A Stage-Level Recovery Scheme in Scalable Pipeline Modules for High Dependability

Jun Yao, Hajime Shimada
Graduate School of Information Science,
Nara Institute of Science and Technology,
Ikoma 630-0192, Japan
Email: {yaojun,shimada}@is.naist.jp

Kazutoshi Kobayashi
Graduate School of Science and Technology,
Kyoto Institute of Technology
Kyoto 606–8585, Japan
Email: kobayasi@kit.ac.jp

***Abstract*—In the recent years, the increasing error rate has become one of the major impediments for the application of new process technologies in electronic devices like microprocessors. This thereby necessitates the research of fault toleration mechanisms from all device, micro-architecture and system levels to keep correct computation in future microprocessors, along the advances of process technologies.**

Space redundancy, as dual or triple modular redundancy (DMR or TMR), is widely used to tolerate errors with a negligible performance loss. In this paper, at the micro-architecture level, we propose a very fine-grained recovery scheme based on a DMR processor architecture to cover every transient error inside of the memory interface boundary. Our recovery method makes full use of the existing duplicated hardware in the DMR processor, which can avoid large hardware extension by not using checkpoint buffers in many fault-tolerable processors. The hardware-based recovery is achieved by dynamically triggering an instruction re-execution procedure in the next cycle after error detection, which indicates a near-zero performance impact to achieve an error-free execution.

A TMR architecture is usually preferred as it provides a seamless error correction by a majority voting logic and therefore generates no recovery delay. With our fast recovery scheme at a low hardware cost, our result shows that even under a relatively high transient error rate, it is possible to only use a DMR architecture to detect/recover errors at a negligible performance cost. Our reliable processor is thus constructed to use a DMR execution with the fast recovery as its major working mode. It saves around 1/3 energy consumption from a traditional TMR architecture, while the transient error coverage is still maintained.

***Index Terms*—Fault tolerance, redundancy, system recovery**

I. Introduction

Nowadays, failures in the electronic devices have presented a serious challenge for the correct operations of the modern processors. The electronic failures are usually caused by soft and hard ones. A soft error is marked as transient and may occur in a processor when a high-energy cosmic particle charges/discharges and inverts the transistor logical state. A hard error is caused by permanent physical defects and the circuit may not recover to its normal status as under a transient one. The pressure from faults will be even threatening with the technology trends in the device processing area leading to the reduction in operating voltage, the increase in processor frequency and the increase in the density of on-chip transistors, as indicated in papers [1]–[4]. Specifically, paper [4] gives a study of the relationship between the soft error rate (SER) and the supply voltage in many processor units, including latches, a chain of 11 fan-out-of-four (FO4) inverters (as a representation of combinational logic unit), and SRAMs. Increasing tendencies of SER in these processor units can be observed along the decrease of the supply voltage, which is a typical trend as the process technology improves. Meanwhile, the rate of permanent faults caused by electronicmigration, stress-migration, time-dependent dielectric breakdown, or thermal cycling is likely to take a similar increasing trend in consequence of technology scaling, as introduced in paper [5]. For these reasons, it can be predicted that future process technology will be unfeasible to produce processors with sufficient reliability. Schemes specially for reliable executions from either the device or the architecture levels are thereby required to keep processors advancing along with the continuous scaling of process technology.

Many fault tolerable mechanisms at the architectural level, such as dual executions in IBM's S/390 G5 and z990 microprocessors [6], [7], simultaneous and redundantly thread (SRT) [8], and Chip-level Redundantly Threaded multiprocessor with Recovery (CRTR) [9] have been employed to alleviate the increasing pressures from electronic errors. The error detection in these architectures is mainly performed by checking results from duplicated executions, and the recovery is achieved by rolling back to a previously stored checkpoint state. A coarse checkpoint granularity is commonly applied in these architectures to prevent a very frequent checkpoint update. However, a major drawback of the coarse granularity is that all processor running statuses including contents of register file, system control registers, and memory updates are necessary to be periodically buffered. The hardware extension to achieve the storage of checkpoint data can hardly be neglected. In addition, a relatively complex recovery sequence based on a software interrupt is generally required, as described in paper [6]. It may be a visible impact to performance under a future technology where error occurrence may become more frequent than the current period.

In this research, we proposed a very fine-grained recovery scheme for a space redundancy processor, in which the recovery granularity is at a stage level. This baseline processor architecture is from a previous research, where a pipeline structure was designed for the purpose of modularizing high reliable system via space redundancy [10]. In the constructed reliable processor, data sanity checks are performed per each pipeline stage, which provides thorough information of the correctly executed stages inside the processor. These detailed execution information are effectively used to achieve the proposed recovery which dynamically schedules proper instruction re-executions after an error detection, following a similar route after the resolution of a branch misprediction. In summary, this paper has presented the following contributions:

1) It gives a recovery method by including very few additional hardware units like checkpoint buffer. The already redundant hardware units in a DMR processor are fully used as the checkpoint information for recovery.
2) The delay from recovery is minimized by using an extremely short distance rollback. It is achieved by re-executing the instruction after the latest correctly executed stage inside the pipeline. With a proper control, the re-execution starts from the cycle right after the error is detected.
3) It reduces 1/3 working energy as compared to a traditional TMR processor while maintaining an equal coverage for transient errors. With a negligible performance loss from error recovery, a DMR processor with this fine-grained recovery can be a substitution for a TMR processor in tolerating all transient faults.

The paper is organized as follows: Section II introduces the design of a scalable pipeline module for constructing processors with adaptive redundancy. Section III describes our hardware based fine-grained recovery proposal, which effectively uses the detailed per stage error detection information. In Section IV, performance and hardware results are presented to study the effectiveness of the proposal. Section V concludes the whole paper.

II. Scalable Pipeline Module for Constructing Dependable Processors

In our research, we are focusing on a framework with relatively ideal reliability, whose fault tolerating abilities can meet the top requirement from systems working under a very high electronic error occurrence environment, such as computational units in a satellite. Space redundancy [7], [8], [11], [12] is assumed because of its better coverage for both transient and permanent faults than the time redundancy [13]–[15].

Generally, a Triple Modular Redundancy (TMR) [10] architecture is preferred for its seamless coverage for all transient and permanent faults. However, a traditional fixed connection between the three identical modules in TMR presents little flexibility. Its seamless recovery capability will be unsustainable after a permanent fault and will thus require a new set of TMR to continue the recovery function, despite that 2/3 of the original logics may still work properly. In addition, the triple redundancy is usually an over-design under the assumption that transient faults are still more common cases than permanent ones. A possible solution to these problems

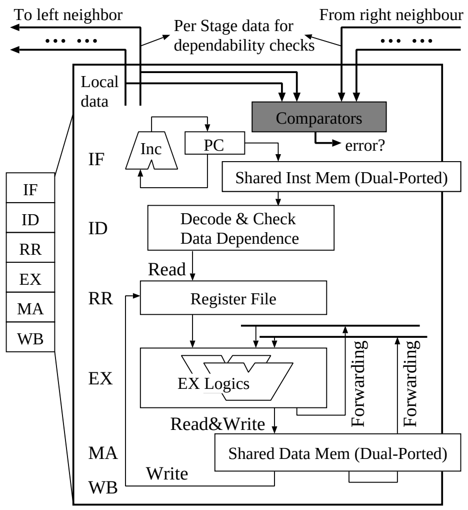

Fig. 1. A scalable pipeline module with dependable check

can be a flexible connection and an adaptive space redundancy based on proper reconfiguration. For these reasons, we use a scalable pipeline module from a previous research to construct processors with an adaptive space redundancy. The fine-grained error detection in the baseline architecture also serves as a background to achieve a fast recovery in this paper.

A. Scalable Pipeline Module Design with Dependability

The scalable pipeline design to construct dependable processors is given in Fig. 1, which contains six traditional textbook-style stages as instruction fetch (IF), instruction decode (ID), register read (RR), execution (EX), memory access (MA), and writeback (WB).

Different from the implementation in papers [7], [8], [11] which are also fault-tolerable designs based on space redundancy, we designed the error detection to be performed at a stage boundary. The distributed comparators inside every stage can help achieve an early and thorough error detection. With a fast recovery scheme, it is possible to introduce a minimal performance impact even under a relatively high error rate.

As shown in Fig. 1, input/output unidirectional links are included in the pipeline module, which are used for dependability data bypassing. A space redundancy processor can be constructed by connecting multiple pipeline modules where copies of a single thread are simultaneously executed and compared.

The storage units including register files, memory units (instruction and data caches) are designed to be covered by Error Correcting Codes (ECC) [16] logics to guarantee a reliable data storage. Data stored into the memory structures are regarded to be safe if they were checked before committing. Different to the register file, caches are shared at the processor level, since duplicating these large units will cause a huge area cost.

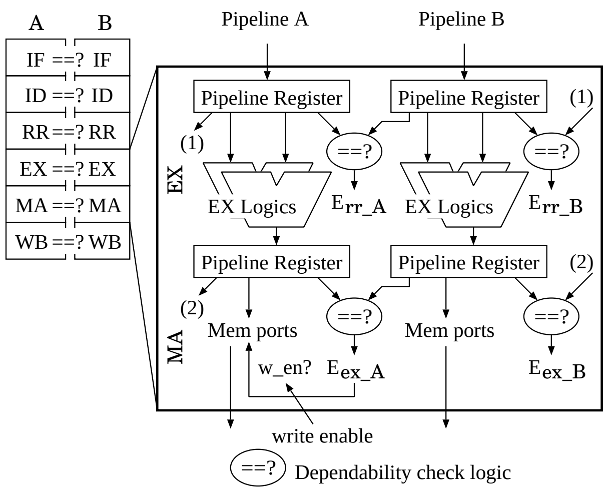

Fig. 2. DMR execution scheme in EX and MA stages

B. Dual Modular Redundancy (DMR) Based Fault-Tolerance

Based on the scalable pipeline module introduced in Section II-A, processor cores with high dependability can be constructed via proper scaling. Figure 2 shows the DMR combination by including two pipeline modules.

Using the framework given in Fig. 2, reliable computation is guaranteed by a redundant execution of each instruction. The instruction replication starts at the fetch stage. The starting program counter (PC) of a single application thread is respectively provided to both pipelines when the thread is launched. The duplicated PCs are then accumulated in each pipeline module. With the dual-ported instruction memory shared between the two pipeline modules as shown in Fig. 1, same instructions can be read out in either fetch stage from its own memory I/O port. After that, the two identical instructions will be provided to the latter stages in sequence, achieving a mirrored execution of the original program. Since the two pipelines work on a same instruction stream, one memory write port will be sufficient for this DMR mode. Data inside memory and register files are covered by ECC logics to tolerate single event upset (SEU).

Although this DMR architecture is very similar to some traditional lock-step based fault tolerating microprocessors such as IBM's S/390 G5 [6], the lock-step mechanism in this research is designed not to impede the processor working frequency. As Fig. 2 illustrates, the dependability check logic is employed after each pipeline register, which contains the output of the last stage generated in the previous cycle. Specifically, for the EX stage, the pipeline register before it holds the information of operation code, source/destination register numbers, and the source operands data. These information—from last RR stage—serve as the inputs for the combinational logic in the EX stage in this clock period[1]. Meanwhile, as shown in Fig. 2, all the pipeline register outputs will be passed to the left neighbour pipeline by a unidirectional link. Accordingly, at the same time that EX stage in either pipeline works on the instruction provided by the pipeline registers before it, the outputs of these two pipeline registers are compared by the data dependability check logics. The results of these data validity checks are marked as E_{rr_A} and E_{rr_B} (with a notation of "rr") because they actually indicate the correctness of outputs from the RR stage of the previous cycle. By this way, the critical path of EX stage is not impacted by the dependability check logics and the clock frequency can remain uninfluenced after adding reliable features.

[1]Forwarding paths also provide inputs for EX stage. However, they are from latter pipeline registers in MA and WA stages and will be check there accordingly.

An erroneous state indicated by either signal of E_{rr_A}, E_{rr_B} reveals that some faults have occurred in either the previous RR stage or the pipeline register between RR and EX stages. This error may most probably influence the execution correctness of the EX stage, and it should thereby not be propagated forward. For stage like MA, as shown in Fig. 2, when signals E_{ex_A} and E_{ex_B} contain error information, data committing to the data memory shall also be disabled. Only for the data committing part of MA and WB stages, the critical paths are extended by the lock-step mechanism. As these two stages usually are not the bottleneck in determining the working frequency, we can regard the frequency to be unchanged after including the dependability check logics.

III. Stage-Level Error Recovery Scheme

According to the background introduction, two identical instructions are simultaneously executed in the DMR based processor architecture. Since those duplicated units in the space redundant system—including the register files and sequential elements such as pipeline registers—already provide a secondary storage for the processor running information, they may be sufficient to serve as the checkpoint data in the traditional sense so as to reduce the hardware extensions for the recovery supports. Moreover, in this architecture, comparisons are made per each stage boundary, which provides very fine-grained information of the accurate executions in the processor. Therefore, the recovery can start from the stage that error is detected, instead of rolling back to a historical checkpoint state. In this section, a stage-level instruction re-execution scheme is proposed to achieve a quick hardware based recovery, in which a minimal rollback is performed based on the thorough understanding of erroneous locations inside the DMR processor.

A. Basic Idea of the Recovery

Many coarse-grained checkpoint based recovery method does not require frequent data sanity checks. The performance and frequency impacts from error detection are therefore less likely to be visible. However, after detecting the error, the coarse-grained recovery methods will usually require a software sequence like an interrupt to help roll back to the checkpoint state. The software sequence is also required to be implemented with a full understanding of the processor specification. In this section, we are trying to propose a hardware based recovery scheme for a fine-grained rollback,

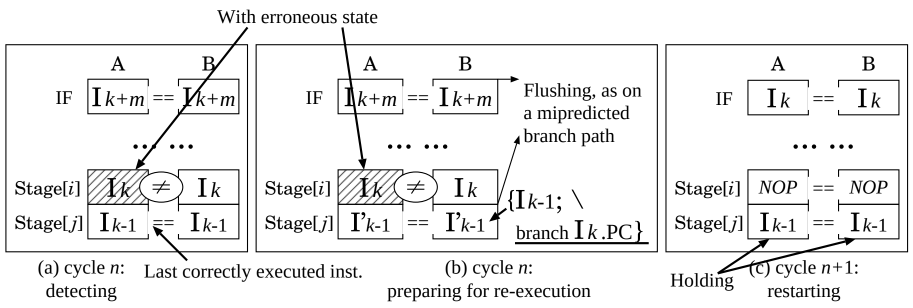
With erroneous state
A
B
IF
Stage[i]
Stage[j]
I_{k+m} == I_{k+m}
I_k ≠ I_k
I_{k-1} == I_{k-1}
(a) cycle n: detecting
Last correctly executed inst.
I'_{k-1} == I'_{k-1}
Flushing, as on a mipredicted branch path
{I_{k-1}; \ branch I_k.PC}
(b) cycle n: preparing for re-execution
I_k == I_k
NOP == NOP
Holding
(c) cycle n+1: restarting

Fig. 3. Restarting execution by inserting a branch style instruction.

where software interference may not be necessary if the error is caused by a temporary fault.

We designed our recovery scheme by using the idea of instruction re-execution, assuming that the temporary state change in combinational logics and sequential elements like pipeline registers will fade out after cycles. Basically, the control of re-execution is designed to be inside the DMR processor, which now has the runtime information of the stage holding the first erroneously executed instruction, and the stage with the last correctly handled instruction as well. The instruction in the last correct stage can thus be used as the checkpoint data.

The re-execution starts from an unconditionally jump to the correct restart point by citing the information stored in the last correct stage. Figure 3 shows the basic idea of this re-execution scheme. Using the example demonstrated in Fig. 3(a), I$_{k-1}$ and I$_k$ are two consecutive instructions and are executed in sequence. Assume that in this example, no empty cycles caused by hazards from data dependences or branch target resolution delay are between I$_{k-1}$ and I$_k$. As in Fig. 3(a), at cycle n, the comparators detect that one copy of the executions of I$_k$ is problematic. Assuming that the executions of I$_{k-1}$ are checked to have no error, I$_{k-1}$ can thus serve as the last correctly executed instruction at this point.

According to the design in Fig. 2, the error detected in cycle n may have occurred in the last stage's execution of I$_k$ in cycle n-1, or have been introduced by the state change in pipeline register which holds I$_k$ in cycle n. Since the execution is based on a dual replication, it is impossible to tell which copy of I$_k$ execution is correct. Alternatively, the information stored in the stages that contain I$_{k-1}$ will be used to indicate the correct restart point.

As shown in Fig. 3(b), after detecting the error in I$_k$, the last correct instruction as I$_{k-1}$ will be extended to compound with a dummy jump instruction, as $branch$ I$_k.PC$. I$_k$'s program counter (PC) will be filled into the jump instruction as the branch target. If the jump is correctly handled, I$_k$ will be re-fetched from the instruction cache and the re-execution will thereby start in cycle n+1. Note that the augmented branch instruction is used to express the idea for a proper rollback. It is not a real instruction and will only be valid inside Stage[j] in cycle n. Considering the possible delay issue by manipulating I$_k$, as the target of this dummy branch can be directly calculated after I$_k$ and its PC enter Stage[j], we can roughly regard that this augmentation does not extend the critical path.

When restarting the execution of I$_k$, pipelines will be partially flushed as shown in Fig. 3(b). The pipeline stages from IF to the stage that contains erroneous I$_k$ will be emptied by filling no-operation (*NOP*) instructions. The propagations of the last correct instruction I$_{k-1}$ in this example and instructions in the latter stages will be stalled by stopping the clock signals to those corresponding pipeline registers. This is for the fault toleration when further fault attacks during the recovery, by guaranteeing that there is always one or more correctly executed instructions serving as the checkpoint data in the pipeline. Although it is possible to use a hardened storage to cache committed instructions as a checkpoint like in IBM z990 microprocessors, the maximum distributions of checkpoint data in our design may alleviate the high dependence to the single point in an IBM processor, especially when facing the increasing threats from hard faults. Moreover, the recovery can be started directly from the erroneously executed instruction so that the depth of rollback may be smaller than many other system recovery schemes which use coarse-grained historical checkpoint.

As introduced in Section II-A, the memory structures including data memory, instruction memory, and register file are covered by ECC-like technologies. A correct data storage in them can thus be assumed. Accordingly, no further large checkpoint buffer will be required to cache changing logs.

B. Implementation of Transient Error Recovery

Figure 4 gives the implementation of the stage-level recovery by slightly extending the architecture in Fig. 2. In this figure, the units in gray are augmented for the recovery purpose. For simplicity, only part of the whole DMR processor is given.

Since the program counter (PC) of each instruction is the key information that we use to indicate the restart point, we compound them in the pipeline register in each stage, as depicted in Fig. 4. This cost of hardware extension may be negligible because usually PC is attached as a part of the pipeline register—at least till the execution stage—for calculating branch target and caching the correct return point of function or interruption calling. Normally, the pipeline register

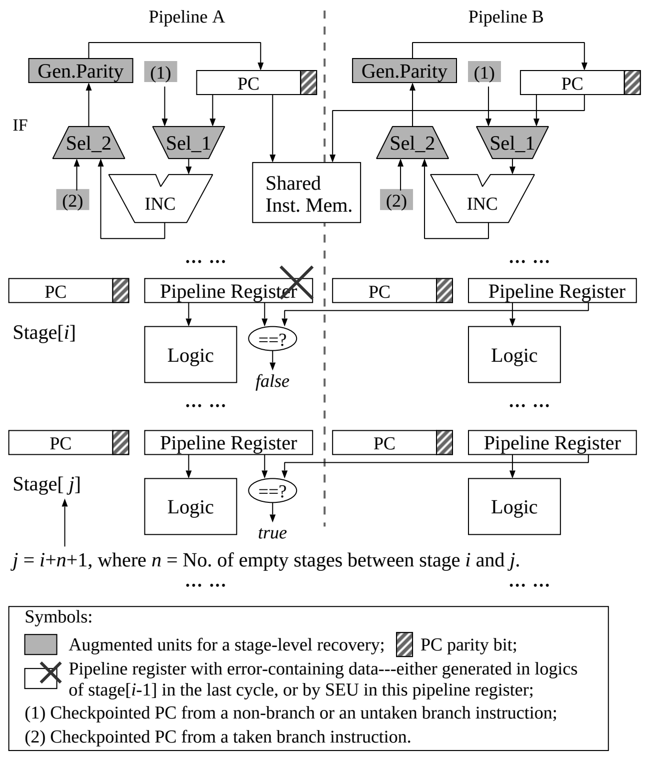
Pipeline A
Pipeline B
Gen.Parity
(1)
PC
IF
Sel_2
Sel_1
(2)
INC
Shared Inst. Mem.
... ...
PC
Pipeline Register
Stage[i]
Logic
==?
false
Stage[j]
true
j = i+n+1, where n = No. of empty stages between stage i and j.
Symbols:
Augmented units for a stage-level recovery;
PC parity bit;
Pipeline register with error-containing data---either generated in logics of stage[i-1] in the last cycle, or by SEU in this pipeline register;
(1) Checkpointed PC from a non-branch or an untaken branch instruction;
(2) Checkpointed PC from a taken branch instruction.

Fig. 4. Units required for recovery from a transient error.

(1) Procedure to restarting execution:

```
enum {IF, ID, RR, EX, MA, WB} i, j, k;

/* i) Locate the last stage with error */
for (i=WB;i≥IF;i--)
  if (pipeA_stage[i].error || pipeB_stage[i].error)
    break;
if (i<IF) return; /* No Error */

/* ii) Locate the next stage, skipping stages
      that contain hazard_NOP inst. */
for (j=i+1;j≤WB+1;j++)
  if (pipeA_stage[j].OP != hazard_NOP) break;
/* I_good in stage[j], I_error in stage[i] */

/* iii) Jump to the correct restart point
 * by using the correct value in stage[j] */
if (pipeA_stage[j].OP != BRANCH
    || ~pipeA_stage[j].taken) {
  /* Cond.a: I_error is the successive inst. of I_good */
  /* PC is only covered by parity */
  good_PC = getCorrectPC(pipeA_stage[j].PC,
                           pipeB_stage[j].PC);
  pipeA_stage[IF].CurrentPC = good_PC;
  pipeB_stage[IF].CurrentPC = good_PC;
  /* CurrentPC is (1) in Fig.4 */
} else {
  /* Cond.b: I_error is the branch target of I_good */
  pipeA_stage[IF].NextPC = pipeA_stage[j].brTarget;
  pipeB_stage[IF].NextPC = pipeA_stage[j].brTarget;
  /* NextPC is (2) in Fig.4 */
}

/* iv) Flush and enable stage[IF] to stage[j-1] */
for (k=IF;k<j;k++) {
  pipeA_stage[k].OP = pipeB_stage[k].OP = hazard_NOP;
  pipeA_stage[k].clk_EN = pipeB_stage[k].clk_EN = true;
}

check_point = j; /* Checkpointing */
/* end of restart execution procedure */
```

(2) Hold other stages till re-execution safely reaches stage[check_point-1]

```
while (pipeA_stage[check_point-1].op == hazard_NOP
    || pipeA_stage[check_point-1].error
    || pipeB_stage[check_point-1].error) {
  for (k=check_point;k≤WB;k++)
    pipeA_stage[k].clk_EN = pipeB_stage[k].clk_EN = false;
} /* Hold inst. from stage[check_point] */
```

Fig. 5. The algorithm of recovery procedures.

is the object for checking data correctness, as described in Section II-B. However, from the viewpoint of recovering, the role of PC to indicate the correct restart point will only be used under an error detection, which may be relatively rare. For the cost consideration, we do not include the paired PCs in the dependability verifying sphere. PCs will still be checked when they are a source operand of an instruction such as PC based load and many short range branches.

Since the processor is running under a DMR execution mode, it is thus impossible to identify the correct one from the dually replicated PCs if they are not identical. To address this problem, PC is additionally designed to be covered with a single parity bit which is generated in the instruction fetch (IF) stage, shown as "*Gen.Parity*" logic in Fig. 4. The parity bit will be attached to each PC and will only be used under a situation that an error occurs and the paired PCs in the last correct instruction are not the same. As multiple faults happening to the paired PCs in their short life cycle may be already rare, combining with a transcendental condition that these errors are visible only when another error has been detected in pipeline registers—which further decreases the possibility, we can assume that the duplicated PC and their parity will provide the correct result as a checkpoint.

As introduced in Section III-A, the last correctly executed instruction in the paired pipelines will be used as the checkpoint one to indicate the correct restart point when an error is detected. Based on the additional units for recovery in Fig. 4, Fig. 5 gives the detailed procedure to extract the proper restart point from the checkpoint instruction. We use the prefix "$pipeA$" and "$pipeB$" to indicate the two duplicated pipelines, and the "$stage$" array to represent the six stages in each pipeline, which are IF, ID, RR, EX, MA and WB. The "$error$" field is the error signal, generated by the comparator that verifies the correctness of the paired pipeline registers. The "OP" field in each stage is the operation code of the currently processed instruction. The expression of "$hazard\_NOP$" is the no-operation instruction added due to pipeline hazards. "$BRANCH$" serves as all branch-like instructions whose next instruction may not be the successive one in the incremental order. The "clk_EN" field in each stage is the enabling signal of the clock to this pipeline register. Disabling or enabling it can help stall or propagate the corresponding instructions.

Part (1) in Fig. 5 gives the sequence of preparations for restarting execution. In this part, Block i) is used to locate the earliest stage with an erroneous execution, which indicates the location of first execution with the error. Block ii) tries to find out the last correctly executed instruction. Note that these two blocks are written in a loop style for clarity. In the real implementation, these checks are done in parallel by using multiplexors to parse the error flags in all stages. After these two steps, variable i and j hold indices to the stages with a same meaning in Fig. 3.

Block iii) presents the core of this algorithm to extract the restart point from the information stored in $stage[j]$. Assume instruction I_{good} is the instruction being processed in $stage[j]$, and I_{error} is its next instruction in $stage[i]$, detected to be erroneous. According to the type of I_{good}, there are two different situations of its following instruction I_{error}, as:

1) Condition a (**Cond. a**): I_{good} is not a branch-like instruction or it is not a taken branch instruction, so that I_{error} is its successive one in the instruction memory. I_{error}'s PC can be calculated by incrementing I_{good}'s PC. Because PC in each stage is not covered by data comparison logic, a function "$getCorrectPC()$" will be triggered under this situation. Its major execution is to compare the PCs of I_{good}. If the two paired PCs are identical, they can be regarded as correct. Otherwise, the parity bit will be employed to indicate the correct one. After this procedure, the correct PC will be sent to IF stage as the correct restart point ("(1)" in Fig. 4), where it will be incremented like in normal IF stage processing.
2) Condition b (**Cond. b**): I_{good} is a taken branch. In this case, I_{error} is the branch target instruction whose PC will be the target calculation result of I_{good}. Since target calculation will be performed in ALU of EX stage, the ALU result is also output to the normal pipeline register. As pipeline registers of I_{good}'s executions have been verified by data comparisons, the target PC in instruction I_{good} can thus be directly forwarded to IF stage ("(2)" in Fig. 4) for the retrieve of I_{error} again in the next cycle.

By separately handling these two conditions, we can extract I_{error}'s PC from the correct information in I_{good}. However, there may be problems that I_{error} is already in the WB stage or the empty stages with $hazard\_NOP$ between I_{error} and I_{good} may cross the WB boundary. Both of these two circumstances will make I_{good} unavailable for the recovering procedure. To solve this, similar to the design of IBM z990 processor [7], we need to add a hardened storage after WB stage to serve as the last checkpoint data. Both PC and the branch target of the last committed instruction—excluding $hazard\_NOP$—will be committed into this storage which will be implemented as special control registers.

Besides the preparation of correct restart point from error detected in $stage[i]$, the erroneous data after I_{error} should be discarded from the processor. It is performed as a pipeline flush from IF to $stage[i]$, which is the purpose of block iv) in Fig. 5. According to the above design, the rollback scheme to start re-execution is very similar to the idea of recovery from a mispredicted branch. By using this mispredicted branch like idea, this rollback scheme can be extended to an out-of-order execution environment. It can be achieved by making the augmented dummy branch instruction "$branch\ \mathrm{I}_{error}.PC$" as a mispredicted branch. All instructions after I_{good} in the program order will be flushed, which is a normal processing in an out-of-order processor.

Part (2) in Fig. 5 is to stall the propagation of correct instructions, to keep them as multiple checkpoints for the consideration of tolerating further fault attacks during the recovering procedure. When the re-execution of I_{error} runs to $stage[j-1]$, these checkpoint instructions can start propagation again.

If the previous error is caused by a single event upset (SEU) or a single event transient (SET), it may probably vanish by proper re-executions. If no error signal is indicated after re-executing these instructions, the transient fault can be regarded as fixed by the error detecting and recovering in this DMR processor.

According to this design, the re-execution can be started from the next cycle of the error detection point. The recovering delay is at a comparable level to a normal branch misprediction penalty. Since the error rate is far smaller than the branch misprediction rate, the additional execution cycles caused by this small rollback recovery scheme can be regarded as negligible.

C. Triple Modular Redundancy (TMR) Mode

The fine-grained error recovery scheme in Section III-B can help the DMR processor overcome all transient errors by invoking proper re-executions. However, although a rare case, there may still be threats that permanent errors may occur after a long time utilization according to paper [5]. DMR mode can not find out the permanently damaged part and will always remain at the status of restarting the erroneous instruction. To handle this problem, we will include a third pipeline module into the DMR processor core, which is originally prepared but disabled by power gating inside the processor core[2]. This reconfiguration will not be activated under the normal situation until it detects a very frequent fault occurrence. At that time, the TMR processor will be employed for the diagnosis of system health. The reconfiguration will require a state machine to turn on the third pipeline module, prepare the data of both general purpose and special registers, and the control registers to define the correct connection.

Assume that pipelines *A*, *B* and *C* will be reconfigured to form a new TMR core, as illustrated in Fig. 6. Similar to the DMR design in Section II-B, an individual copy of execution is performed in each pipeline module. Also, in each pipeline, per stage data dependability check logics work on its local data and data from its right neighbour. The processor level voter can determine the permanently defected pipeline. After that, the processor can fall back to DMR mode by removing the defected pipeline.

Therefore, TMR is only required for diagnosing the defected units. DMR plus proper re-execution is the normal working mode to achieve a cost-effective reliable implementation.

[2]It is possible to use this third pipeline for performance boosting. For simplicity, we assume a spare third pipeline in this paper.

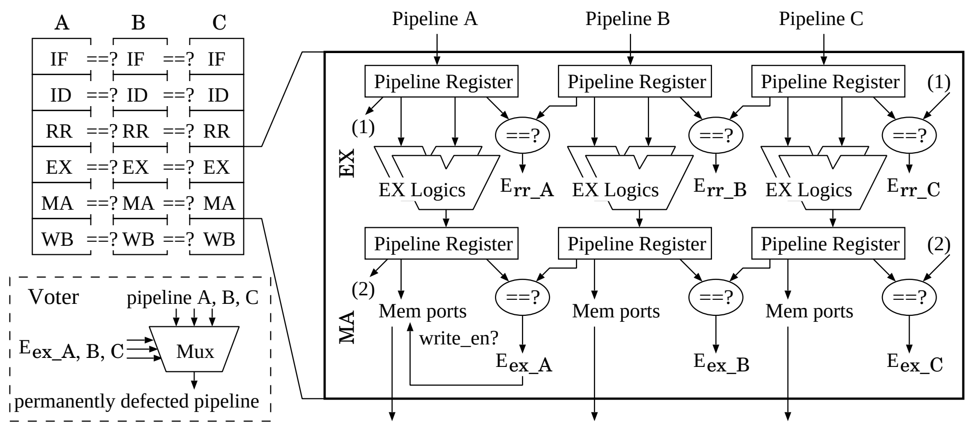

Fig. 6. TMR execution scheme in EX and MA stages

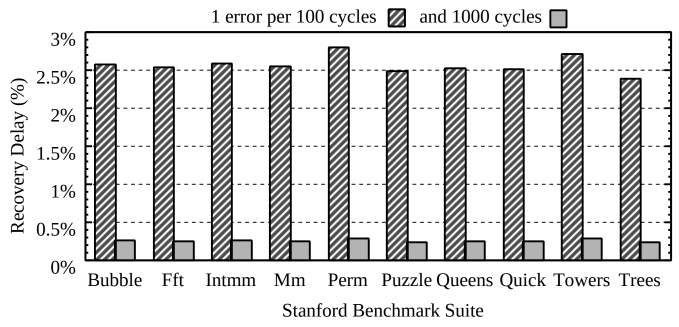

Fig. 7. Performance impact from the recovery under very high error rates.

IV. Performance and Synthesis Results

We designed the proposed scalable single pipeline module under Verilog HDL [17]. The processor follows SH-2 instruction set architecture (ISA) [18]. We assumed that the processor has 512-word instruction memory and 2KB data memory.

With the designed recovery scheme, it is possible to use DMR processor to cover every transient error. The total execution time also includes the delay that is necessary for the recovery from the erroneous execution. We conduced several experiments based on the register transfer level (RTL) simulation, using the above HDL implementation. Figure 7 gives the performance impact under a simple fault injection in which the faults are limited on the input of the EX stage.

Stanford benchmarks are used as the workload for the performance test. In these experiments, we assumed very high fault injection rates to enlarge the possible performance impact. Two sets of error rates, as 1 error per 10^2 cycles and 1 error per 10^3 cycles, are used. As shown in Fig. 7, even under an impractically high error rate like 1/(10^2 cycles), the performance impact is around 2.5%. It roughly equals to the product of the cycle level recovery delay which equals to a branch misprediction resolution delay and the error rate. The main difference between each execution is that sometimes an error hits insensitive instructions and recovery is not required. When error rate shrinks to 1/(10^3 cycles), the performance loss will be around 0.25%, which can be regarded as a negligible cost.

TABLE I
Area estimation of different pipeline units.

	Pipeline units	Active area (in NAND2)
(1)	Combinational logics + pipeline registers	35236
(2.a)	Non-ECC register file	9336
(2.b)	ECC register file	14345
(3.a)	Single-ported non-ECC memory (inst. + data)	23212
(3.b)	Dual-ported ECC memory (inst. + data)	36588
(4)	Units for Dependability (Error detection/recovery)	5489

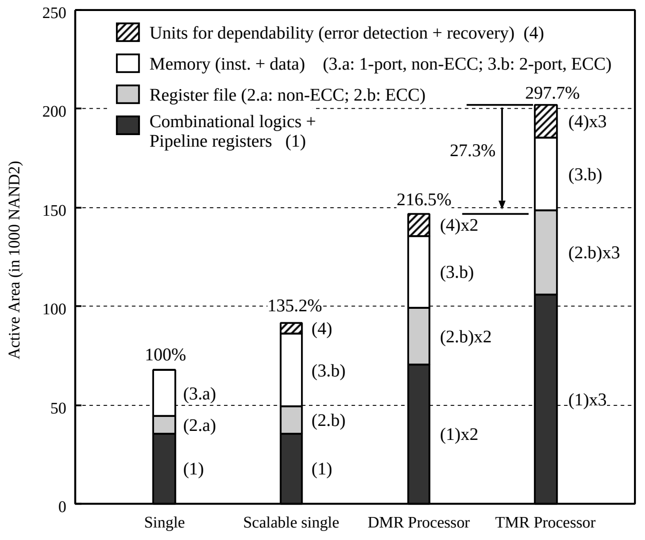

Fig. 8. Area estimation of several processor organizations

The area costs of the proposed high performance and dependable architectures were synthesized with Synopsys Design Compiler under Rohm 0.18μm cell library. Table I denotes the estimated active areas in several different kinds of pipeline units. Based on these units, we give a detailed hardware cost study of the pipeline architectures and reliable processors that are used in this research. Figure 8 shows the area of a simple single pipeline, the designed scalable pipeline module for reliable processors, and the two reliable processors following DMR and TMR architecture respectively. The very simple single pipeline processor given in the first bar is designed to be without any scalability and dependability, whose memory structures (I$ and D$) are single ported. It is only listed for comparison. For scalability and dependability, the memory is required to be dual ported and covered under ECC logics, as in the latter three bars. The meaning of each stacked bar in Fig. 8 is listed beside it, which corresponds to the units in Tab. I.

All the processor areas are normalized to the simple processor in the first bar. As in Fig. 8, the augmentation of scalability and ECC-ed storages requires an area increase of 35.2%. While being extended to space redundancy processors for high dependability, the area cost will be doubled or tripled. As in the third bar, a DMR processor requires a 116.5% more area to achieve a dual execution for error detection. A TMR processor uses 297.7% area of the unreliable and non-scalable processor to implement the processor health diagnosis under a permanent defect attack.

The recovery scheme introduced in this paper can be used to help the DMR processor to tolerate temporary errors. Therefore, a third unit replication to form a TMR processor will only be required to diagnose the permanently defected units. The DMR processor reduces the area by 27.3% compared to the TMR one. If we simply assume that power consumption is roughly proportional to the area, the same level energy consumption reduction can be achieved from a traditional TMR processor, under the finding that the proposed recovery scheme only introduces a very minor performance impact.

V. Conclusions

In this paper, a fine-grained recovery scheme is given to provide proper instruction re-execution under an error detection in a constructed DMR processor. The recovering procedure introduces a very small hardware extension for checkpoint data by making full use of the information of a stage-level error detection. The recovery performs a minimal rollback so that it can be finished in a same cycle level delay as a normal branch misprediction. Even under an impractically high error rate, the performance impact from recovery itself can be easily neglected.

With this fine-grained recovery scheme, a DMR execution can be primarily used as the reliable architecture for its ability to handle the occurrences of all transient errors caused by temporary faults. A TMR execution is only required when diagnosing permanently defected units. The major DMR working mode shows a 27.3% area reduction as compared to the traditional TMR processor. Power consumption in this part can be saved to achieve an energy-efficient reliable processor.

In addition, by using the very fast and low cost error recovery in this research, it is possible to allow more advanced process technology in the reliable processor while tolerating some minor performance loss from recovering procedures. The balance between processor area, working frequency, and recovery-included execution time under a future process technology will be the next optimization task of this research.

Acknowledgment

This work is supported by VLSI Design and Education Center (VDEC), University of Tokyo with the collaboration with Synopsys Corporation. This work is supported by JST CREST, JST A-STEP (FS) 10102567, and Grant-in-Aid for Young Scientists (B) No. 23700060.

References

[1] Xiaodong Li, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. SoftArch: An Architecture Level Tool for Modeling and Analyzing Soft Errors. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 496–505, 2005.

[2] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 389–398, 2002.

[3] Subhasish Mitra, Norbert Seifert, Ming Zhang, Quan Shi, and Kee Sup Kim. Robust System Design with Built-In Soft-Error Resilience. *Computer*, 38(2):43–52, 2005.

[4] Kazutoshi Kobayashi, Yusuke Moritani, and Hidetoshi Onodera. Soft-error Resiliency Evaluation of Delayed Multiple-modular Flip-Flops. In *Proceedings of DA Symposium 2008*, pages 181–186, 2008.

[5] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. The Impact of Technology Scaling on Lifetime Reliability. In *DSN'04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pages 177–186, 2004.

[6] T.J. Slegel, III Averill, R.M., M.A. Check, B.C. Giamei, B.W. Krumm, C.A. Krygowski, W.H. Li, J.S. Liptay, J.D. MacDougall, T.J. McPherson, J.A. Navarro, E.M. Schwarz, K. Shum, and C.F. Webb. Ibm's s/390 g5 microprocessor design. *Micro, IEEE*, 19(2):12–23, Mar/Apr 1999.

[7] P.J. Meaney, S.B. Swaney, P.N. Sanda, and L. Spainhower. Ibm z990 soft error detection and recovery. *Device and Materials Reliability, IEEE Transactions on*, 5(3):419–427, September 2005.

[8] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 25–36, 2000.

[9] Mohamed Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-Fault Recovery for Chip Multiprocessors. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 98–109, 2003.

[10] Daniel P. Siewiorek and Robert S. Swarz. *Reliable computer systems (3rd ed.): design and evaluation*. A. K. Peters, Ltd., 1998.

[11] Mohamed A. Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-Fault Recovery for Chip Multiprocessors. *IEEE Micro*, 23(6):76–83, 2003.

[12] Jared C. Smolens, Jangwoo Kim, James C. Hoe, and Babak Falsafi. Efficient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitectures. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, pages 257–268, 2004.

[13] Moinuddin K. Qureshi, Onur Mutlu, and Yale N. Patt. Microarchitecture-Based Introspection: A Technique for Transient-Fault Tolerance in Microprocessors. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 434–443, 2005.

[14] Eric Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, pages 84–91, 1999.

[15] N. Oh, P.P. Shirvani, and E.J. McCluskey. Error Detection by Duplicated Instructions in Super-Scalar Processors. *IEEE Transactions on Reliability*, 51(1):63–75, Mar 2002.

[16] C. L. Chen and M. Y. Hsiao. Error-Correcting Codes for Semi-Conductor Memory Applications: A State of The Art Review. pages 771–786, 1992.

[17] Jun Yao. openDARA: an RTL Open Source for a Fault Tolerable Processor Architecture. http://arch.naist.jp/pub/openDARA/, September 2010.

[18] Renesas Technology. *SH-1/SH-2/SH-DSP Software Manual Rev. 5.00*, 2004.