Improving Instruction Issue Bandwidth for Concurrent Error-Detecting Processors

Sato, Toshinori
System LSI Research Center, Kyushu University

Chiyonobu, Akihiro
Department of Artificial Intelligence, Kyushu Institute of Technology

Joe, Kazuki
Department of Information and Computer Sciences, Nara Women's University

出版情報：9th International Workshop on Innovative Architecture for Future Generation High Performance Processors and Systems, pp.21-28, 2006-01. International Workshop on Innovative Architecture for Future Generation High Performance Processors and Systems
バージョン：
権利関係：

Improving Instruction Issue Bandwidth for Concurrent Error-Detecting Processors

Toshinori Sato[1] Akihiro Chiyonobu[2] Kazuki Joe[3]
[1] *System LSI Research Center, Kyushu University*
[2] *Department of Artificial Intelligence, Kyushu Institute of Technology*
[3] *Department of Information and Computer Sciences, Nara Women's University*
[1] toshinori.sato@computer.org

Abstract

Soft error tolerance is a hot research topic for modern microprocessors. We have been investigating soft error tolerance microarchitecture, RED, which exploits time redundancy to achieve soft error tolerance without requiring prohibitive additional hardware resources. Unfortunately, our previous study unveiled that a RED-based processor suffers severe performance penalty. We guess that it comes from the reduction in effective instruction issue queue (ISQ) capacity. Since RED uses a register update unit (RUU), which combines an ISQ and a reorder buffer (ROB) into a single structure, redundant instructions occupy the ISQ. Actually, contemporary microprocessors use a dedicated ISQ, which is decoupled from the ROB rather than the RUU. In this paper, in order to reduce the performance penalty, we adopt RED into ROB-based microprocessors. We reduce the penalty from 17.4% to 12.4% and from 23.9% to 18.3% for integer and floating-point programs, respectively.
Keywords: microprocessors, soft errors, transient errors, dependable processors, fault tolerance

1. Introduction

The investigation of fault tolerance techniques for microprocessors is driven by two issues, (1) deep submicron fabrication technologies and (2) the increasing popularity of mobile platforms. Current semiconductor technologies have become more susceptible to high-energy neutrons from deep space. Following the trends in smaller transistors, lower supply voltage, and higher clock frequency, future microprocessors will become susceptible to soft errors, which constitute the vast majority of hardware failures. On the other hand, cellular telephones are currently used for applications which are critical to our financial security, such as mobile banking and mobile trading. Such applications demand that computer systems always work correctly. Based on these trends, it is expected that the quality with respect to reliability will become important as well as performance and cost for modern microprocessors.

In light of this, many soft error tolerance microarchitectures have been recently proposed [2, 5, 7, 9-14, 22, 25-27]. In this work, we are investigating the use of an instruction reissue mechanism, which exploits time redundancy to achieve soft error tolerance [15]. Originally, this concept was proposed for incorrect data speculation recovery [16]. We have modified and applied this mechanism for error tolerance. In order to detect transient faults, every committed instruction is duplicated and two results of the single instruction are compared. Since future microprocessors will utilize data speculation and include the instruction reissue mechanism [6], this mechanism costs the least hardware overhead. In addition, it is simple because detection of errors and their recovery processes are done simultaneously by means of dynamic instruction scheduling. We call this fault tolerance mechanism *RED* (Reissue-based Error Detection) microarchitecture.

Unfortunately, we found severe performance penalty of 21.9% on average [15]. While we are studying how to mitigate the loss [17-19], there is still a considerable performance gap. This paper focuses on how the reduction in effective capacity of the ISQ affects performance. Since RED uses the RUU, redundant instructions occupy the ISQ. Actually, contemporary microprocessors use a dedicated ISQ, which is decoupled from the ROB rather than the RUU. In this paper, in order to reduce the performance penalty, we adopt RED into ROB-style micro-processors.

The rest of this paper is organized as follows: Section 2 surveys related work. In Section 3, we introduce the RED microarchitecture and adopt it to

ROB-style microprocessors. Section 4 describes our evaluation methodology. Section 5 discusses our simulation results. Finally, Section 6 presents our conclusions.

2. Related Work

Recently, researchers have proposed soft error tolerance mechanisms that utilize several kinds of redundancies: time redundancy, space redundancy, and instruction redundancy.

2.1. Time redundancy

Time redundancy involves the process of storing information to handle transients that is shifted in time to check for unwanted changes. O3RS [9] exploits instruction reissue technique. Every instruction is duplicated and its two redundant results are compared to detect an error. REESE [25] also executes every instruction twice by injecting the completed instruction into its instruction scheduling window. It does not rely on the reissue mechanism but on a dedicated FIFO queue. SHREC [22] relies on an in-order checker processor to redundantly execute every instruction. To achieve fault recovery, O3RS uses the existing rollback mechanism, which is provided to support precise interrupts and to handle misspredicted branches.

Another error detection technique exploiting time redundancy is utilized in AR-SMT [14] and SRT [10, 13] processors. They are based on simultaneous multithreading (SMT) processors. A program thread is duplicated and two redundant copies of the single thread are executed simultaneously on an SMT processor. This technique only detects errors but cannot recover from every error transparently. The recovery should be supported by OS. SRTR [27] is an extension of SRT and has the ability of fault recovery, which is also based on the recovery mechanism for misspredicted branches.

2.2. Space redundancy

Space redundancy provides separate physical copies of a resource. DIVA [2] and SSD [7] utilize space redundancy. A checker processor attached behind its main processor pipeline is used for dynamically verifying committed instructions. When an error is detected, DIVA raises an exception, which flushes the processor pipeline and then restarts the processor. This mechanism is basically the rollback mechanism.

The Slipstream [26] processor is a variant of AR-SMT on a chip-multiprocessor (CMP). Two redundant threads are executed on separate processor cores. It can not detect all single errors because it does not duplicate all instructions from a single thread. On the other hand, CRT [10], realized as a CMP, enhances SRT and achieves lockstepping. CRTR [5] combines CRT and SRTR.

DIE [12] duplicates and simultaneously issues every instruction. It detects errors by comparing two instructions which originate from a single instruction. Hence, functional unit level space redundancy is exploited. DIE also uses the recovery mechanism for misspredicted branches for the recovery.

2.3. Instruction redundancy

Instruction redundancy [23] is a characteristic of programs: Dynamic instances of a static instruction are executed with the same operand values many times. Thus, if the previous computation is kept in a table, the next same computation can be eliminated by looking up the table. The reuse buffer proposed by Sodani et al. [23] exploits instruction redundancy to boost processor performance. In contrast, we investigated instruction redundancy not for performance gain but for reliability by replacing redundant execution with table lookup [17]. Parashar et al. [11] proposed a similar technique.

3. Reissued-based Fault Detection Mechanism

This section describes the transient fault tolerance mechanism based on the instruction reissue technique [6, 16], which we call in this paper RED (Reissue-based Error Detection) microarchitecture. While RED is applicable to any dynamic instruction schedulers, we will explain it using our proposed instruction reissue mechanism [16], which is based on the RUU [24]. First, we explain how the reissue mechanism works. Then we extend it for detecting transient faults. And last, we discuss how to correct the faults transparently.

3.1. Instruction reissue mechanism for value prediction

This section introduces the instruction reissue mechanism proposed for managing misspeculated data dependences. Interested readers can find more detailed descriptions on the mechanism in [16].

When a data value prediction [4, 8] is performed, the predicted value should be kept in the instruction window. When the instruction finishes and its actual value is obtained, the predicted value has to be compared with the actual one. If they match, the

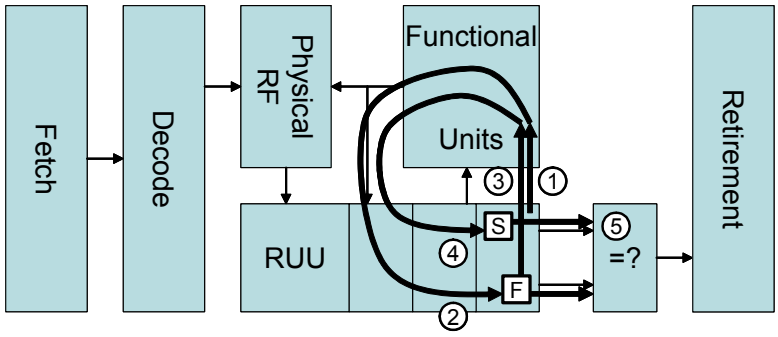

Figure 1: RUU-based RED microarchitecture

prediction is correct. Otherwise, the prediction fails and it is necessary to revert the processor state to a safe point where the speculation is initiated. All instructions dependent upon the mispredicted instructions have to be reissued.

3.2. RED microarchitecture

This section explains how the RED micro-architecture is utilized for detecting errors. Recovery from every error will be discussed in the following section.

In order to detect transient faults, we propose to duplicate committed instructions and compare two results of a single instruction as shown in Figure 1. It is assumed the comparator is fault-free. This is possible by using large strong cells or triple modular redundancy. We use instruction reissue to execute each committed instruction twice.

1. First, every instruction is issued from the RUU to a functional unit, where it is executed. This forms a main thread (M-thread).
2. Second, its execution outcome which is generated for the first time (denoted as F in the figure) is written in the RUU.
3. When the instruction becomes ready for commitment, it is reissued in the RUU and issued into a functional unit again. This forms a redundant thread (R-thread).
4. After that, its execution outcome which is generated for the second time (denoted as S in the figure) is delivered to the RUU again.
5. And last, the two results of the single instruction will be compared with each other. If they do not match, an error is detected.

There is an advantage in limiting only committed instructions to be reissued. When an instruction is ready for commitment, its dependences - both control and data dependences - have been resolved. Therefore, it can be issued unconditionally if an appropriate functional unit is free. Another role of the reissue policy is to ensure that the first and second executions are not effected by the same transient event. Completed instructions stay in the instruction window for a while before they are reissued. This works just like stagger evaluated in [22].

The proposed mechanism detects transient faults occurring in arithmetic and logical functions. We focus on these functions because they are unchecked in modern microprocessors, except for a few products [1, 21]. Since the data cache can be protected by parity or error-correcting code (ECC), it is not necessary to detect faults in the data cache using the instruction reissue. We reissue only the address calculation but perform the memory access once. Caches, register files, and the RUU should be protected using parity or ECC, which is common for modern microprocessors.

RED has the following bottlenecks. Since RED uses the RUU, which combines the ISQ and the ROB into a single structure, instruction scheduler is shared between the M-thread and the R-thread, and the R-thread competes for issue bandwidth and functional units with the M-thread. If the R-thread is handled by a separate scheduler and then if the R-thread fills only in the slack issue bandwidth and functional units left idle by the M-thread, processor performance that diminished by redundant execution will be improved. RED requires the ROB larger than the ISQ to perform flexible scheduling. Figure 2 shows this modification, and the R-thread is issued from the ROB.

1. First, every instruction is issued from the ISQ to a functional unit, where it is executed. This forms an M-thread. Its associated ISQ entry is released.
2. Second, its execution outcome which is generated for the first time (denoted as F in the figure) is written into the ROB.

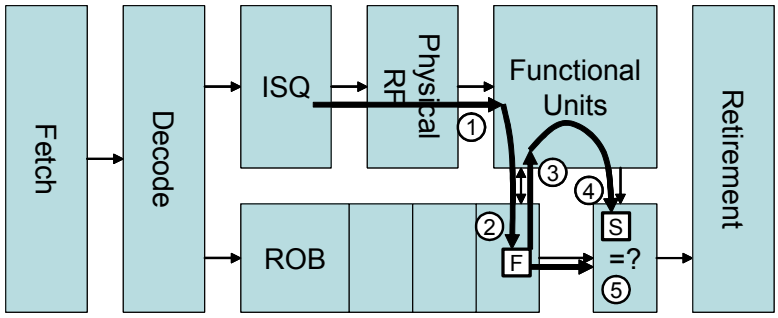

Figure 2: ROB-based RED microarchitecture

Table 1: Processor configuration

OoO Core	128-entry RUU, 64-entry LSQ, 8-wide decode, issue, and retirement
Memory System	64K 2-way L1 I/D caches, 64-byte line size, 3-cycle hit, unified 2M 4-way L2 cache, 64-byte line size, 12-cycle hit, 200-cycle memory access, 4 cache ports
FUs (latency)	8 iALUs (1), 2 iMUL/iDIVs (3/19), 2 fALUs (2), 2 fMUL/fDIVs (4/12), all pipelined except iDIV and fDIV
Branch Predictor	Combining predictor with 64K gshare, 64K bimodal, 64K meta table, 2K 4-way BTB, 64-entry RS, 7-cycle branch misspredict penalty

Table 2: Benchmark programs

Program	Input set	Baseline IPC
164.gzip	input.combined 32	1.76
175.vpr	net.in arch.in	1.93
176.gcc	cp-decl.i	1.94
181.mcf	inp.in	0.90
197.parser	2.1.dict train.in	1.59
255.vortex	lendian.raw	3.28
256.bzip2	input.compressed 8	3.48
177.mesa	mesa.in mesa.ppm	3.29
179.art	c756hel.in a10.img	2.70
183.equake	inp.in	1.34
188.ammp	ammp.in	0.86

3. When the instruction becomes ready for commit, it is reissued into a functional unit from the ROB. This forms an R-thread.
4. After that, its execution outcome which is generated for the second time (denoted as S in the figure) is delivered to the comparator.
5. And last, the two results of the single instruction will be compared with each other. If they do not match, an error is detected.

3.3. Transparent fault recovery

Even when error detection is successful, a system provided with no scheme for recovery will usually hang, causing application failure. Recovery can be handled by either the OS or hardware. In this section, we explain a transparent hardware-based recovery scheme. It uses the existing speculation recovery mechanisms for mispredicted branches [2, 12, 15]. In this paper, we assume every error is transient, and thus instruction retry is successful. That is, when an error is detected by the instruction reissue mechanism, re-executing the fault instruction is sufficient. If we utilize the recovery mechanism for mispredicted branches, the microprocessor flushes its pipeline and then restarts at the corresponding instruction, where an error is detected. All instructions following the faulty instruction are squashed and thus the penalty is very large. However, performance degradation is trivial since the frequency of transient faults is low [20].

4. Evaluation Environment

In this section, we describe our evaluation environment by explaining a processor model and benchmark programs. We implemented a timing simulator using SimpleScalar/PISA tool set [3]. The baseline model is an out-of-order execution superscalar

processor based on the RUU, and its configuration is summarized in Table 1.

The SPEC2000 benchmark suite is used in this study. Table 2 lists the benchmarks and the input sets. For each program, 1 billion instructions are skipped before actual simulation begins. Each program is executed to completion or for 1 billion instructions. Table 2 also shows the baseline IPC when the fault-tolerance mechanism is not included. We count only committed instructions.

5. Simulation Results

First, in this section, we present the baseline RED performance to validate it as a beginning point of our discussion. After that, we compare the two RED microarchitectures; RUU- and ROB-based ones. And last, we further try to improve RED performance.

5.1. Baseline RED performance

Figure 3 presents the performance penalty of the baseline RED which uses the RUU. We use execution cycles for evaluating processor performance and the figure presents the percent increase in execution cycles. This paper does not assume any soft errors occur but we only evaluate performance penalty caused by introducing the error detection mechanism. While the impact of the fault recovery on performance is not evaluated, the overall processor performance is determined by the fault-free behavior [20].

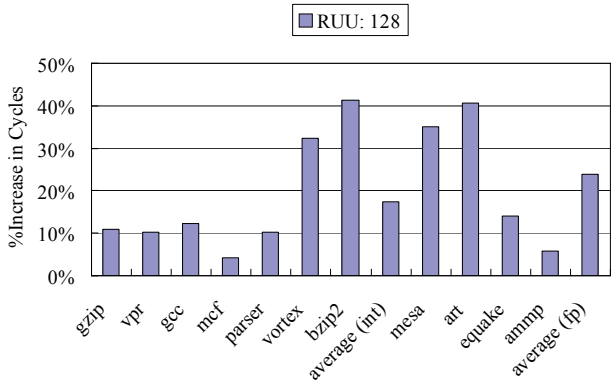

Figure 3: Performance penalty in RUU-based RED

The average penalty is 17.4% on integer and 23.9% on floating-point applications. This is slightly smaller than that found in our previous study [15], where we found the average performance loss of 21.9% on integer applications. This is due to the differences in instruction set architectures and in processor configurations. We also find a slight difference from that reported in DIE study [12], where we see an average penalty of 15% on integer and of 32% on floating-point applications. This may come from difference in benchmark programs. From these observations, we confirm the results shown in Figure 3.

5.2. RED performance on ROB machine

Figure 4 presents the percentage of cycles where the RUU is full. The left bar is for the baseline processor and the right one is for RED. As can be seen, in RED, the RUU is more frequently full than in the baseline, resulting in the increase in the stall cycles. Since the RUU is a combination of the ISQ and the ROB, this effectively reduces both ISQ and ROB capacity.

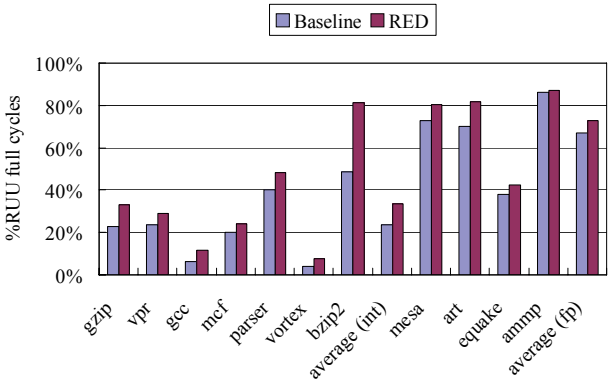

Figure 4: Percentage of RUU full cycles

Thus, next, we adopt the RED microarchitecture on an ROB-based machine and increase ROB capacity to be larger than RUU capacity. We decouple the RUU into the separate ISQ and ROB, and redundant instructions are reissued from the ROB. In this evaluation, we use a 128-entry ISQ and a 512-entry ROB. Figure 5 presents simulation results. The large ROB size contributes to performance as expected. We find the average penalty of 16.0% on integer and of 18.3% on floating-point applications. In the case of 181.mcf, performance is improved. This is because the increase in ROB size is very effective. Note that the baseline processor only has a 128-entry RUU. We find that the performance loss on floating-point applications is considerably reduced.

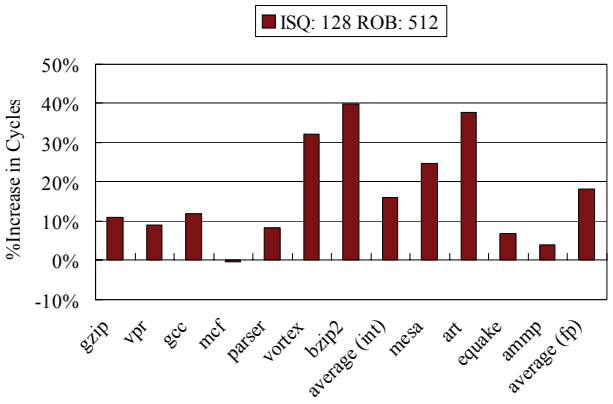

Figure 5: Effect of partitioning RUU into ISQ and ROB

Next, we consider if a much larger ROB could bridge the difference. Figure 6 presents simulation results when RED has a 1024-entry ROB. The larger ROB does not contribute to performance. In the simulation, we did not see any ROB-full cycles. Hence, a further increase in ROB size does not have any influence on performance.

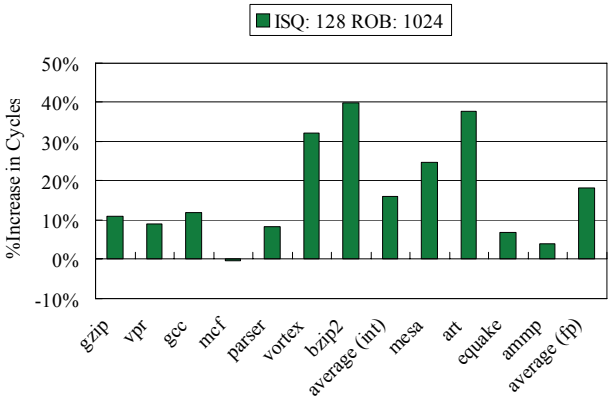

Figure 6: Effect of large ROB

5.3. Effect of priority on M-thread

In RED, the R-thread competes for issue bandwidth and functional units with the M-thread. In order to put a priority on the M-thread, we modify the scheduler so that instructions are redundantly reissued only when idle issue slots are found. Figure 7 presents simulation results after this change. In the simulations, the ROB has 512 entries. Prioritizing the M-thread has a considerable effect on integer performance. We see penalty of only 12.4%.

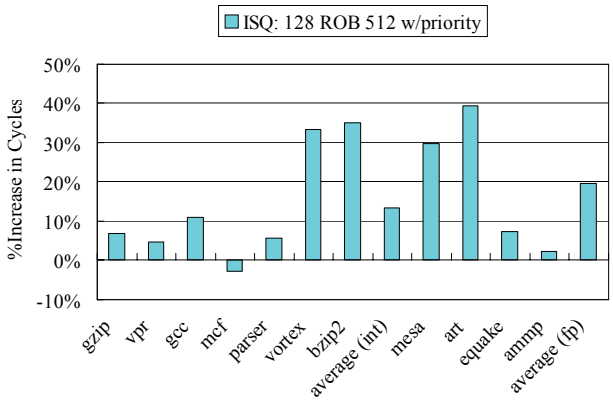

Figure 7: Effect of priority on M-thread

6. Concluding Remarks

The current trend of deep submicron technology will reduce reliability of microprocessors, while it is a key technology to increase their performance. This increases soft errors in LSIs, which result in logic errors. Considering the background, a lot of processor microarchitectures to attack the problem have recently been proposed. Almost all of them exploit some kinds of redundancies: time redundancy, space redundancy, and instruction redundancy.

We have investigated the RED microarchitecture, which utilizes time redundancy. Unfortunately, from our previous studies, we found that it suffers considerable performance penalty.

In this paper, in order to reduce the penalty, we adopted RED into ROB-based processors. This is based on our observation that the performance loss is due to the reduction in the effective ISQ capacity. From detailed simulations, we found that the average penalty is reduced from 17.4% to 16.0% and from 23.9% to 18.3% in integer and floating-point programs, respectively.

We also investigated to put a priority in issue bandwidth on the M-thread. This modification further reduces the average penalty to 12.4% in integer programs.

Acknowledgements

This work is partially supported by Grants-in-Aid for Scientific Research #16300019 and #176549 from Japan Society for the Promotion of Science.

References

[1] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama, "A 1.3-GHz Fifth-Generation SPARC64 Microprocessor," IEEE Journal of Solid-State Circuits, vol.38, no.11, 2003.

[2] T. M. Austin, "DIVA: a Reliable Substrate for Deep Submicron Microarchitecture Design," In Proceedings of 32nd International Symposium on Microarchitecture, 1999.

[3] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," ACM SIGARCH Computer Architecture News, Vol.25, No.3, 1997.

[4] F. Gabbay, "Speculative Execution based on Value Prediction," Technion - Israel Institute of Technology, Technical Report of EE Department #1080, 1996.

[5] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz, "Transient-Fault Recovery for Chip Multiprocessors," In Proceedings of 30th International Symposium on Computer Architecture, 2003.

[6] I. Kim and M. H. Lipasti, "Understanding Scheduling Reply Schemes," In Proceedings of 10th International Symposium on High Performance Computer Architecture, 2004.

[7] S. Kim and A. K. Somani, "SSD: an Affordable Fault Tolerant Architecture for Superscalar Processors," In Proceedings of 8th International Symposium on Dependable Computing, 2001.

[8] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. "Value Locality and Load Value Prediction," In Proceedings of 7th International Conference on Architectural Support for Programming Languages and Operating Systems, 1996.

[9] A. Mendelson and N. Suri, "Designing High-Performance & Reliable Superscalar Architectures - the Out of Order Reliable Superscalar (O3RS) Approach," In Proceedings of 1st International Conference on Dependable Systems and Networks, 2000.

[10] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed Design and Implementation of Redundant Multithreading Alternatives," In Proceedings of 29th International Symposium on Computer Architecture, 2002.

[11] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam, "A Complexity-Effective Approach to ALU Bandwidth Enhancement for Instruction-Level Temporal Redundancy," In Proceedings of 31st International Symposium on Computer Architecture, 2004.

[12] J. Ray, J. C. Hoe, and B. Falsafi, "Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery," In Proceedings of 34th International Symposium on Microarchitecture, 2001.

[13] S. K. Reinhardt and S. S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," In Proceedings of 27th International Symposium on Computer Architecture, 2000.

[14] E. Rotenberg, "AR-SMT: a Microarchitectural Approach to Fault Tolerance in Microprocessors," In Proceedings of 29th Fault-Tolerant Computing Symposium, 1999.

[15] T. Sato and I. Arita, "In Search of Efficient Reliable Processor Design," In Proceedings of 30th International Conference on Parallel Processing, 2001.

[16] T. Sato, "Evaluating the Impact of Reissued Instructions on Data Speculative Processor Performance," Microprocessors and Microsystems, Vol.25, Issue 9, 2002.

[17] T. Sato, "Exploiting Instruction Redundancy for Transient Fault Tolerance," In Proceedings of 18th International Symposium on Defect and Fault Tolerance in VLSI Systems, 2003.

[18] T. Sato, "Exploiting Sub-Word Parallelism for Dependable Processors," In Proceedings of 5th International Conference on Automation & Information, 2004.

[19] T. Sato, "Exploiting Trivial Computation in Dependable Processors," In Proceedings of 20th International Conference on Computers and Their Applications, 2005.

[20] T. Sato and A. Chiyonobu, "Evaluating the Impact of Fault Recovery on Superscalar Processor Performance," 12th International Symposium on Pacific Rim Dependable Computing, 2006.

[21] T. J. Slegel, E. Pfeffer, and J. A. Magee, "The IBM eServer z990 microprocessor," IBM Journal of Research and Development, Vol.48, No.3/4, 2004.

[22] J. C. Smolens, J. Kim, J. C. Hoe, B. Falsafi, "Efficient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitectures," In Proceedings of 37th International Symposium on Microarchitecture, 2004.

[23] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse," In Proceedings of 24th International Symposium on Computer Architecture, 1997.

[24] G. S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional unit, pipelined computers," IEEE Transactions on Computers, Vol.39, No.3, 1990.
[25] A. K. Somani and J. Nickel, "REESE: a Method of Soft Error Detection in Microprocessors," In Proceedings of 2nd International Conference on Dependable Systems and Networks, 2001.
[26] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream Processors: Improving Both Performance and Fault Tolerance," In Proceedings of 9th International Conference on Architectural Support for Programming Languages and Operating Systems, 2000.
[27] T. N. Vijaykumar, I. Pomeranz, and K. K. Cheng, "Transient Fault Recovery Using Simultaneous Multithreading," In Proceedings of 29th International Symposium on Computer Architecture, 2002.