IEICE TRANS. INF. & SYST., VOL.E96–D, NO.8 AUGUST 2013

PAPER Special Section on Reconfigurable Systems

Selective Check of Data-Path for Effective Fault Tolerance

Tanvir AHMED[†a)], ***Student Member***, **Jun YAO**[†], **Yuko HARA-AZUMI**[†], ***Members***, **Shigeru YAMASHITA**[††], ***Senior Member***, ***and*** **Yasuhiko NAKASHIMA**[†], ***Member***

SUMMARY Nowadays, fault tolerance has been playing a progressively important role in covering increasing soft/hard error rates in electronic devices that accompany the advances of process technologies. Research shows that wear-out faults have a gradual onset, starting with a timing fault and then eventually leading to a permanent fault. Error detection is thus a required function to maintain execution correctness. Currently, however, many highly dependable methods to cover permanent faults are commonly over-designed by using very frequent checking, due to lack of awareness of the fault possibility in circuits used for the pending executions. In this research, to address the over-checking problem, we introduce a metric for permanent defects, as operation defective probability (ODP), to quantitatively instruct the check operations being placed only at critical positions. By using this selective checking approach, we can achieve a near-100% dependability by having about 53% less check operations, as compared to the ideal reliable method, which performs exhaustive checks to guarantee a zero-error propagation. By this means, we are able to reduce 21.7% power consumption by avoiding the non-critical checking inside the over-designed approach.

key words: *low power, fault-tolerant computing, FU array*

1. Introduction

With the improvement in CMOS technology, the size of semiconductor devices is shrinking rapidly, leading to many advantages in modern microprocessor design such as low energy consumption per transistor, low manufacturing cost, high operating frequency, and high density of transistors. However, at the same time, such benefits make the transistors more vulnerable to fault attacks. Specifically, high operating frequency and chip-level overheat accompanying the shrinking of the technology also increase the potential wear-out rates of transistors and inner-connections. As a result, the lifetime of the microprocessors and digital circuits becomes shorter and less predictable [1]–[4]. Moreover, a recent study on the memory system of different data-centers shows that 73% errors are from permanent faults [5]. Fault-tolerant techniques are thus necessary to guarantee execution correctness and keep technology advancing.

Error check and correction (ECC) is a conventional fault tolerant technique. Although ECC logic has been effectively working for data-center memory systems [5], it may be not suitable for reconfigurable systems with many FUs, which mainly consist of combinational logic. In order to detect the faults in combinational logic, on-line tests [6], [7] and explicit checks [8] have been used. On-line test [6] is not real-time, while the real-time explicit check [8] increases power consumption continuously and visibly. Figure 1 (a) gives an example of exhaustive check to keep a full understanding of defective units in the data-path. The thorough check adds pressure to the power utilization limitation, and may be an over-design by not considering the criticality of faults in different operations. An alternative way is a light-weighted technique, which add only one check at the end of the data-path (Fig. 1 (b)). However, it allows the tainted data from the early stage, such as `I1'` in Fig. 1 (b), to propagate inside the path. The erroneous data may taint more data, which possibly downgrades the reliability and becomes unpredictable when a second fault occurs.

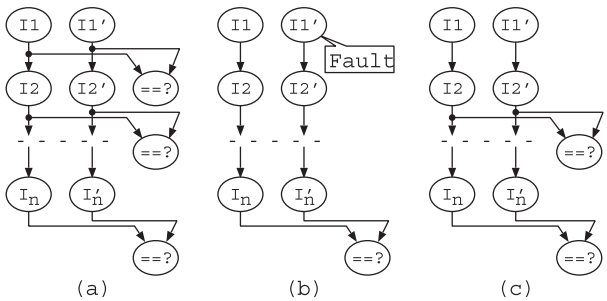


Fig. 1 DMR-based permanent fault locating: (a) Exhaustive check technique, (b) Light-weighted technique, and (c) Selective check based technique.

Unlike both Fig. 1 (a) and Fig. 1 (b), in this research we propose to insert selective checks into the data-path based on the operation defective probability (ODP), as shown in Fig. 1 (c). The ODP of an operations is calculated according to the number of gates the operation uses. The branch of the data-flow-graph (DFG) with ODP larger than a threshold is regarded as a potential candidate to contain permanent faults, and a check instruction will be inserted at that branch to help a deterministic verification. Furthermore, our ODP calculation also takes the influence of special inputs into account, i.e., certain inputs of circuits can put parts of the circuits into *don't care zones*. Potential errors in these gates will not contribute to the final potential error rate and are thus carefully removed from the influence chain, in order to further reduce the number of check operations without af-

Manuscript received November 9, 2012.
Manuscript revised March 8, 2013.

[†]The authors are with the Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma-shi, 630–0192 Japan.
[††]The author is with the College of Information Science and Engineering, Ritsumeikan University, Kusatsu-shi, 525–8577 Japan.
a) E-mail: tanvir-a@is.naist.jp

DOI: 10.1587/transinf.E96.D.1592

fecting the dependability.

The results show that our proposed technique achieves near-optimal dependability by reducing the checking instructions to 60%. By allowing a slight increase of the error propagation rate (2%), our approach is able to further reduce the number of check instructions to a 37% level, as compared to the exhaustive check method. The removal of non-critical check instructions can contribute to the energy saving in the dependable execution. Under the allowance of 2% error propagation rate execution, 22% energy can be saved by avoiding the thorough checking.

The rest of this paper is organized as follows. The proposed method of optimizing the cost for locating a permanent fault is discussed in Sect. 2. Section 3 shows the results of the proposed approach. Section 4 concludes the paper.

2. Our Proposal: Constructing an ODP-Aware Data-Path

2.1 Selective Check Instructions in Redundant Data-Path

In this research, we mainly focus on reconfigurable architectures [9], [10] which contain a large pool of resources to exploit extreme parallelism. We use LAPP [11], [12] as our baseline architecture, which is a reconfigurable architecture containing an array of FUs for accelerating image processing applications. LAPP consists of a large set of combinational units and networks to which ECC protection cannot be applied as easily as in the data-center research [5] with an acceptable cost. Several architectural methods have already been proposed to achieve a relatively effective fault tolerance in combinational logic. Specifically, dual modular redundancy (DMR) with a check after the dual executions can guarantee that no soft error goes undetected [8]. As reconfigurable architectures such as LAPP [11], [12] are originally rich in resources, DMR with checks from software level can be applied flexibly and efficiently with the understanding of the DFG inside the architecture.

To detect the permanently defective unit inside the DMR execution, the straightforward method is to exhaustively check all the instruction executions in order to gain the information of all units used, as have been shown in Fig. 1 (a). After that, on-line test or tuning can locate the defective unit at that erroneous spot. An alternative way is to add one check instruction at the end of data-path (Fig. 1 (b)). However, it downgrades the dependability as described in Sect. 1. In this research, we use selective check instructions together with DMR to create correct executions and locate the possible erroneous spots. Figure 2 gives a brief introduction of this method.

Figure 2 gives the algorithm to put the duplicated DFG into the FU array. During the mapping of the DFG, basically, each instruction will be duplicated by `map(i,i')`. Along with the mapping, we also study the vulnerability of each instruction by using `get_vulnerability()`, which provides the possibility of permanent fault inside this hardware unit. Later sections will give a detailed explanation

```
sum = 0; /* Accumulated vulnerability in data-path */
last_need_check = 0; last_inst = NOP;

while (!end(DFG)) {
    /* To map data-flow-graph (DFG) in FU array */
    i = fetch_inst(PC);

    /* Duplicate & selective check in map */
    if (last_need_check)
        map(i, i', check(last_inst));
    else
        map(i, i');

    /* Analyze vulnerability of instruction */
    sum += get_vulnerability(i);
    if (sum > th) {
        /* Critical spot */
        last_need_check = 1;
        sum = 0;
    }
    else
        last_need_check = 0;
    last_inst = i;

    PC++;
} /* end of this cycle */
```

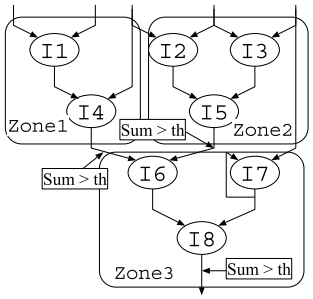
Fig. 2 Algorithm to selectively add check instruction.

of `get_vulnerability()`. Only when the accumulated vulnerability '`sum`' becomes larger than a predetermined threshold '`th`', a check instruction will be added, following `map(i,i',check(last_inst))`, where '`last_inst`' is the instruction in previous cycle. '`last_inst`' is used in the check because the results of the duplicated instructions will be known in this cycle. By this way, the data-path in Fig. 1 (c) is obtained.

By introducing a selective check according to ODP, the number of check instructions is reduced, as shown in Fig. 3 (b). With the help of selective check instructions, only the segment with an error report needs to be checked in detail for the defective unit. For example, in Fig. 3 (b) the `chk-I5` is the first check instruction to report an error, since `I2` is mapped on a defective unit. Thus, only the dependent instructions (in Fig. 3 (a)), `I2` and `I3`, need to be checked to locate the permanent failure as shown in Fig. 3 (c). At the same time, due to the previous checks, the instructions inside other segments are judged to be previously mapped inside the correct units. Therefore, they need to be inside the DFG only for the completion of the data-path. No DMR execution is necessary for these instructions, as is shown in Fig. 3 (c). Consequently, the number of check and redundant instructions is reduced significantly during the permanent defect location. Although the period of locating permanent fault in Fig. 1 (a) is short, we are not increasing 1/3 power consumption by avoiding exhaustive checking. The burst of power hungry mode as in Fig. 1 (a) is thus avoided.

2.2 Calculation of Defective Probability

Mukherjee and et al. [13] have stated that various programs



(a)

Stage	FU0	FU1	FU2
0	I1	I1′	
1	I2 (x)	I2′	
2	I3	I3′	
3	I4	I4′	
4	I5	I5′	chk-I4
5	I6	I6′	chk-I5
6	I7	I7′	
7	I8	I8′	
8			chk-I8

(b)

Stage	FU0	FU1	FU2
0	I1		
1	I2	I2′	
2	I3	I3′	chk-I2
3	I4		chk-I3
4	I5	I5′	
5	I6		chk-I5
6	I7		
7	I8		
8			

(c)

Fig. 3 Cost-effective permanent fault locating: (a) DFG, (b) DMR mode with reduced check instructions, (c) Locating a permanent fault after `chk-I5` reports error.

will respond differently to the same fault rate, according to their different architectural vulnerability factors (AVFs). AVF gives a measure of the probability that a fault will turn into a visible error. Given that the soft error occurs in a certain memory block, it will become an error only when the latter calculation depends on this faulty block. The AVF of a program depends on its working behaviors, especially the memory access intensity and frequency.

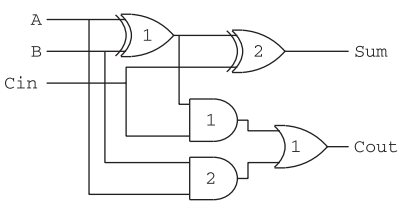
Similarly, we are using the idea of operation defective probability (ODP) in this research to selectively add data verification instructions. We extend the above AVF, which is for soft errors only, to handle permanent faults. In this research, we treat the permanent fault occurrence probability as in proportion to the gate number inside the functional unit. For example, a 1-bit AND operation requires two 2-input NAND gates, while a 1-bit XOR operation uses four 2-input NAND gates. As a result, the lifespan of the XOR will be relatively shorter than the AND unit under a given gate defect ratio. Applying the consideration to arithmetic operations, the defective probability to a permanent defect may be even larger due to the large area and the complex wire interconnections inside the arithmetic operations. For example, a 1-bit full adder takes nine 2-input NAND gates to finish the calculation. A large word-length multiplication uses several stages of adder chains and partial product generators. These units are thereby vulnerable to soft error and permanent defects because of their large hardware areas and relatively long data paths.

We give ODP to permanent defects in Table 1 by studying the typical operations from the FRV Instruction-set architecture [14]. As shown in Table 1, these operations can be categorized into four types: logic, arithmetic, media and memory. Although the FUs for these operations may merge and share circuits between operations to achieve the optimized design in both power and area, in this research, we implement these operations into separated units, from the viewpoint that each operation takes an independent and individual path in the FUs and for every calculation, only the corresponding path is activated. The FUs are implemented into Verilog HDL modules and then synthesized by Design Compiler with a 180 nm cell library to obtain the area in the number of gates, as listed in Table 1. In addition, we treat AND operation as having a defective probability of 0.1%[†]. The other vulnerable probabilities are thus calculated by $0.1\% \times \frac{\text{Area}_{\text{OP}}}{\text{Area}_{\text{AND}}}$. As discussed above, logic operations are relatively less complex in hardware than other units, and their ODPs are thus relatively small. The arithmetic instructions take medium ODPs, except for the very large multiplication unit whose ODP reaches 2.9%. The media operations are a combination of logic and arithmetic operations and thus tend to show large ODPs. Finally, for LOAD instruction, it has been assumed that the memory is protected with error correcting code (ECC) so that the loaded data can be regarded as error free results. The only vulnerability in LOAD comes from the address calculation part which is the same as the ADD operation. To guarantee that there will be no tainted value by faults to the data storage, in our high

Table 1 ODP of the baseline ISA for this study.

Types of Operations	Operations	Number of Gates	Defective Probability (%)
Logic	AND	176	0.10
	OR	176	0.10
	XOR	208	0.12
	SLL/SRL	1,020	0.58
Arithmetic	ADD	892	0.50
	SUB	1,022	0.58
	MUL	5,130	2.90
	SLA	1,005	0.57
Media	MSRL	792	0.45
	BYTE-HALF	219	0.12
	SUML/H	996	0.57
	HALF BYTE	854	0.48
	SAD	2,970	1.69
	UADD	1,320	0.75
	USUB	1,398	0.86
	MUL	2,569	1.46
Memory	LOAD	892	0.50
	STORE	892	0.0

[†]Although the value may be far larger than practical meanings, we simply use it here to demonstrate how our methods work accordingly to these assumed ODP values.



Inputs	Gate Usage XOR	AND	OR
X 0 0	2	0	0
X 0 1	2	1	1
X 1 0	2	1	1
X 1 1	2	2	1

Fig. 4 NAND gate error analysis for constant Inputs.

dependable LAPP, the STORE operation is originally designed to take checked data before the real commitment, by embedding a check instruction inside it to check both data and address. For this reason, although the address calculation part of STORE still contains 892 gates, the ODP of this address calculation will be updated to 0% as the in-store check determines the correctness of the address and makes the defective possibility to 0%.

2.3 Optimization of the ODP for Constant Input

In Sect. 2.2, we calculated the ODP of a particular operation based on the size of the circuit. Due to the behavior of certain gates such as NAND2, NXOR2, the sensitivity to faults—measured as ODP in our approach—can be further reduced when a constant value is provided to an operation. As an example, given a constant LOW input in an AND gate, it will always have the LOW output. Thus, any fault (stuck at 0 or stuck at 1) on the other input wire of this AND gate is masked and turned into don't care. Similarly, with an input provided to be logic HIGH, an OR gate can ignore the sensitivity of faults in its other input ports. As a result, compared to operations with variable inputs from prior circuits or register files, an operation with an immediate source operand will demonstrate less sensitivity to faults, since part of its sub-circuits can be logically masked into don't care zones which contribute 0% ODP.

Rather than counting the insensitive inputs, we use don't care gates by the constant value in order to update the ODP with immediate operand in our approach. Figure 4 shows the method to calculate the sensitive gates of an adder while one input is constant. In Fig. 4, the gate-level schematic of the adder and a table containing the gate usages for different constant input patterns have been shown. For the pattern <X, 0, 0> = <A, B, Cin> the input 'A' is a variable and 'B' and 'Cin' are LOW. Two XOR gates are used for calculating the sum. However, two AND gates can be replaced with wires since both have a constant LOW input. Again, similarly for the input pattern <X, 1, 0> AND gate 1 can be ignored. This illustrates the sensitive gates for a constant input has been reduced. Accordingly, we change the ODP calculation from $0.1\% \times \frac{\text{Area}_{\text{OP}}}{\text{Area}_{\text{AND}}}$ to $0.1\% \times \frac{\text{Area}_{\text{OP}} \times \text{S(\#imm)}}{\text{Area}_{\text{AND}}}$, where S(#imm) represents the ratio of sensitive circuit area under the given #imm inputs.

Based on the above assumption, the sensitive gates are calculated for the operations of our baseline ISA. The average of different constant values of the benchmark functions for an operation is included in Fig. 5. Specifically, the operations with large areas under variable inputs can be significantly reduced to a level of halved or even smaller circuit areas. The originally small-sized operations such as AND, OR and BYTE-HALF do not show visibly important area reduction, due to their relatively less complex implementation. All these results thus give a more accurate estimation of the ODP inside a real program, which is able to help reduce further the number of exhaustive checks and lower the resource utilization. The approach to calculate the ODP of operations in data-flow-graph (DFG) will be introduced in the next subsection by using the updated ODPs.

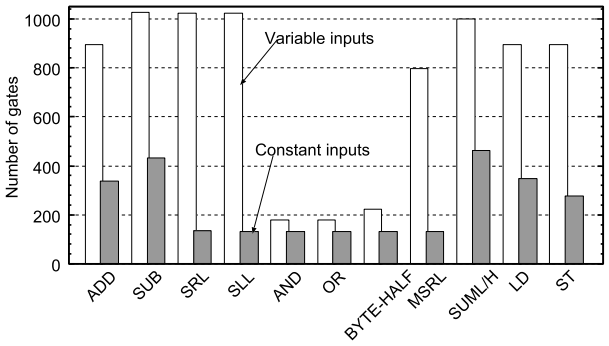


Fig. 5 Area required for the constant input operations.

2.4 Adding Check According to ODP

Table 1 shows the defective probability of each instruction. Assuming that each operation takes two source operands and produces one result, we can design `get_vulnerability()` in Fig. 2 by calculating the probability of the error of the result as follows:

$$1 - \Pr(\text{out}) = (1 - \Pr(\text{op})) \prod_{i=1}^{2} (1 - \Pr(s_i)) \quad (1)$$

$\Pr(s_1)$ and $\Pr(s_2)$ are the probabilities of errors in the source operands, while Pr(op) is the error probability coming from the operation itself. It can be imagined that the Pr(op) has a direct connection to the defective probability in Table 1, augmented with the insensitivity from constant inputs in Sect. 2.3. Assume that the whole data path starts from several checked inputs such as values from the ECC-ed register file which has 0% probability of error. The output of the first operation will have a probability of error regarding the operation itself. The successive dependent data will inherit this probability of error and adds a new probability when the data goes forward through the data flow graph. Although the values of defective probability in Table 1 are actually much larger than a practical probability of error, we are still directly using these values as Pr(op) in the remaining parts of this paper to introduce the idea. By this means, we are able to tag the results with the probability of permanent error inside the whole data path.

Figure 6 gives a detailed illustration of using ODP in `get_vulnerability()` in algorithm of Fig. 2. For simplicity, we assume that the threshold of error probability is

1.0%. The data flow graph starts by taking inputs R1, R2, R3 and R4 from the register file or memory, which are previously checked results and protected by ECC. It ends by committing the final result R6 into the memory. Basically, every operation will be doubly executed and the final result R6 will be compared before being written to memory.

In Fig. 6, the defective probability of each operation is shown inside the operation. After stage 2, both R1 and R2 get error probabilities that exceed the threshold. The check instruction is thus added to make a fast determination whether or not an error has happened there. This also covers Zone1 and Zone2, as shown in Fig. 6.

It is possible that the data-path will take backward bypassing data, such as OP7(R4+=R5) in Fig. 6. Considering that this data path represents a loop body, operation R4+=R5 takes its first operands from the register file in the first iteration and updates itself afterward. For the first iterations, the vulnerability of the output of OP7 denoted as Pr(R4[1]), is calculated as previous Eq. (1). From the second iteration using Bayes' theorem the correctness of the output of OP7, as Pr(R4[2] correct) is calculated as:

$$\begin{aligned}\Pr(\text{R4[2] correct}) =& \Pr(\text{R5, OP7 correct})\times \\ & \Pr(\text{R4[1] correct} \mid \text{R5, OP7 correct})\end{aligned} \tag{2}$$

However, as R4[1] depends on R4[0], R5 and OP7, when R5, OP7 are correct, the only dependency becomes R4[0]. For this reason, we can have:

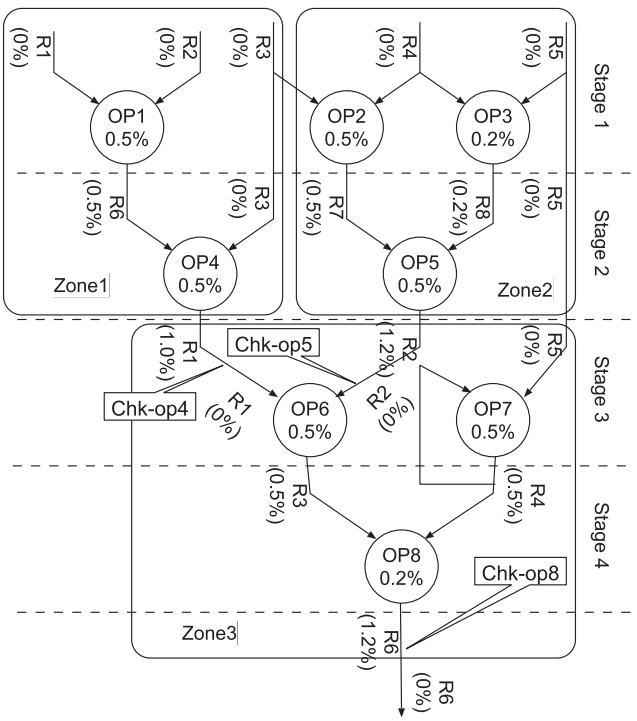


Fig. 6 Adding check instruction on a DFG based on Eq. (1).

$$\Pr(\text{R4[1] correct} \mid \text{R5, OP7 correct}) = \Pr(\text{R4[0] correct}) \tag{3}$$

Using Eq. (3) in Eq. (2) we can have

$$\begin{aligned}\Pr(\text{R4[2] correct}) &= \Pr(\text{R4[0] correct})\times\Pr(\text{R5, OP7 correct}) \\ &= \Pr(\text{R4[1]correct})\end{aligned} \tag{4}$$

Therefore, we can expect that from the second iteration Pr(R4[n]) = Pr(R4[n − 1]). We thus fixed the possibility of faults in loop-back R4 by the above means.

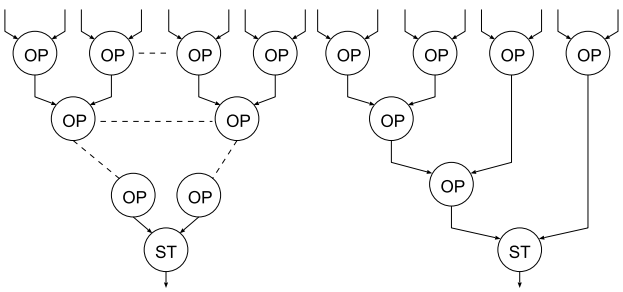
3. Results

3.1 Workloads and Characteristics

In this section, we present the results of our proposed technique based on ODP-aware selective checking. We tried our techniques on eight image filter functions. The size and the number of independent tree of the functions are described in Table 2. Figure 7 show three different types of data flow graph in FU array. There is a long data graph in Fig. 7 (a), and a small data path in Fig. 7 (b). Both of them have a final single output to the memory. Figure 7 (c) gives a data graph with many independent branches, where all the branches are

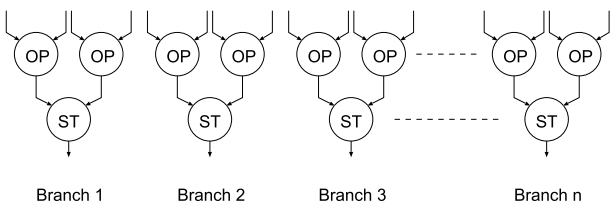
Table 2 Benchmark programs.

Functions	Number of Instructions	Number of Independent Trees	Types of Program
Expand4k	73	2	
Unsharp	63	1	
Wdifline	57	1	*A*
FI-1	40	1	
Blur	20	1	
FI-3	14	1	*B*
Tone	12	1	
FI-2	68	8	*C*



(a) A long data-path with many instructions.

(b) A small data-path with few instructions.



(c) A program with many independent branches.

Fig. 7 Three kinds of data flow graphs.

Table 3 Simulator specification.

Stage0	
Decode Width	max. 8 inst./cycle
General Register	32
Media Register	32
Data transfer speed (with ext. cache)	8 bytes/cycle
Instruction Cache	4 ways 16 KB (64 byte/line)
L1 Data Cache	4 ways 16 KB (64 byte/line)
L1$ → L0$ Data Transfer Rate	16 bytes/cycle
Store Buffer	4 entries
50 Stages	
Number of FUs	200 (4 FUs × 50 rows)
Instruction Mapping Speed	4 ways 256 B (16 bytes/cycle)
Inter L0$ Data propagation rate	16 bytes/cycle
L1$→L0$ Data transfer rate	16 byte/cycle
L0$→LSU Data transfer rate	4 byte/cycle
Store Buffer	1 entry

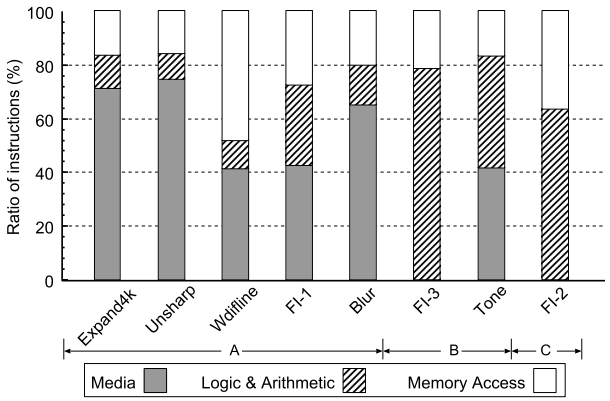


Fig. 8 Incidence of instructions.

end up in writing results to memory. The benchmark functions are classified as Type A, Type B and Type C according to Fig. 7 (a), 7 (b) and 7 (c), respectively.

We used a cycle-accurate architectural simulator [12] to get the energy data of the executions of the programs with the selective check instructions. The parameters of the baseline processor in the architectural simulator are listed in Table 3. Specifically, power data of each unit in the baseline processor has been obtained by the PrimeTime with a 180 nm cell library, under the working condition of a 1.8 V supply voltage. Together with the utilization of each unit, which is extracted from the simulator itself, the total energy consumption with the fine-grained power gating scheme can be obtained. Paper [15] has introduced the accuracy of this simulator, verified by the real data from a 180 nm-based prototype ASIC.

As can be expected, this study mostly depends on the instruction types and their distribution. Thus, we calculate the incidence rate of the different instructions of the benchmark functions, which are shown in Fig. 8. We categorize the instructions by memory accesses and logic/arithmetic/media operations. As can be easily observed from Fig. 8, the ratio of instructions in a function varies according to the application characteristics. For example, FI-2 and FI-3 have only memory accesses and logic operations. Differently, Expand4k, Unsharp and Blur have a high ratio of media operations. According to our technique, these functions require more check instructions, since media operations have a higher level of ODP and may be more possibly turned into a defective unit.

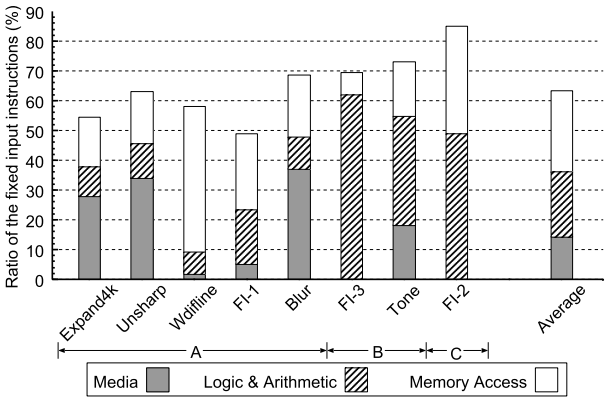


Fig. 9 Incidence of instructions with a fixed input.

Furthermore, we optimized the ODP of the instructions with a constant input to reduce the number of check instructions further as introduced in Sect. 2.3. Figure 9 shows the ratio of instructions with a constant input inside the functions. Note that most of the address calculations for the memory access operations have one constant input as the offset to the base address. As an example, `LD R1,@(R10,#4)` is a memory access operation, in which, `(R10,#4)` calculates the memory address by taking an immediate input #4. Therefore, there is a rough tendency that programs with a large portion of load operations is likely to have a higher ratio of operations with constant inputs. As an example, the difference between FI-2 and Expand4k in Fig. 8 and Fig. 9 gives a good demonstration of this tendency. Another rough observation is that media operations contribute far less constant inputs than other types. As a result, for Wdifline, the low ratio of constant inputs overwhelms the gaining of high ratio from the memory operations, which results in a medium level of fixed input ratio in this benchmark in Fig. 9. Overall, the average result shows that it is possible to update the ODP more precisely in 65% instructions with the insensitivity from constant values, which may have a visible increase in the accuracy in avoiding non-critical checking.

3.2 Reduction of Check Instructions

Figure 10 (a) gives the number of check instructions by using algorithm (Fig. 2) and ODPs in Table 1. Figure 10 (b) shows the results of more precisely updated ODP by taking constant inputs into account. For comparison, both figures also include the ratio of check instructions from the light-weighted technique, which only adds the check instruction at the end of the execution-path. It can be observed from Figs. 10 (a) and 10 (b) that the light-weighted method can

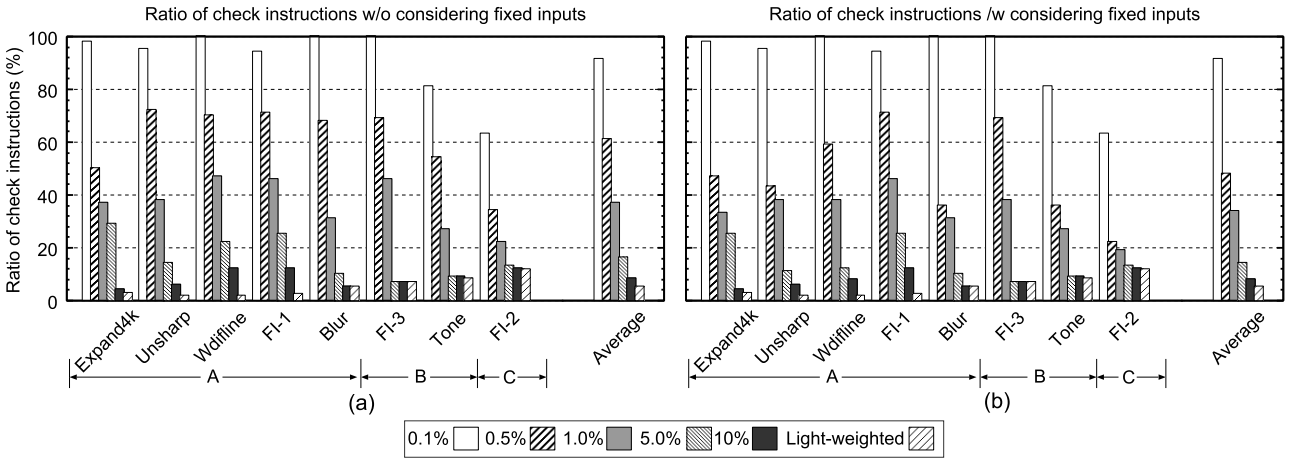


Fig. 10 Ratio of check instructions for different thresholds.

achieve the smallest number of check instructions. However, we can also find in these figures that the difference of the check instruction ratio is very small between the light-weighted method and our proposal when ODP_{th}=10%, as compared to other ODP_{th} values. A detailed analysis indicates that the difference between ODP_{th}=10% and the light-weighted method is usually within one check instruction, which helps divide the loop kernel into two branches and accordingly isolates the error propagation so as to increase the dependability. From this view, we can roughly represent the light-weighted method by taking a large ODP_{th} such as 10% or more. However, although these large ODP_{th}s help reduce the check instructions largely, their corresponding reliability will be traded off, which will be discussed in detail in Sect. 3.3.

It can be predicted that the number of check instructions will be dominant by the ODP threshold used in our method. However, there are some other parameters that change the final insertion of selective check instructions, such as the type of instructions, the length of the critical paths, and the number of branches inside the functions. For example, in Fig. 10, functions from Unsharp to FI-1 (Type A) contain a lot of media operations. A low ODP threshold 0.1% leads to about 95% selective check instructions, and they will divide the program into 1-instruction zones. However, this represents situations of impractically high error rates. With a threshold 0.5% or higher, the number of selective check instructions can be largely reduced. Specifically, for a threshold of 10%, the number of check instructions can be averagely reduced to a 7% level, in which the added check instructions segment the data path into 5 or 6 zones, each contains 17% instructions.

Secondly, programs with a short critical path (Type B), as Blur, FI-3 and Tone, behave differently from the programs in Type A. The increase ratio of the check instructions is high for low ODP thresholds. However, for the thresholds over 5%, no additional check instructions are required other than checking the final result. On average, these programs have 15% check instructions, and for large thresholds, only the check instruction for the final result covers the whole data path.

Finally, the Type C function, FI-2, which contains many independent data paths, also behaves in the same way as the Type B function. In FI-2, there is one store per each independent data branch, which has been designed to contain a built-in check. These in-store checks have already segmented the whole data-path into small zones, which do not eagerly require additional checks to lower the criticality of the ODP accumulation. For this reason, the number of check instructions stops growing at ODP_{th} = 5%. The total number of check instructions, including the in-store checks, remains at 8% even when ODP_{th} = 10%.

According to the Figs. 10 (a) and 10 (b), the number of check instructions differs by taking or not taking the influence from the constant input into account. The maximum difference can be found when ODP_{th}=0.5%, averagely. Furthermore, studying the individual programs, we can have the following detailed observations:

1. Under ODP_{th}=0.5%, benchmark programs Unsharp, Wdifline, Blur, Tone and FI-2, show large differences by using the insensitivity from the fixed input in calculating ODP. These benchmark programs contribute to most of the reductions from Figs. 10 (a) to 10 (b), which is 13.2% under ODP_{th}=0.5%. This may come from their relatively high ratios of constant input operations. However, in other than ODP_{th}=0.5%, minor changes can be found between Figs. 10 (a) and 10 (b) for these benchmarks.
2. Programs Expand4k and FI-1, show very minor changes under most ODP_{th}s.
3. The best ODP_{th} to distinguish Figs. 10 (a) and 10 (b) for benchmark *FI-3* is ODP_{th}=1.0%.

The above observations can be connected to the combination of the program characteristics such as the ratio of the operations with a fixed input, the distribution of operations with a fixed input among all the operation types, and the weight of all input operation types. For example, Expand4k

and FI-1 have a small number of operations with a fixed input, they therefore have less changes after taking the fixed input into account. FI-3 has a medium ratio of operations with a fixed input. However, it has a relatively short data-path, which is 14 operations in Table 2. In addition, the 14 operations in FI-3 are mainly the logic and arithmetic ones, which have a simpler distribution as the other benchmarks. Therefore, FI-3 shows some differences between Figs. 10 (a) and 10 (b) under ODP_{th}=1.0%, which is slightly different from other benchmarks.

3.3 Dependability and Energy Savings

By adding different numbers of check instructions, we are changing the dependability of the data-path by preventing the tainted data from the defective unit propagating inside the data path. The data propagation is usually safe when every instruction is duplicated and checked at the output point of the data-path. However, a prior research [5] on hard error in a data-center also states that the hard error rate will be very high for the servers that have previously experienced a hard fault. Therefore, it is possible that during the propagation of the tainted data, the dependability will go unpredictable when facing a second error before the tainted data is detected. For this reason, we use the metric of error propagation distance to measure the dependability of the data-path. The propagation distance is defined as the delay between the error generation node and the detection node inside the DFG. For a 10-node single branch data-graph, if we have only one check instruction at the end, the propagation distance will be 9 if an error occurs in the first node.

Accordingly, the reliability is calculated in term of vulnerability to a second error, as follow:

$$E = \sum_{i=1}^{n} P_i \times D_i \tag{5}$$

In Eq. (5), *n* is the number of instructions in a segment. This segment refers to a DFG segment between two check instructions added according to the ODP_{th} in this research. P_i is the ODP of an instruction and D is the error propagation distance. According to Eq. (5), the vulnerability to a second error is 0% when all the instructions are protected by a check instruction, as the distance D_i is 0. With the decreasing number of check instructions under an increasing ODP threshold, the vulnerability to a second error, as E in Eq. (5), will increase.

The reliability of our proposed technique with considering and without considering the fixed input is depicted in Fig. 11. Figure 11 clearly shows that the propagation distance decreases sharply from the light-weighted method to ODP_{th}=10% and then to ODP_{th}=5.0%, which indicates that these large ODP_{th}s may suffer more from the occurrence of a second error when the first error is still inside the data-path before detection. However, the decrease of error propagation distance goes rapidly flat when ODP_{th} corsses 1.0%, indicating a saturation in the check instructions. This also

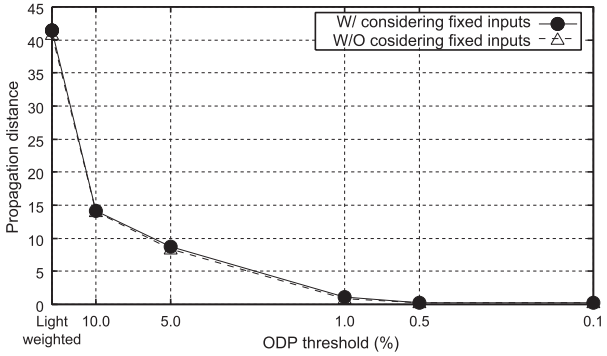


Fig. 11 Vulnerability to the second error for different check instructions.

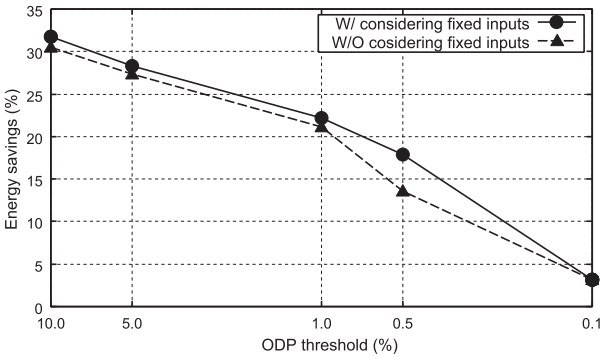


Fig. 12 Energy savings for 10M executions.

matches the expectation that the dependability increase will be exponentially difficult after it reaches a certain level so that balancing cost and efficiency is necessary. In addition, from another view, at ODP_{th}=1.0%, when a 1.0% longer error propagating distance is allowed, the check instructions can be further reduced to a 66% level.

Studying the influence of applying constant inputs in ODP, it can be easily observed that two lines are almost overlapping each other, which indicates that applying the insensitivity of fixed input to decrease ODP does not hurt the reliability. There is almost no difference between ODP_{th}=0.5% and ODP_{th}=0.1%, and for ODP_{th}=0.5%, 39% check instructions can be saved while the influence of the fixed input are not considered. There are 52% check instructions that can be removed from the DFG by considering the influence of fixed inputs for the same ODP_{th}. Another observation, at ODP_{th}=0.1%, is that when a 1.0% longer error propagating distance is allowed, the check instructions can be further reduced to the 66% level.

Figure 12 gives the energy saving results by the proposed method, as normalized by the energy of the original exhaustive checking method. Figure 12 also shows that we can save more energy by means of optimized ODP than the normal ODP especially for ODP_{th}=0.5%. Combining with the results in Fig. 11, we can achieve the same near-optimal reliability at ODP_{th} = 0.5% ODP_{th} = 0.1% by saving 17% more energy. If 1.0% downgrading of reliability is allowed, further 22% energy reduction is possible by using

ODP_{th} = 1.0%.

4. Conclusion

In this paper, we have presented a technique to remove check instructions from non-critical positions to avoid an exhaustive checking for fault-tolerable execution. The method can efficiently work with a reconfigurable FU array architecture to achieve high dependability with awareness of the fault possibility. In our approach, check instructions are added selectively according to the error probability (ODP) along the data path when the accumulated possibility exceeds a threshold. In addition, we also studied the influence of constant inputs as they can turn parts of the circuit into don't care zones and therefore help reduce the sensitivity to the faults.

Our study of the dependability of the updated data-path has shown that the reliability can still be kept at a near-optimal level when properly removing 52% non-critical check operations. This results in an energy saving of 17% for high dependable execution. With an allowance of downgrading 1.0% reliability, it is possible to reduce the energy further by 22%. In summary, a cost-effective high dependability method has been achieved by using our ODP metric in the FU array processor.

Acknowledgements

This work is supported by VLSI Design and Education Center (VDEC), University of Tokyo with the collaboration of Synopsys Corporation. This work is supported by JST ALCA, KAKENHI (No. 24240005, No. 24650020, and No. 2370060), and JST A-STEP No. AS242Z02732H.

References

[1] J.M. Rabaey, A. Chandrakasan, and B. Nikolic, Digital integrated Circuits- A Design Perspective, 2nd ed., Prentice Hall, 2004.
[2] J.H. Stathis, "Reliability limits for the gate insulator in CMOS technology," IBM J. Research and Development, vol.46, no.2.3, pp.265–286, March 2002.
[3] J. Srinivasan, S. Adve, P. Bose, and J. Rivers, "The impact of technology scaling on lifetime reliability," Proc. 2004 International Conference on Dependable Systems and Networks, DSN'04, pp.177–186, June-July 2004.
[4] J.W. McPherson, "Reliability challenges for 45nm and beyond," Proc. 43rd Annual Design Automation Conference, DAC '06, pp.176–181, 2006.
[5] A.A. Hwang, I.A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design," Proc. Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '12, pp.111–122, 2012.
[6] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin, "Ultra low-cost defect protection for microprocessor pipelines," SIGARCH Comput. Archit. News, vol.34, no.5, pp.73–82, Oct. 2006.
[7] S. Arslan and G. Shah, "A flexible in-field test controller," Proc. 2011 IEEE 14th International Multitopic Conference, INMIC '11, pp.71–75, Dec. 2011.
[8] N. Oh, P. Shirvani, and E. McCluskey, "Error detection by duplicated instructions in super-scalar processors," IEEE Trans. Reliab., vol.51, no.1, pp.63–75, March 2002.
[9] R. Hartenstein, "A decade of reconfigurable computing: A visionary retrospective," Proc. 2001 Conference on Design, Automation and Test in Europe, DATE '01, pp.642–649, 2001.
[10] S. Hauck and A. DeHon, Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation, Morgan Kaufmann, Amsterdam, 2007.
[11] K. Yoshimura, T. Iwakami, T. Nakada, J. Yao, H. Shimada, and Y. Nakashima, "An instruction mapping scheme for FU array accelerator," IEICE Trans. Inf. & Syst., vol.E94-D, no.2, pp.286–297, Feb. 2011.
[12] N. Devisetti, T. Iwakami, K. Yoshimura, T. Nakada, J. Yao, and Y. Nakashima, "LAPP: A low power array accelerator with binary compatibility," Proc. 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, IPDPSW '11, pp.854–862, 2011.
[13] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," Proc. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003, MICRO-36, pp.29–40, Dec. 2003.
[14] A. Suga and K. Matsunami, "Introducing the FR500 embedded microprocessor," IEEE Micro, vol.20, no.4, pp.21–27, July/Aug. 2000.
[15] M. Saito, S. Shitaoka, D.V. Naveen, S. Oue, K. Yoshimura, J. Yao, T. Nakada, and Y. Nakashima, "Development of high power-efficient processor with linear FU array accelerator," IEICE Trans. Inf. & Syst. (Japanese Edition), vol.J95-D, no.9, pp.1729–1737, Sept. 2012.

Tanvir Ahmed received the M.Sc. in Electrical Engineering from Linköping University in 2011. He is currently a doctoral student in the Graduate School of Information Science, Nara Institute of Science & Technology. His research interests include computer architecture and VLSI implementation of DSP algorithms. He is a student member of IEEE.

Jun Yao received the B.E., M.E. degrees from Tsinghua University in 2001 and 2004 respectively. He received the Ph.D. degree from Kyoto University in 2009. Since 2013, he has been an associate professor in Graduate School of Information Science, Nara Institute of Science and Technology. His current researches are in computer architecture field, especially low-power consumption and high-reliable architectures. He is a member of IEEE, ACM and IPSJ.

Yuko Hara-Azumi received her Ph.D. degree in computer science from Nagoya University in 2010. From 2010 to 2012, she was a JSPS postdoctoral research fellow at Ritsumeikan University and a visiting researcher at University of California, Irvine, USA and Karlsruhe Institute of Technology, Germany. Since 2012, she has been with the Graduate School of Information Science, Nara Institute of Science and Technology, where she is currently an assistant professor. Her research interests include system-level design automation for embedded/dependable systems. She currently serves as organizing and program committees of several premier conferences including ICCAD, ASP-DAC, and so on. She is a member of IEEE, and IPSJ.

Shigeru Yamashita is a professor of Department of Computer Science, College of Information Science and Engineering, Ritsumeikan University. He received his B.E., M.E. and Ph.D. degrees in information science from Kyoto University, Kyoto, Japan, in 1993, 1995 and 2001, respectively. His research interests include new types of computation and logic synthesis for them. He received the 2000 IEEE Circuits and Systems Society Transactions on Computer-Aided Design of Integrated Circuits and Systems Best Paper Award, SASIMI 2010 Best Paper Award, 2010 IPSJ Yamashita SIG Research Award, and 2010 Marubun Academic Achievement Award of the Marubun Research Promotion Foundation. He is a member of IEEE and IPSJ.

Yasuhiko Nakashima received the B.E., M.E. and Ph.D. degrees in Computer Engineering from Kyoto University, Japan in 1986, 1988 and 1998 respectively. He was a computer architect in the Computer and System Architecture Department, FUJITSU Limited in 1988-1999. From 1999 to 2005, he was an associate professor in the Graduate School of Economics, Kyoto University. Since 2006, he has been a professor in the Graduate School of Information Science, Nara Institute of Science and Technology. His research interests include processor architecture, emulation, CMOS circuit design, and evolutionary computation. He is a member of IEEE CS, ACM and IPSJ.