathbf{Z}^+ \times \mathbf{Z}^+$) and reconfiguration overhead ($REC : CE \rightarrow \mathbf{Z}^+$) if implemented in hardware.

The total available FPGA area which can be used to implement the error detection is also given.

Goal: to minimize the average execution time of the program, while meeting the imposed hardware area constraints.

Output: a conditional reconfiguration schedule table.

6. OPTIMIZATION USING SPECULATIVE RECONFIGURATION

Latest generations of commercially available FPGA families provide support for partial dynamic reconfiguration (PDR) [21]. This means that parts of the device may be reconfigured at runtime, while other parts remain functional [17].

We propose a constructive algorithm to solve the problem defined in Section 5. The idea is to use the knowledge regarding already taken branches, in order to take the best possible decision for the future and speculatively reconfigure on FPGA the checkers with the highest potential to provide a performance improvement.

Since for each checker $c \in CE$ we have the corresponding acyclic path specified as a sequence of instrumented branches (edges from the CFG), we define the *reach probability* ($REACH\_PROB : N_{cf} \times CE \rightarrow [0; 1]$): *REACH_PROB(n, c)* is the probability that the checker's path is followed at runtime, given that node $n \in PATH(c)$ is reached and all the nodes on *PATH(c)* up to *n* where visited. This is computed as the product of the probabilities of the edges in the checker's path, from *n* up to *HOST(c)*. If *n* is not on *PATH(c)*, then the reach probability will be zero. Considering the example from Figure 1, e.g. $REACH\_PROB(a, c_1) = 0.45 \times 0.60 = 0.27$, $REACH\_PROB(b, c_1) = 0.60$ and $REACH\_PROB(d, c_1) = 0$.

For each checker $c \in CE$ we also define its *time gain* as the difference between its time overhead when implemented in SW versus in HW ($GAIN : CE \rightarrow \mathbf{Z}^+$, $GAIN(c) = SW(c) - HW(c)$). Finally, we denote with *ITERATIONS(c)* the average frequency of the loop inside which *c* is executed. If *c* is not inside a loop, $ITERATIONS(c) = 1$. We will use this value to give higher priority to checkers that get executed inside loops.

We also assign a weight to each checker $c \in CE$:

$$WEIGHT(n,c) = \frac{REACH\_PROB(n,c) \times ITERATIONS(c) \times GAIN(c)}{AREA(c)}$$

This weight represents the average time gain per area unit corresponding to *c*. As the execution of the program proceeds, the weights of some checkers might become zero (if certain paths are followed, such that those checkers will never get executed, i.e. *REACH_PROB* becomes zero), while the weight of other checkers will increase (as we approach their *HOST* on a certain path).

6.1 Reconfiguration Schedule Generation

The pseudocode of our reconfiguration schedule generation algorithm is presented in Figure 2. Considering the CFG received as an input, we first determine its interval structure (line 1), then we build the ECFG (line 2) according to Definition 6, and the FCDG (line 3) according to Definition 4.

The next step is to traverse the ECFG and build the reconfiguration schedule (line 5). Once the schedule is generated all the checkers have been assigned an implementation (in hardware or in software), on any path that might get executed at runtime. Thus we can estimate the average execution time of the given CFG (line 6).

The actual optimization is performed in the recursive procedure **build_schedule**. The ECFG is traversed such that each back edge and each loop exit edge is processed exactly once (line 8). In order to be able to handle the complexity, whenever a back or loop exit edge is taken, we reset the current context (lines 10-11). This means that in each loop iteration we take the same reconfiguration decisions, regardless of what happened in previous loop iterations. The same applies to loop exit: decisions after a loop are independent of what happened inside or before the loop. This guarantees that the conditional reconfiguration schedule table that we generate is com-

[1] Since any non-structured program is equivalent to some structured one, our assumption loses no generality.

```
build_reconfiguration_schedule (CFG cfg)
1    determine interval structure of CFG
2    build ECFG
3    build the FCDG
4    //traverse ECFG and build the reconfiguration schedules
5    build_schedule(root, TRUE)
6    compute the average execution time of CFG
end build_reconfiguration_schedule

build_schedule(NODE n, CONDITION cond)
7    for each outgoing edge e = (n, m)
8        if (e is not a back edge and not a loop exit) or (e has not been visited)
9            mark e as visited
10           if e is a back edge or a loop exit edge
11               cond = TRUE; //reset context
12           new_cond = cond Λ e
13           for each checker c ϵ CE
14               compute REACH_PROB(m, c) and WEIGHT(m, c)
15               compute PATH_LENGTH(n, c)
16           build ACE(e)
17           if reconfiguration controller available and ACE(e) ≠ Ø
18               schedule_new_reconfigurations(e, new_cond)
19           build_schedule(m, new_cond)
end build_schedule

schedule_new_reconfigurations(EDGE e = (n, m), CONDITION cond)
20   sort ACE(e) in descending order by WEIGHT
21   do
22       pick and remove c ϵ ACE(e) with highest WEIGHT
23       if c already configured on FPGA but inactive
24           activate c at its current location
25       else
26           loc = choose_FPGA_location(c)
27           mark start of reconfiguration for c, at loc, on new_cond
28           for all checkers c' ϵ ACE(e)
29               recompute PATH_LENGTH(n, c')
30   while (end of reconfiguration < end of m's execution)
end schedule_new_reconfigurations

choose_FPGA_location(CHECKER c)
31   if there is enough free contiguous space on FPGA
32       return the empty rectangle with lowest fragmentation metric
33   else
34       choose the least recently used inactive modules to be replaced
35       return this location
end choose_FPGA_location
```

Figure 2. Speculative Optimization Algorithm

plete (i.e. we tried to take the best possible decision for any path that might be followed at runtime). The generated schedule table contains a set of conditions and the corresponding reconfigurations to be performed when a condition is activated at runtime. Considering the example in Figure 1 a condition and its corresponding action is:

(a, b) Λ (e, g): reconfiguration of c_2.

At runtime, a condition is activated only if the particular path specified by its constituent edges is followed. Once it is activated, the condition will be restarted, in order to prepare for possible future activations (if, for example, the path is inside a loop).

At each control node $n \in N_{ecf}$ in the ECFG we take a different reconfiguration decision on every outgoing edge (line 7). We will next describe the decision process for one such edge, $e = (n, m) \in E_{ecf}$. First, the new condition is constructed (line 12). This condition corresponds to a conjunction of all the taken edges so far, and the current edge, *e*.

As we process a control edge $e = (n, m) \in E_{ecf}$ in the ECFG, we compute the *REACH_PROB(m, c)* and the *WEIGHT(m, c)* of all checkers in *CE* (line 14). These values will obviously be different from the previously computed ones, *REACH_PROB(n, c)* and *WEIGHT(n, c)*. To be more exact they will increase as we approach *HOST(c)*.

We also compute the *path length* of each checker (denoted with *PATH_LENGTH(n, c)*). This represents the length (in time) of the path that leads from the current node *n* to *HOST(c)* in the CFG. Please note that all the checkers active along that path will have assigned an implementation (either SW or HW) for the current traversal of the ECFG, so we can compute (line 15) the length of the path (from the current node up to the location of the checker):

$$PATH\_LENGTH(n,c) = \sum_{\substack{m \in PATH(c),\\ m \text{ successor of}\\ n \text{ on } PATH(c)}} \left(TIME(m) + \sum_{\substack{k \in CE, k \neq c,\\ HOST(k)=m,\\ k \text{ active}}} OVERHEAD(k) \right)$$

$$\text{where } OVERHEAD(k) = \begin{cases} SW(k), & \text{if } k \text{ is implemented in software} \\ WAIT(k) + HW(k), & \text{if } k \text{ is implemented in hardware} \end{cases}$$

and WAIT(k) = eventual waiting time introduced due to the fact that reconfiguration is not finished when the checker is reached.

We denote with *ACE(e)* the set of currently active checking expressions (checkers) if edge *e* is taken in the ECFG, i.e.:

ACE(e) = {c ϵ CE | e ϵ PATH(c) Λ
AREA(c) fits in currently available FPGA area Λ
REC(c) + HW(c) < PATH_LENGTH(n, c) + SW(c) Λ
path from m to HOST(c) does not contain any back edge}.

We discard all the checkers for which we do not currently have enough contiguous FPGA area available for a feasible placement, as well as the ones for which it is actually more profitable (at this point) to leave them in software (because they are too close to the current location and reconfiguring and executing the checker in hardware would actually take longer time than executing it in software). We also don't start reconfigurations for checkers over a back edge, because we reset the context on back edges; thus, if a reconfiguration would be running, it would be stopped, and this might possibly lead to a wasteful use of the reconfiguration controller.

The next step after building *ACE(e)* (line 16) is to check if the reconfiguration controller is available, in order to schedule new reconfigurations (line 17). We distinguish three cases:

1. the reconfiguration controller is free. In this case we can simply proceed and use it;
2. the reconfiguration controller is busy, but it is currently reconfiguring a checker that is not reachable anymore from the current point (we previously started a speculative reconfiguration and we were wrong). In this case we can drop the current reconfiguration, mark the FPGA space corresponding to the unreachable checker as free and use the reconfiguration controller.
3. the reconfiguration controller is busy configuring a checker that is still reachable from the current point. In this case we leave the reconfiguration running and schedule new reconfigurations only in case the current reconfiguration will finish before the next basic block will finish. Otherwise, we can take a better decision later, on the outgoing edges of the next basic block.

In case $ACE(e) \neq \emptyset$ we start new reconfigurations (line 18) by calling the procedure **schedule_new_reconfigurations** (described below). Otherwise, we do not take any reconfiguration action. Then we continue traversing the ECFG (line 19).

6.2 FPGA Area Management

When we start a new reconfiguration, we choose a subset of checkers from *ACE(e)* so that they fit on the currently available FPGA area, trying to maximize their total weight (lines 22-29).

Let us next describe how the FPGA space is managed. After a checker is placed on the FPGA, the corresponding module is marked as active and that space is kept occupied until the checker gets executed. After this point, the module is marked as inactive but it is not physically removed from the FPGA. Instead, if the same checker is needed later (e.g. in a loop), and it has not been overwritten yet, we simply mark it as active and no reconfiguration is needed, since the module is already on the FPGA (lines 23-24). In case the module is not currently loaded on the FPGA we need to choose an FPGA location and reconfigure it. We distinguish two cases: (1) We cannot find enough contiguous space on the FPGA, so we need to pick one inactive module to be replaced. This is done using the least recently used policy (line 34). (2) There is enough contiguous free FPGA space, so we need to pick a location (line 32). The free space is managed by keeping a list of maximal empty rectangles; we pick the

location that generates the lowest fragmentation of the FPGA. The fragmentation metric used to take this decision is described in [5]. The basic idea is to compute the fragmentation contribution of each cell (FCC) in an empty area (that can fit our module) as:

$$FCC_d(C) = \begin{cases} 1 - \frac{v_d}{2L_d - 1}, \text{ if } v_d < 2L_d \\ 0, \quad otherwise \end{cases}$$

where v_d represents the number of empty cells in the vicinity of cell C and L_d is the average module size in direction d of all modules placed so far. We assume that if an empty rectangle can accommodate a module as large as twice the average size of the modules placed so far, the area inside that rectangle is not fragmented. The total fragmentation of the empty area can be computed as the normalized sum of fragmentation (FCC) of all the cells in that area.

After deciding on a free location, we mark the reconfigurations in the schedule table as active on the previously built condition (line 27). We stop as soon as the sum of reconfiguration times for the selected subset of checkers exceeds the execution time of the next basic block (line 30). For all the other checkers we can take a better decision later, on the outgoing edges of the next basic block.

7. EXPERIMENTAL RESULTS

In order to evaluate our algorithm we first performed experiments on synthetic examples. We randomly generated control flow graphs with 100, 200 and 300 nodes (15 CFGs for each application size). The execution time of each node was randomly generated in the range of 10 to 250 time units. We then instrumented each CFG with checkers (35, 50 and 75 checkers for CFGs with 100, 200 and 300 nodes, respectively). For each checker, we generated an execution time corresponding to its software implementation, as well as execution time, area and reconfiguration overhead corresponding to its hardware implementation. The ranges used were based on the overheads reported in [12] for this error detection technique.

The size of the FPGA available for placement of error detection modules was varied as follows: we sum up all the hardware areas for all checkers of a certain application:

$$MAX\_HW = \sum_{i=1}^{card(CE)} AREA(c_i)$$

Then we generated problem instances by considering the size of the FPGA corresponding to different fractions of *MAX_HW*: 3%, 5%, 10%, 15%, 20%, 25%, 40%, 60%, 80% and 100%. As a result, we obtained a total of 3 × 15 × 10 = 450 experimental settings. All experiments were run on a PC with CPU frequency 2.83 GHz, 8 GB of RAM, and running Windows Vista.

We compared the results generated by our optimization algorithm (OPT) with a straight-forward implementation (SF), which statically places in hardware those expressions that are used most frequently and give the best gain over a software implementation, until the whole FPGA is occupied. Module placement is done according to the fragmentation metric (FCC) described in Section 6.2.

In order to compute our baseline, we considered that all checkers are implemented in hardware (for a particular CFG) and then calculated the execution time of the application ($EX_{HW\text{-}only}$). We also calculated $EX_{SW\text{-}only}$ considering that all checkers are implemented in software. For the same CFGs, we then considered the various hardware fractions assigned and for each resulting FPGA size we computed the execution time after applying our optimization, EX_{OPT}, and after applying the straight-forward approach, EX_{SF}. In order to estimate the average execution time corresponding to a CFG (line 6 in Figure 2), we used a modified version of the methodology described in [16], adapted to take into account reconfiguration of checkers.

For a particular result, *EX*, we define the normalized distance to the HW-only solution as:

$$D(EX) = \left(\frac{EX - EX_{HW-only}}{EX_{SW-only} - EX_{HW-only}} \times 100 \right)\%$$

This distance gives a measure of how close to the HW-only solution we manage to stay, although we use less HW area than the maximum needed.

Table 3. Time and area overheads for checkers

Check	Gain (µs)	Area (slices)	Rec. (µs)	Check	Gain (µs)	Area (slices)	Rec. (µs)
C1	0.8	6	3.69	C29	0.7	16	9.84
C2	0.71	8	4.92	C30	0.66	13	7.995
C3	0.725	7	4.305	C31	0.515	9	5.535
C4	0.59	2	1.23	C32	1	18	11.07
C5	0.88	3	1.845	C33	1.02	22	13.53
C6	0.45	2	1.23	C34	0.795	17	10.455
C7	0.675	6	3.69	C35	0.82	18	11.07
C8	0.665	6	3.69	C36	0.78	16	9.84
C9	0.82	4	2.46	C37	0.6	19	11.685
C10	1.005	9	5.535	C38	0.625	21	12.915
C11	0.46	6	3.69	C39	0.9	32	19.68
C12	0.67	7	4.305	C40	1.35	45	27.675
C13	0.625	11	6.765	C41	1.34	40	24.6
C14	0.7	13	7.995	C42	1.01	30	18.45
C15	0.73	15	9.225	C43	1.02	40	24.6
C16	0.67	10	6.15	C44	1	35	21.525
C17	1.1	15	9.225	C45	1.01	26	15.99
C18	0.905	15	9.225	C46	1.05	32	19.68
C19	0.91	12	7.38	C47	1.4	34	20.91
C20	0.915	13	7.995	C48	1.45	33	20.295
C21	0.7	14	8.61	C49	1.605	37	22.755
C22	0.62	12	7.38	C50	1.56	37	22.755
C23	0.58	10	6.15	C51	0.5	10	6.15
C24	1.01	16	9.84	C52	0.79	16	9.84
C25	0.705	12	7.38	C53	0.78	14	8.61
C26	0.735	12	7.38	C54	0.7	16	9.84
C27	0.5	14	8.61	C55	0.715	18	11.07
C28	0.96	9	5.535	C56	0.8	22	13.53

In figures 3a and 3b we compare the average $D(EX_{OPT})$ with $D(EX_{SF})$ over all testcases with 100 and 300 nodes[2]. The setting with 0% HW fraction corresponds to the SW-only solution. It can be seen, for both problem sizes, that our optimization gets results within 18% of the HW-only solution with only 20% of the maximum hardware needed. For big fractions the straight-forward solution (SF) also performs quite well (since there is enough hardware area to place most of the checkers with high frequencies), but for the small hardware fractions our algorithm significantly outperforms the SF solution. As far as the running time of our optimization is concerned, Figure 3d shows how it scales with the number of nodes in the CFG. Finally, Figures 3e and 3f present the size of the reconfiguration tables (number of entries), for testcases with 100 and 300 nodes, for each hardware fraction considered.

Case study

We also tested our approach on a real-life example, a GSM encoder, which implements the European GSM 06.10 provisional standard for full-rate speech transcoding. This application can be decomposed into 10 tasks executed in a sequential order: Init, GetAudioInput, Preprocess, LPC_Analysis, ShortTermAnalysisFilter, LongTermPredictor, RPE_Encoding, Add, Encode, Output. We instrumented the whole application with 56 checkers, corresponding to the 19 most critical variables, according to the technique described in Section 2.1. The execution times were derived using the MPARM cycle accurate simulator, considering an ARM processor with an operational frequency of 60 MHz. The checking modules were synthesized for an XC5VLX50 Virtex-5 device, using the Xilinx ISE WebPack. The reconfiguration times were estimated considering a 60 MHz configuration clock frequency and the ICAP 32-bit width configuration interface. We used a methodology similar to the one presented in [18] in order to reduce the reconfiguration granularity. The gain, area and reconfiguration overheads for each checker are given in Table 3. The CFGs for each task, as well as the profiling information was generated using the LLVM suite [8] as follows: *llvm-gcc* was first used to generate LLVM bytecode from the C files. The *opt* tool was then used to instrument the bytecode with edge and basic block profiling instructions. The bytecode was next run using *lli*, and then the execution profile was generated using *llvm-prof*. Finally, *opt-analyze* was used to print the CFGs to *.dot* files. We ran the profiling considering several audio files (*.au*) as input. The results of this step revealed that many checkers (35 out of the 56) were

[2] Since the results for 200 nodes CFGs were similar, we omit them here due to space constraints.

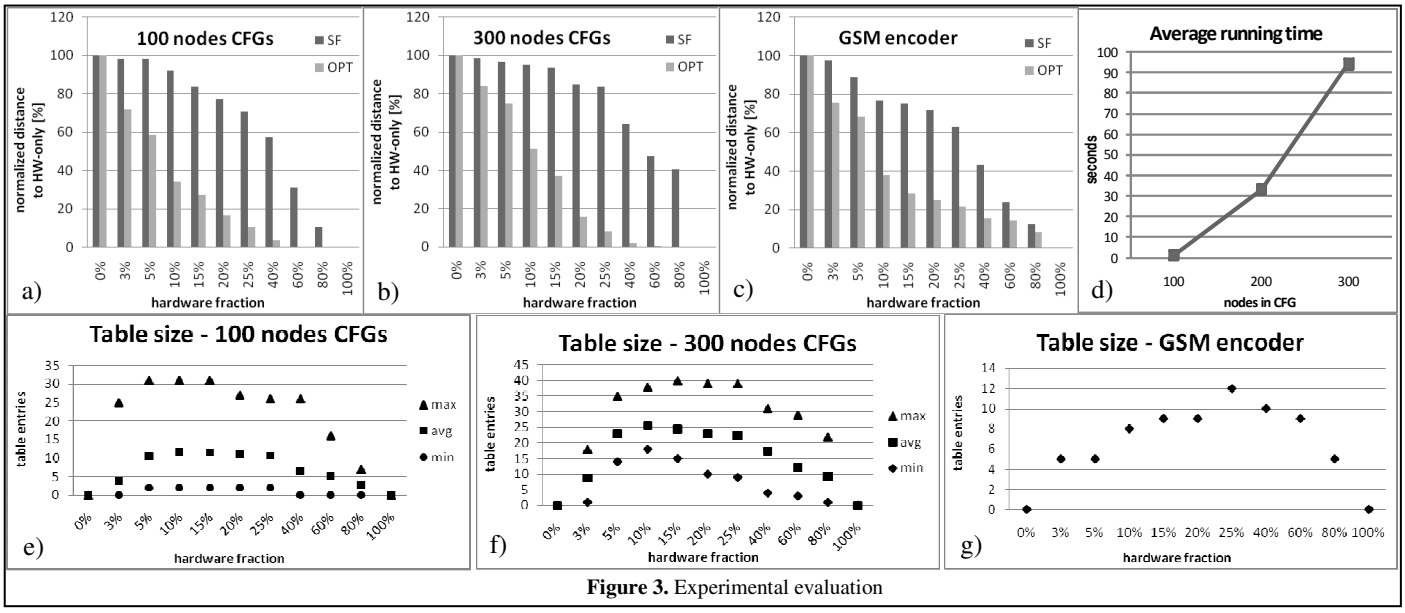

Figure 3. Experimental evaluation

placed in loops and executed on average as much as 375354 times, which suggests that it is important to place them in HW.

Using the information generated we ran our optimization algorithm. The results obtained are shown in Figure 3c. It can be seen that the results follow trends similar to those for the synthetic experiments. Our optimization generated results within 27% of the HW-only solution with just 15% hardware fraction. Also note that for hardware fractions between 15% and 40%, the solution generated by our algorithm (OPT) was roughly 2.5 times closer to the HW-only solution than that generated by the straight-forward algorithm (SF). Finally, Figure 3g shows the sizes for the reconfiguration tables.

8. CONCLUSION

This paper presented an algorithm for performance optimization of error detection based on speculative reconfiguration. We managed to minimize the average execution time of a program by using partially reconfigurable FPGAs to place in hardware only those error detection components that provide the highest performance improvement. One direction of future work would be to extend this approach to inter-procedural analysis as well. Another direction would be to extend the work to a more flexible solution, that uses online learning to adapt the system to eventual changes (e.g. in the inputs or in the environment).

REFERENCES

[1] Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 2006.

[2] 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.

[3] Constantinescu, C. "Trends and challenges in VLSI circuit reliability". *IEEE Micro*, 23, 4 (2003), 14-19.

[4] 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.

[5] Handa, M. and Vemuri, R., "Area Fragmentation in Reconfigurable Operating Systems". *Intl. Conf. on Engineering of Reconfigurable Systems and Algorithms*, 2004, 77–83.

[6] 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 systems". *ACM Trans. Embed. Comput. Syst.*, 8, 4 (2009), 1-30.

[7] Jean, J. S. N., Tomko, K., Yavagal, V., Shah, J. and Cook, R. "Dynamic reconfiguration to support concurrent applications". *IEEE Transactions on Computers*, 48, 6 (1999), 591-602.

[8] Lattner, C. and Adve, V. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". *Intl. Symp. on Code Generation and Optimization, 2004*, 75 - 86.

[9] Li, Z. and Hauck, S. "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation". *Intl. Symp. on Field-Programmable Gate Arrays.* Monterey, CA, USA, 2002, 187-195.

[10] Lifa, A., Eles, P., Peng, Z. and Izosimov, V. "Hardware/Software Optimization of Error Detection Implementation for Real-Time Embedded Systems". *CODES+ISSS'10,* Scottsdale, AZ, USA, 2010, 41-50.

[11] Lima, F., Carro, L. and Reis, R. "Designing fault tolerant systems into SRAM-based FPGAs". *Design Automation Conference,* Anaheim, CA, USA, 2003, 650-655.

[12] Lyle, G., Chen, S., Pattabiraman, K., Kalbarczyk, Z. and Iyer, R. "An end-to-end approach for the automatic derivation of application-aware error detectors". *Dependable Systems & Networks, 2009,* 584-589.

[13] Panainte, E., Bertels, K. and Vassiliadis, S. "Interprocedural Compiler Optimization for Partial Run-Time Reconfiguration". *The Journal of VLSI Signal Processing*, 43, 2 (2006), 161-172.

[14] Pattabiraman, K., Kalbarczyk, Z. and Iyer, R. K. "Application-based metrics for strategic placement of detectors". *Pacific Rim Intl. Symp. on Dependable Computing, 2005.*

[15] 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.

[16] Sarkar, V. "Determining average program execution times and their variance". *SIGPLAN*, 24, 7 (1989), 298-312.

[17] Schuck, C., Kuhnle, M., Hubner, M. and Becker, J. "A framework for dynamic 2D placement on FPGAs". *Intl. Symp. on Parallel and Distributed Processing, 2008*, 1-7.

[18] 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.

[19] Sim, J. E., Wong, W., Walla, G., Ziermann, T. and Teich, J. "Interprocedural Placement-Aware Configuration Prefetching for FPGA-Based Systems". *Symp. on Field-Programmable Custom Computing Machines*, 2010.

[20] 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*, Washington, DC, USA, 2003, 133.

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