Vol.2012-ARC-201 No.4
2012/8/1

Achieving Near-Optimal Dependability with Minimal Hardware Costs in an FU Array Processor by Soft Error Rate Monitoring

Tanvir Ahmed[1,a)] Jun Yao[1,b)] Yasuhiko Nakashima[1,c)]

Abstract: From the program characteristics of applications like image processing, the loop index to control the accessing of all pixels is relatively more important than the pixel data themselves. This assumption can be usually used to effectively balance the fault toleration coverage and the hardware cost to achieve a dependable execution with tolerable error rates, especially when the fault rate is not high . However, it can also be expected that when the fault rate exceeds some boundary, this partial error coverage will go unreliable. Furthermore, it also loses the track of the current error information due to the incomplete coverage. For this purpose, we propose methods to help make a decision about the switching between partial and full redundancies by dynamically monitoring the soft error rate. From our experiment results, our method can achieve a steady dependability with a smallest hardware cost in an FU array based processor.

1. Introduction

Process technology has been scaled aggressively in past years to continuously lower supply voltage, to fasten the transistor switching speed, and shrink transistor sizes to increase the chip density, which have both contributed mainly to the device performance increases. However, the shrinking transistor size, decreasing node capacitance, increasing operating frequency, and reducing switching voltage add new pressure to the switching correctness of transistors, especially when working under possible transient fault attacks [1–4]. For all these reasons, it is thus essential to embed architecture-level fault-tolerance in microprocessors [5] to get the correct calculations with current or future transistors which lack of sufficient reliability.

Redundancy, either in temporal or spatial duplication form, is the usual architectural way to detect errors in devices and then guarantee dependability after a proper recovery [6–9]. When every operation is executed for multiple times, a full set of fault detection and recovery in these technologies are supposed to guarantee a 100% dependability inside the redundant sphere.

Meanwhile, due to the high cost of duplicated execution time or hardware resources, balancing the redundancy and cost is always an urgent issue for effective fault toleration. For example, program characteristics of applications like image processing have already demonstrated that the program flow controller, which is usually the loop index, is relatively more important than the pixels themselves. Like image processing, applications based on approximate calculation such as decision making from probability exploration, machine learning and etc, may similarly require a different dependability in control and data.

The above assumption is now gaining popularity in achieving an effective dependable system. Partial redundancies [10–12] have shown better performance to balance the energy and dependability for the applications like image processing, communication signal processing and approximate programming. Relax [10] proposed a handful of simple extensions to the programming language, compiler, ISA, and microarchitecture levels that simplify hardware design by enabling efficient software-level recovery of hardware faults. This framework constructed a spectrum of language models combining retry and discard behaviors with coarse and fine recovery granularities to enable flexible application handling of errors. Consequently, architecture support for the in-order and the out-of-order processors has been proposed for the disciplined programming in [11]. This study divide the program based on the importance of the operations. For example, the loop index calculation has a high priority than the other calculations so that the dependability was only introduced for the loop index calculation. Similar idea has been proposed for the functional unit array in EReLA [12], which is a loop accelerator. A minor hardware increase is used to double-check correctness of loop counter. It can already achieve a near optimal error coverage for image processing, especially in a low fault rate environment.

Partial redundancy has shown an excellent solution for normal cases where the fault rate remains low and the program behavior is clear. However, it certainly lacks the ability of tolerating worst-case situation. Under a possible burst of fault occurrence, it is still possible that the error rate in the data reaches an intolerable level, despite that the program flow is carefully maintained. As the program flow controlling portion is usually small in code

[1] Computing Architecture Lab, Graduate School of Information Science, Nara Institute of Science and Technology, Takayama Cho 8916-5, Ikoma, Nara 630-0192, Japan
[a)] tanvir-a@is.naist.jp
[b)] yaojun@is.naist.jp
[c)] nakashim@is.naist.jp

space, it is possible that the remaining data calculation part is much larger in size and thus is more likely to accumulate errors. Furthermore, the partial redundancy structure make it impossible to keep a track of the current execution information. All these problems downgrade the availability of service of this program.

To solve the above problems, we propose a method to achieve near optimal dependability by predicting the future error rate and calibrating the past execution. We use LAPP [13, 14] and its dependable execution EReLA [9, 12] as the baseline architecture for this study. EReLA uses a VLIW ISA to give an easy way to manipulate the dependability level by duplicating operations inside the VLIW instructions. Both partial redundancy and a higher level full redundancy can be easily achieved. In this paper, a small test program is executed to measure the error strikes during a certain period. Two counters are used to keep a track of a long-term error rate of past 1−billion cycles and a short term error rate of the current loop execution. Based on the error strike rate on the long term error rate, decisions will be taken of the dependability for the next loop execution. Differently, before ending the current loop, the short term error counter will be checked to decide whether it should be re-executed under a higher level of redundancy, in case of many errors strike at the time of partially redundant execution. This method thus adapts the execution redundancy according to the error rate. Results show that, our proposed technique uses 7% hardware overhead as a test program to achieve the optimal dependability. Under a relatively high fault rate, this method can still achieve near-optimal dependability while the partial redundancy one can only ensure the reliable execution of the control information.

The rest of this paper is organized as follows. Background studies are discussed in Section 2. Section 3 introduces our proposed scheme and policies to achieve optimal reliability in FU array. The results are given in Section 4 and finally, Section 5 concludes the paper.

2. Background Study

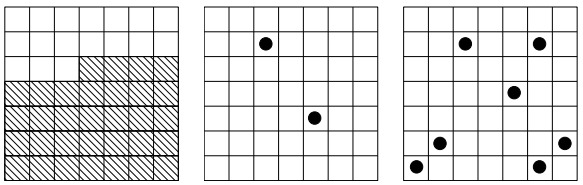
As discussed above, the partial redundancy [11, 12] has shown better performance for balancing the power consumptions and reliability in the executions of programs where approximate calculation is allowed. As an example, **Fig. 1(a)** shows an small loop and the corresponding VLIW code based on the FR-V instruction set architecture, where partial redundancy is specifically applied by duplicating the loop index calculation. The loop counter calculations are executed redundantly in the `I0`, `subicc gr39,#1,gr39,icc3` and `subicc gr7,#1,gr7,icc1`. Their results, `gr39` and `gr7`, are compared by the `dcei gr39,gr7` instruction in `I1`. The rest of the instructions are executed once, under certain knowledge that imprecise data calculation may be tolerable. As introduced in paper [12], any error on the loop index calculation is detected by this technique and will be recovered by restarting the loop.

This architecture can thus guarantee a full execution of all loop iterations. In **Fig. 1(b)**, 1(c), 1(d), the blocks represent the loop iterations. Each block has the instructions of that loop inside. As shown in Fig. 1(b), when the loop counter becomes erroneous or a branch takes place prior to the real ending in non-dependable

```
int a[N];
for (i=N-1; i>=0; i--)
  a[i]++;
```

```
loop:
I0: ld @(gr4,0),gr12;
    subicc gr39,#1,gr39,icc3;
    subicc gr7,#1,gr7,icc1;
I1: dcei gr39,gr7;
    beq icc1,0x0,end;
I2: addi gr12,#1,gr12;
I3: st gr12,@(gr4,gr0);
    addi gr4,#4,gr4;
    bra loop;
end:
```

(a) Partial Redundant VLIW code



(b) Loop execution without dependability (c) Effect of Low error rate (d) Effect of high error rate

Fig. 1 Partial refundant execution

system, all the loop iterations thereafter may become totally unprocessed or shifted. **Fig. 1(c)** shows an example of a partial redundant execution. By means of the keeping correct loop counter, all the blocks are at least processed. However, there are some errors on the instructions inside the loop and blacks dots inside the square represent these errors. This technique, partial redundant execution, is thereby showing better performance in a low error rate environment.

Conversely, when the error rate is high, where the errors are likely to strike frequently, this partial dependability is not enough to keep the reliable execution of any program. **Fig. 1(d)** shows an example of the partial dependable execution, where the error rate is higher than in Fig. 1(c). Therefore, there are many black dots in the figure, even though all the loop index calculations are executed correctly. The overall quality of the program execution is thus degraded significantly.

It can be expected that other program characteristics will also affect the data accuracy in a partial redundant environment. As we are studying the program in parallel loop form, two major parameters are the loop kernel size and the number of iterations. For EReLA, a loop with a large size contains more instructions and will use more functional units (FUs) inside the FU array. Under a certain error rate, the ratio of data errors and control errors will therefore increase. The partial redundancy will more likely to miss the detection of data errors and thus degrade the quality of data processing. Accordingly, keeping a track of the error rates inside the data calculation part is more essential inside the large loops.

The affects from number of loop iterations are more complicated. In one hand, a small amount of loop iterations will make the program short and have a small chance to be hit by the faults. On the other hands, for extremely large number of loop iterations, the possibility of fault hitting on loop counter will also increasing, causing a roll-back of partial redundant execution to keep dependability. However, frequent roll-back of large loops may come with an affordable performance loss. There is also a balancing point to make decisions whether switch or not to a higher

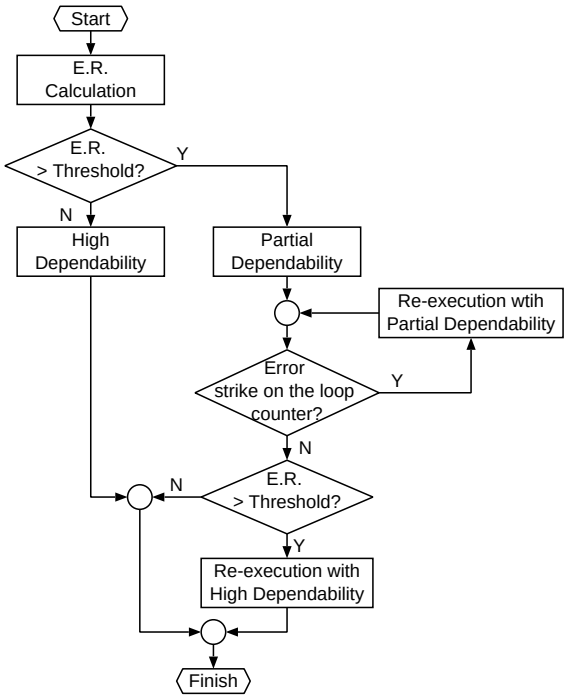


Fig. 2 Execution flow of the proposed technique

redundancy at the beginning of loops with many iterations, according to the estimation of the error rate.

3. Proposed technique

The above background study requires a good estimation of execution error rate which the partial redundancy is not able to provide. In this section, we introduce our hardware structure to estimate the error rate and policies that adapt redundancy according to the monitored error information.

3.1 Outline of our proposal

Fig. 2 gives an outlined flow of our method to adapt the redundancy level for the current and next loop execution, which is accelerated by the FU array inside the LAPP and EReLA architecture. Error sampling and counting schemes, known as test programs are added along the whole execution. As shown in Fig. 2, the method starts by detecting the loop. Based on a long-term measure of the number of errors detected error by the test program for the past 10^9 cycles, the number of loop iterations and instructions inside the loop, the expected error rate is calculated. If the calculated expected error rate exceeds the threshold the next loop execution will be executed with a higher level of redundancy to keep the reliable execution of the loop, in which a full duplication and comparison are performed to guarantee all execution results are finally checked before committing. Otherwise, the partial redundancy is predicted to be sufficient to provide enough reliability and is thus used to save the working energy for this next loop body.

In the second part, after a partial redundancy is scheduled, our method also monitors the number of errors on the test program during the real execution. This is known as a short and real-term of error rate during the current execution. If this short-term error rate exceeds the threshold, the loop execution will be re-executed

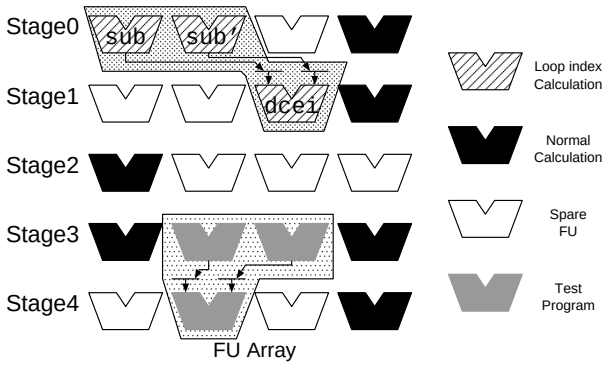


Fig. 3 Test program in FU array

with a higher level of dependability to ensure the reliable execution.

Note that for LAPP and EReLA architecture follows idempotent policy [15] which uses program itself to serve as a clean check-point. Specifically, EReLA holds the store data in a sufficiently large local cache during the loop execution [9]. Before the local cache is committed into the lower-level cache, the loop can be safely restart as a roll-back to a previous processor state.

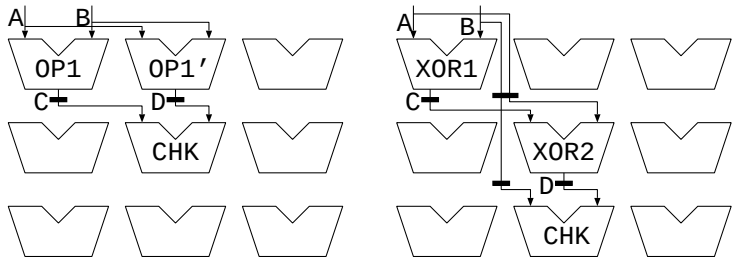
3.2 Monitoring error rate

Our proposed technique to monitor the error rate in FU array by test program is depicted in **Fig. 3**. In the figure, three functional units in the first two stages have been used for the reliable loop index calculation, in which a duplicated calculation and a check in the next stage are used to verify the correctness of the calculation. The black functional units of the FU array in the figure are for calculating the remaining instructions inside the loop, which are without duplication under a partial redundancy. The white ones are the spare functional units. Assuming that there is always several spare units, we can put our test programs inside to gather the information of errors. In this example, three units in gray serve for this purpose. Any error detected by the loop index calculation will restart the loop execution, whereas error detects by the test program will only use for calculating the error rate. A detail discussion about the architecture, instructions mapping and loop acceleration on LAPP and error correction scheme in EReLA and various level of redundancies can be found in literature [9, 12–14].

As discussed above, the EReLA architecture supports the error detection and recovery. In order to report any error on the pipline stages, the error propagates by a chain of flipflops and OR gates to the final stage of the array. In this study we are adding two counters at the end of the chain to calculate the number of errors. One counter is used to calculate the long term errors for future use and the other one starts from initial states at each loop execution, which counts the short term errors.

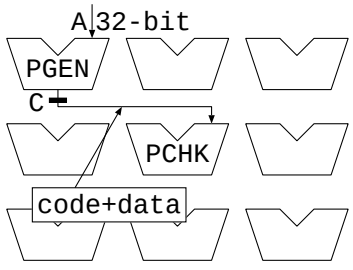
3.3 Test Programs

In this study, different test programs have been used for monitoring the test programs, such as redundant execution of the random operation, RAID-like execution with random input value, parity checking and ECC protection. The test programs are depicted in **Fig. 4**, 5. **Fig. 4(a)** shows the example of redundant ex-



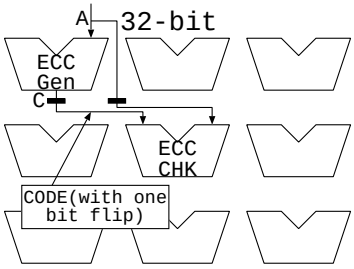
(a) Redundant execution (b) RAID-like execution

Fig. 4 Simple test programs




```
C = A&0X7FFFFFFF|(PGEN(A)<<31); Error =
(C&0X80000000)>>31 == PCHK(C&0X7FFFFFFF)
```

(a) Parity function as a test program




```
C = ECCGEN(A)^0X00000004;
Error = ECCCHK(A,C) == (0X00000000|0X00000002)
```

(b) Error correcting code as a test program

Fig. 5 Parity and ECC for test program

ecution of the random operation, where one random instruction is executed on two different functional units of the same stage of the array, and the results are checked by the functional unit of the next stage. This test program requires three functional units in two different stages. Besides, the random operation is selected based on the number of gates used by the instruction. For example, a multiplication is usually used due to its large number of gates which is accordingly easy to catch error when transient fault attacks. Similarly, **Fig. 4(b)** gives a test program based on RAID-like execution. This test program also requires three functional units, and they are mapped on three different stages. In the figure, at the first stage one functional unit is performed the `XOR` operation between `A` and `B`. The output `C` is `XOR` with `A` in the next stage and the result is written in `D`. Finally, `B` and `D` are compared in the third stage. Both the above test programs can detects multiple errors.

On the other hand, **Fig. 5(a)** shows the test program by parity bit. Two additional instructions, parity generation (`PGEN`) and parity check (`PCHK`), have been introduced for this test program. `PGEN` instruction takes a 32−bit input value and generate parity bit. In order to avoid propagation of the additional bit on the FU array, the `PGEN` instruction takes a 32−bit input and generate a parity bit based on the lower 31−bit, from 0 to 30. The most significant bit (MSB) is discarded and the parity bit takes its place in the result. The output is checked by the `PCHK` instruction in the later stage. In the 32−bit output, the 31^{st} bit is considered as parity and the rest 31 bits are considered as data. This test program can detect one bit error. Finally, **Fig. 5(b)** shows an example of a test program by error correcting code. Two addition instructions are newly introduced for this test program and they are mapped on two different stages of the array. In the first stage ECC generation (`ECCGEN`) instruction takes a 32−bit input and generates the ECC code and flips one bit intentionally. The ECC code and the input data are checked by the `ECCCHK` instruction at the next stage of the array and correct the flipped bit. However, this instruction will report error for two bits error as well as for no error. One error is introduced in `ECCGEN` instruction by flipping one bit. The `ECCCHK` instruction will recover the error and reports a error for another bit error, which is not the intentional error. There is a possibility of error strike on the flipped bit at the `ECCGEN`. Therefore, the `ECCCHK` instruction will also report error, while there is no error. This test program can detect up to 2 bits error.

3.4 Error rate calculation and policies

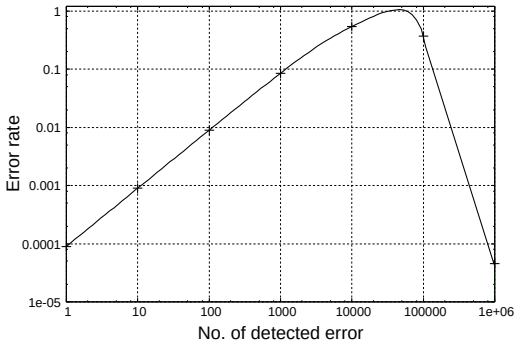
The expected error rate, $E.R.$, of a loop execution is calculated as:

$$E.R. = [1 - (2 \times T \times e)]^N [1 - (1 - (I - 2) \times T \times e)^N] \quad (1)$$

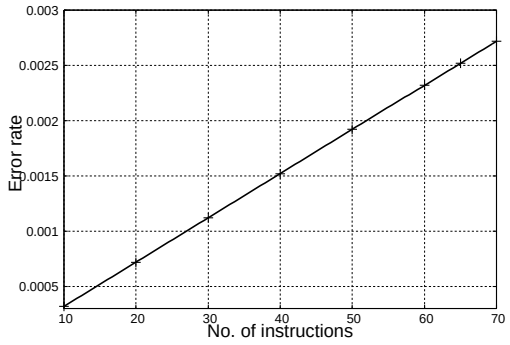
where, $[1 - (2 \times T \times e)]$ calculates the correct execution for the loop index and $[1 - (1 - (I - 2) \times T \times e)^N]$ calculates the expected error rate at the rest of the instructions, in which T is the number of the transistors used in a functional unit, and I is the number of instructions inside a loop. The number 2 in Eq:1 refer to the two FUs used for loop index processing. e is the fault rate, which can be calculated by the detection test program, as

$$e = \frac{C}{10^9 \times T \times n} \quad (2)$$

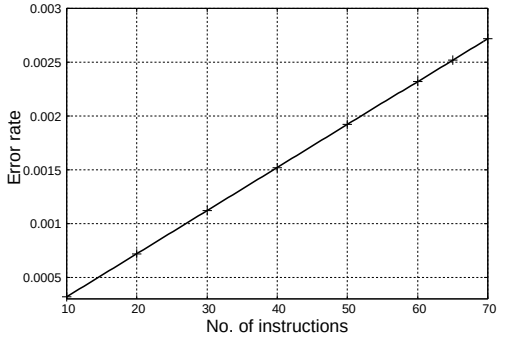
Here, C is the error count in the test program for past 10^9 cycles and n is the number of functional units used by the test program. Based on Eq:1, the expected error rate is calculated for different error count (C) by the test program, different number



(a) Error count as a function of Error rate



(b) Number of iterations as a function of Error rate



(c) Number of instructions as a function of Error rate

Fig. 6 Expected error rate calculation

of loop iteration (N) and different program length(I) in **Fig. 6**. In **Fig. 6(a)**, the expected error rate is increasing with the number of error strikes on the test program for a certain time. However, it is assumed that for many strike on the test program, there will be strikes on the loop index counter, which then recovers the error by restarting the loop and the expected error rate will reduce significantly. This high error rate is notexpected as well as unpractical. For this demonstration, the number of loop iteration is 10^3 and the number of instructions inside the loop body are 20 and the number of instructions in the test program are 2.

As discussed before, the expected error rate will vary for different number of loop iteration and number of instructions inside the program. **Fig. 6(b)** and **Fig. 6(c)** show the example of the iteration and program length as a function of expected error rate, while the number of strikes on the test program is constant. The expected error rate is increasing with the number of iterations and program length.

4. Results

We tried our proposed technique on different benchmark loops to evaluate the hardware overhead for different test programs. Redundant operation and RAID-like test programs require three functional units, whereas the parity and the ECC use two FUs. Thus, we divide the test program in two categories. **Fig. 4** shows the result of hardware overhead for different benchmark loops, where the number of the instructions inside the loops are varies from 72 to 15. In the figure, loops from Expand4k to Blur have instructions over 40 and the overhead for these loops are 5.3% for the redundant and RAID-like test programs and 4.1% for the other test programs. On the other hand, loops, from F3 to Edge, have instructions less than 30, where the overhead is 16.7% for the test programs with three functional units and 11.3% for the the test programs with two functional units. Finally, on an average the overhead is 7% for the test programs on different benchmark loops.

The results also have shown that the size of the loop body and number of iterations mostly contribute the expected error rate rather than the test program. Thus, according to the results, the test programs behave equally, despite of the differences in the number of FUs. The Redundant operation and the RAID-like test programs require more functional units than the parity and the ECC and the overhead is also low for the parity and ECC for the large as well as for the small loops. However, both the parity and the ECC require 2 additional instrions to the ISA, which is the additional cost. Therefore, depending on the hardware overhead and the implementation cost test program can be selected for.

Fig. 8 shows the dependability of the simple partial redundancy mode without or with our test programs to give additional monitoring of data correctness. The Partial redundancy is reliable for the low error rate, as shown in **Fig. 8(a)**. The dependability is near 100% for the low error rate, whereas the partial redundancy becomes unreliable with the increment of the error rate. For example, the dependability degrades from 90% to 50% for a medium error rate and for very high error rate it becomes near 5%. On the other hand, the high redundancy uses almost double hardware to achieve the 100% dependability. EReLA [12] reports, the high redundancy consumes 100% to 150% more energy on an average than the partial redundancy. The above problems can be overcome by improving the dependability of the partial redundancy. By means of our proposed technique the dependability of the partial redundancy has been improved. **Fig. 8(b)** shows the improvement of dependability by monitoring the data correctness with test programs and giving another chance to the processor to use high-level redundancy. Specifically, The dependability of the partial redundancy for very high error rate has been reached to 80% from 5% and for the medium error rate has reached to 90% from 50% with the additional test program.

5. Conclusion

In this paper, we have presented a technique to achieve near optimal dependability by monitoring the error rate from test programs. Specifically, the level of dependability of a loop execution

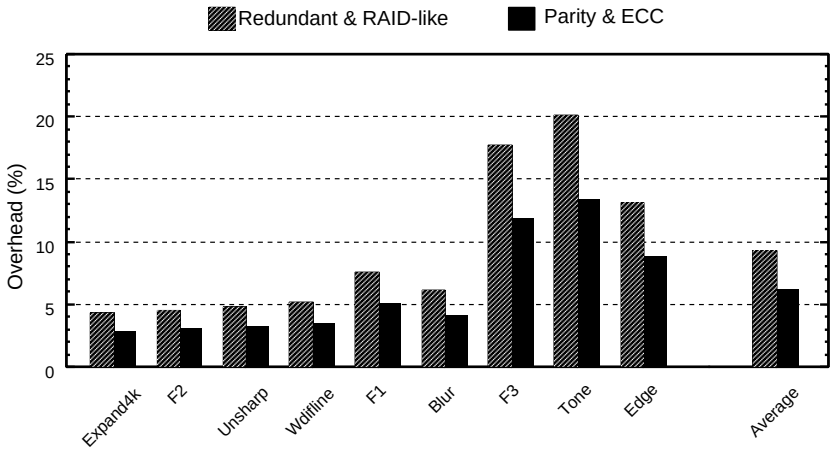
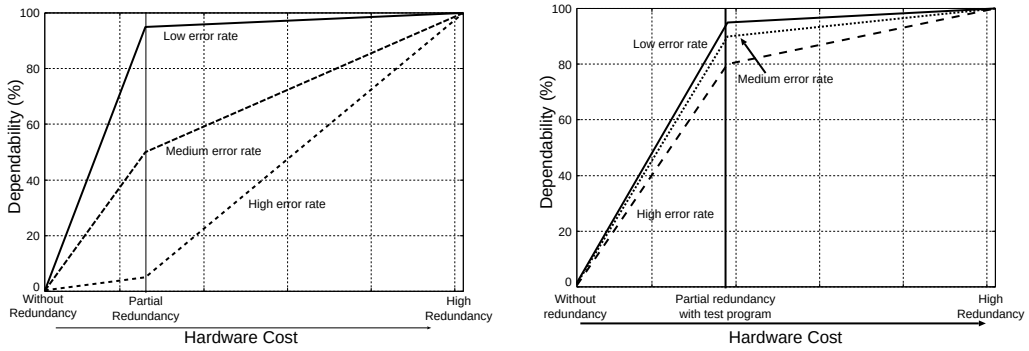


Fig. 7 Hardware overhead for different benchmark loop



(a) Dependability without the proposed technique (b) Dependability with the proposed technique

Fig. 8 Improvement of the dependability in partial redundancy by the proposed technique

is changed according to the error rate of the long term and short term executions. Different test programs have been used to monitor the error rate and the expected error rate is calculated based on the detected error for past loop execution, number of iterations and instructions of the next loop. As a result, this techniques changes the dependability level of a processor when there is the necessity of the improvement. Finally, our results have shown that the overhead of the test program on an average is 7% for different benchmark loops and the dependability for a very high error rate improves from 5% to 80% with the test program giving addtional measurement of data correctness.

Acknowledgments 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. AS232Z02313A.

References

[1] Wang, F. and Agrawal, V.: Single Event Upset: An Embedded Tutorial, *21st International Conference on VLSI Design, 2008. VLSID 2008*, pp. 429 –434 (2008).

[2] Wang, H., Rodriguez, S., Dirik, C., Gole, A., Chan, V. and Jacob, B.: TERPS: the embedded reliable processing system, *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005*, Vol. 2, pp. D/1 – D/2 Vol. 2 (2005).

[3] Pickel, J. C. and Blandford, J. T.: Cosmic Ray Induced in MOS Memory Cells, *IEEE Transactions on Nuclear Science*, Vol. 25, No. 6, pp. 1166 –1171 (1978).

[4] Karnik, T. and Hazucha, P.: Characterization of soft errors caused by single event upsets in CMOS processes, *IEEE Transactions on Dependable and Secure Computing*, Vol. 1, No. 2, pp. 128 – 143 (2004).

[5] Srinivasan, J., Adve, S., Bose, P. and Rivers, J.: Lifetime reliability: toward an architectural solution, *IEEE Micro*, Vol. 25, No. 3, pp. 70 – 80 (2005).

[6] Oh, N., Shirvani, P. and McCluskey, E.: Error detection by duplicated instructions in super-scalar processors, *IEEE Transactions on Reliability*, Vol. 51, No. 1, pp. 63 –75 (2002).

[7] Chen, Y.-Y. and Leu, K.-L.: Reliable data path design of VLIW processor cores with comprehensive error-coverage assessment, *Microprocessors and Microsystems*, Vol. 34, No. 1, pp. 49 – 61 (2010).

[8] Gaisler, J.: A portable and fault-tolerant microprocessor based on the SPARC v8 architecture, *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pp. 409 – 415 (2002).

[9] Yao, J. and Nakashima, Y.: Exploiting Efficiency of Redundant Executions on an FU Array, *IPSJ SIG Notes*, Vol. 2011, No. 9, pp. 1–5 (2011-03-03).

[10] de Kruijf, M., Nomura, S. and Sankaralingam, K.: Relax: an architectural framework for software recovery of hardware faults, *SIGARCH Comput. Archit. News*, Vol. 38, No. 3, pp. 497–508 (2010).

[11] Esmaeilzadeh, H., Sampson, A., Ceze, L. and Burger, D.: Architecture support for disciplined approximate programming, *SIGARCH Comput. Archit. News*, Vol. 40, No. 1, pp. 301–312 (2012).

[12] Oue, S., Yoshimura, K., Yao, J., Nakada, T. and Nakashima, Y.: High Redundancy Instruction Mapping Scheme on FU Array Accelerator, *IPSJ SIG Notes*, Vol. 2011, No. 19, pp. 1–7 (2011).

[13] Yoshimura, K., Iwakami, T., Nakada, T., Yao, J., Shimada, H. and Nakashima, Y.: An Instruction Mapping Scheme for FU Array Accelerator, *IEICE Transaction on Information and Systems*, Vol. E94D, No. 2 (2011).

[14] Devisetti, N., Iwakami, T., Yoshimura, K., Nakada, T., Yao, J. and Nakashima, Y.: LAPP: A Low Power Array Accelerator with Binary Compatibility, *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, IEEE Computer Society, pp. 854–862 (2011).

[15] de Kruijf, M. and Sankaralingam, K.: Idempotent processor architecture, *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pp. 140–151 (2011).