In Search of Efficient Reliable Processor Design

Toshinori Sato[1,2] Itsujiro Arita[1]
[1] Department of Artificial Intelligence
[2] Center for Microelectronic Systems
Kyushu Institute of Technology
{tsato,arita}@ai.kyutech.ac.jp

Abstract

In this paper, we investigate an efficient reliable processor, which can detect and recover from transient faults. There are two driving force to study fault-tolerant techniques for microprocessors. One is deep submicron fabrication technologies. Future semiconductor technologies could become more susceptible to alpha particles and other cosmic radiation. The other is increasing popularity of mobile platforms. Recently cell phones are used for applications which are critical to our financial security, such as flight ticket reservation, mobile banking, and mobile trading. In such applications, it is expected that computer systems will always work correctly. From these observations, we have proposed a mechanism which is based on instruction reissue technique for incorrect data speculation recovery and utilizes time redundancy. In order to mitigate overhead caused by including fault-tolerant facility, we evaluate some alternative designs and find that speculatively updating branch predictors and removing redundant memory accesses are very effective.

1 Introduction

Modern microprocessors have limited hardware error detection, especially for transient faults, which are the vast majority of hardware failures [19]. Transient faults are random events which occur when various noise sources cause an incorrect result. For example, alpha particles and other cosmic radiation can alter the state of latches and dynamic logic, resulting in logic errors. Currently, the frequency of the transient faults is low. However, small device size, increasing transistor counts, high clock frequency, and low power supply, which are accompanied with deep submicron technology, do not only reduce noise margin and reliability but also increase the impact of defects [12]. On the other hand, portable and mobile computer devices such as laptop and cell phone are being widely spread and will be one of the major platforms for world-wide distributed computing. For example, Java is already work on cell phones [7]. Currently, we are provided several service contents such as flight ticket reservation, mobile banking, and mobile trading with a cell phone [7]. For these applications, reliability and dependability are very important. From these considerations, it is expected that even low-end microprocessors should be fault-tolerant.

Current fault-tolerant techniques utilized in commercial systems such as IBM S/390 G5 [19] and Compaq NonStop Himalaya [3] are based on redundancies. For example, error checking is implemented by duplicating chips and comparing outputs. These techniques require two times or more hardware overhead. In addition, the duplicate and compare is adequate for only error detection. The other example is parity or error-correcting code (ECC). Contemporary microprocessors use parity for caches. However, they leave control, arithmetic, and logical functions unchecked, since it is difficult and time-consuming to check these components [19]. Hence, low-cost and simple fault-tolerant technique is necessary for future microprocessors.

Recently, we have investigated the use of instruction reissue technique as a hardware mechanism to detect and recover from transient faults [15]. Originally, the instruction reissue mechanism is proposed for incorrect data speculation recovery. We modify and apply the mechanism for fault-tolerance. Since future microprocessors will utilize data speculation and include the instruction reissue mechanism [5], this fault-tolerant mechanism costs the least hardware overhead. In addition, it is simple because detection of transient faults and their recovery processes are done simultaneously with dynamic instruction scheduling. Unfortunately, the proposed mechanism causes significant performance loss. Thus, in this paper we investigate efficient fault-tolerant mechanism based on the previous proposal.

The rest of this paper is organized as follows. Section 2 surveys related work. Section 3 describes our evaluation methodology. Section 4 explains that the instruction reissue mechanism is applicable to fault-tolerance, and explores design space to enhance the original proposal. Section 5 discusses our simulation results. Finally, Section 6 presents our conclusions.

2 Related Work

Data speculation [4, 6] is a technique which executes instructions speculatively using predicted data values. Data dependences are speculatively resolved and thus instruction level parallelism is increased. When a predicted value is correct, it becomes possible to execute the predicted instruction and its dependent instructions simultaneously, thereby more instruction level parallelism is extracted. Otherwise, it is necessary to revert processor state to a safe point where the speculation is initiated. Instruction reissue is such a technique recovering the processor state when a misspeculation occurs. It invalidates instructions dependent upon the misspeculated instruction selectively and then reissues them in instruction window. Lipasti et al. [6] introduce the instruction reissue concept. Instructions dependent upon a predicted instruction are forced to retain in reservation stations. When the predicted instruction produces an actual value, the predicted value must be compared with the actual one. If they match, the prediction is correct and the dependent instructions release the reservation stations. If

the prediction fails, all dependent instructions are concurrently invalidated and reissued. However, they only propose the concept of the instruction reissue. They do not presented any practical implementation of the scheme. If the scheme were implemented, the processor cycle time would increase since it would be very difficult to find in parallel all dependent instructions using moderate hardware cost. We propose a practical implementation of the instruction reissue in [13, 14].

IBM S/390 G5 [19] is a commercial fault-tolerant microprocessor. However, it relies on traditional fault-tolerant techniques and thus is not adequate for the mobile applications. HaL SPARC64 processor [16] has fault detection mechanism for caches and memories but does not for functional units.

A similar idea of using time redundancy for detecting transient faults is utilized in AR-SMT [11] and SRT [10] processors. They are both based on simultaneous multithreading (SMT) processors [20]. An SMT processor can execute multiple threads simultaneously and thus is attractive for fault detection since two redundant copies of a single thread are executed on the SMT to detect faults by comparing two results. While AR-SMT and SRT processors exploit the characteristics, they only detect transient faults but can not recover from the faults transparently. The recovery should be supported by OS. In addition, generating two redundant threads also requires OS support. Hence, transient fault detection based on the AR-SMT or SRT is not transparent.

Rashid et al. [9] propose the other time redundancy technique suitable for Multiscalar architecture [18]. A Multiscalar processor can execute multiple tasks on several processing units. Hence, it is possible to re-execute committed instructions on an idle processing unit while the remainder of the units execute the program. Multiscalar architecture requires that programs should be re-compiled in order for several tasks to be executed simultaneously, and thus the fault detection technique is not applicable to legacy binaries. In addition, since redundant instructions are executed on the independent processing unit, the technique relies not only on time redundancy but also on space redundancy.

DIVA [1] is an example of a fault-tolerant microprocessor based on space redundancy. A simple checker processor is used for dynamically verifying committed instructions. Any hardware faults are corrected using recovery mechanism for incorrect branch predictions. Hence, DIVA is a hardware-based mechanism and transparent. However, DIVA requires additional ports for register files and caches in order for the checker processor to share processor contexts. This increases design complexity and circuit delay of its main processor.

3 Evaluation Environment

In this section, we describe the evaluation environment by explaining a processor model and benchmark programs.

3.1 Processor model

We implemented a timing simulator using SimpleScalar/ Alpha tool set (ver.3.0a) [2]. The baseline processor models are realistic 4- and 8-way out-of-order execution superscalar processors. Dynamic instruction scheduling is based on register update unit (RUU) [17] which has 64 entries. Each functional unit can execute any operations. The latency for execution is 1 cycle except in the case of multiplication (4 cycles) and division (12 cycles). A non-blocking, 128KB, 32B block, 2-way set-associative L1 data cache is used for data supply. The numbers of ports are 2 and 4 in the 4- and 8-way superscalar processor models respectively. It has a load latency of 1 cycle after the data address is calculated and a miss latency of 6 cycles. It has a backup of an 8MB, 64B block, direct-mapped L2 cache which has a miss latency of 18 cycles for the first word plus 2 cycles for each additional word. No memory operation can execute that follows a store whose data address is unknown. A 128KB, 32B block, 2-way set-associative L1 instruction cache is used for instruction supply and also has the backup of the L2 cache which is shared with the L1 data cache. For control prediction, a 1K-entry 4-way set associative BTB, a 4K-entry gshare-type 2-level adaptive branch predictor, and an 8-entry return address stack are used. The branch predictor is updated at instruction commit stage.

3.2 Benchmark programs

The SPEC2000 CINT benchmark suite is used for this study. Table 1 lists the benchmarks and the input sets. We use the object files provided by University of Michigan. For each program except for `252.eon`, 1 billion instructions are skipped before actual simulation begins. Each program is executed to completion or for 100 million instructions. Table 1 also shows the execution cycles. We count only committed instructions.

Table 1: Benchmark programs

program	input set	#cycles	
		4-way	8-way
164.gzip	input.compressed	43,184,527	31,369,708
175.vpr	net.in arch.in	50,290,784	39,818,320
176.gcc	cccp.i	59,738,067	47,809,252
186.crafty	crafty.in	47,027,477	33,373,516
197.parser	test.in	52,442,075	40,826,549
252.eon	chair	41,150,918	27,141,224
255.vortex	lendian.raw	44,864,197	30,352,704
256.bzip2	input.random	34,787,182	18,740,831

4 Transient Fault-Tolerant Mechanism

First in this section, we explain how the instruction reissue mechanism [13, 14] is applied for tolerating transient faults [15]. Then, we evaluate the base fault-tolerant mechanism. And last, we explore efficient alternative designs.

4.1 Base fault-tolerant mechanism

In order to detect transient faults, we propose to duplicate committed instructions and compare two results of a single instruction. It is assumed the comparator is fault free. This is possible by using large strong cells [16] or triple modular redundancy. We use the instruction reissue to execute each committed instruction twice. When every instruction becomes ready for commitment, it is reissued in the RUU and dispatched into a functional unit again. Its execution outcome which is generated for the first time is held in the RUU and will be compared with its equivalent one when it is generated for the second time. If they do not match,

a transient fault is detected. Note that any bookkeeping must not be done in load/store queue, since every store instruction does not change memories until it is committed. The proposed mechanism detects transient faults occurred in datapaths and their control logics, and then the sphere of replication [10] is as depicted in Figure 1. We focus on these functions because they are unchecked in modern microprocessors [19]. Instruction cache, register files, and the RUU should be protected using parity or ECC, that is common for modern microprocessors. Recovery from the fault will be described later in this section.

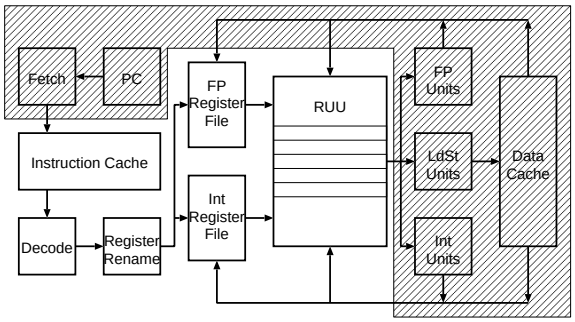


Figure 1: Sphere of replication for base mechanism

There is an advantage of limiting that only committed instructions are reissued. When each instruction is ready for commitment, its dependences — both control and data dependences — have been resolved. Therefore, it can be dispatched unconditionally if an appropriate functional unit is free. This utilizes dispatch bandwidth efficiently. 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 the delay queue in the AR-SMT processor [11]. On the other hand, a possible disadvantage of the reissue policy is that the delay of updating branch prediction table diminishes branch prediction accuracy. Another one is increasing pressure on the effective instruction window capacity. We will evaluate these influence on processor performance later in this section.

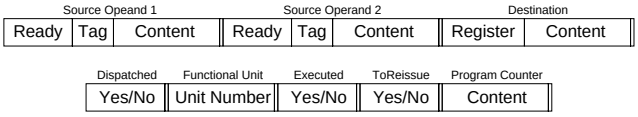


Figure 2: Enhanced RUU entry for fault-tolerance

Figure 2 depicts the enhanced RUU entry which is applied for tolerating transient faults. Each entry of the RUU has two source operand fields, a destination field, a dispatched bit, a functional unit field, an executed bit, a to-reissue bit, and a program counter field. If a source operand is not ready, the ready bit is reset to indicate that the source operand is not available and a tag for the operand is obtained. When the operand is ready, the content of the source register is held in the source operand field and the ready bit is set. The destination register number renamed is held in the destination field, and an execution result is also held in the destination field when the execution completes. The dispatched bit indicates if the instruction is dispatched into a functional unit which is specified by the functional unit field. When the instruction finishes, its execution result is held in the destination field and the executed bit is set. The to-reissue bit is set when the associated instruction is dispatched into a functional unit for the first time. When the instruction becomes ready to commit, the to-reissue bit is checked. If it is set, the instruction is reissued in the RUU and the to-reissue bit is reset. Otherwise, the instruction begins to commit. Lastly, the program counter field is used for the correction of misprediction and precise interrupts.

Even when the fault detection is successful, a system with no recovery will usually hang, causing application down. We propose two transparent hardware-based recovery schemes. One uses the existing speculation recovery mechanism for mispredicted branches, and the other is based on the instruction reissue mechanism for incorrect data speculation. In this paper, we assume the failure is transient, and thus instruction retry is successful. That is, when any faults are detected by the instruction reissue mechanism, it is enough to re-execute the fault instruction again. If we utilize the recovery mechanism for mispredicted branches, the microprocessor flushes its pipeline and then restarts at the corresponding instruction when a transient fault is detected. All instructions following the fault instruction are squashed and thus its penalty is very large. However, it is expected that performance degradation is small since the frequency of transient faults is low.

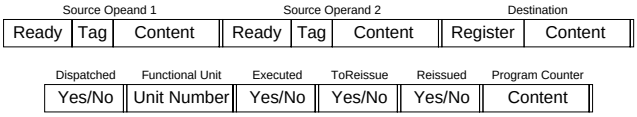
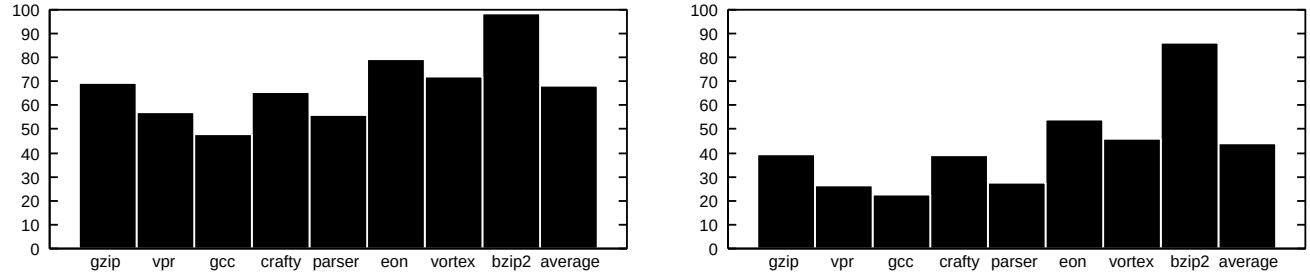


Figure 3: Enhanced RUU entry for fault-tolerance

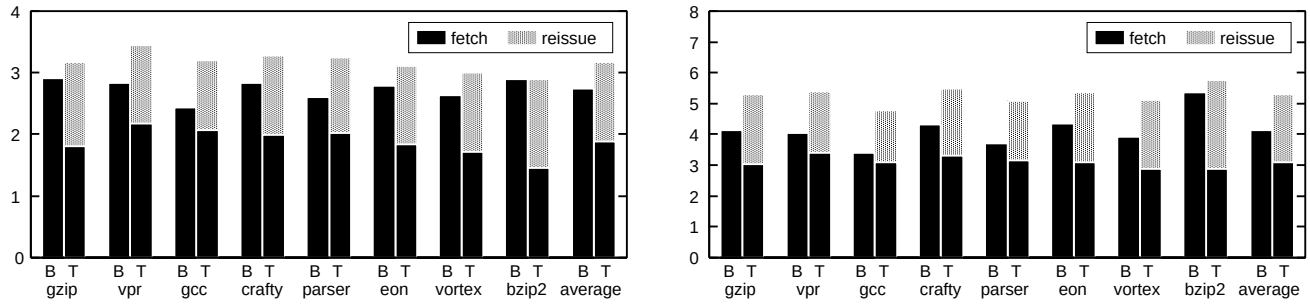
Figure 3 shows the enhanced RUU entry which is extended for transparent fault recovery utilizing the instruction reissue. The additional reissued bit indicates if transient faults are detected so that the corresponding instruction should be reissued. When the instruction produces an execution result for the second time, the signal indicating if a transient fault is detected, is also broadcasted. Here, we call this signal mispredicted signal[1] . The detection of the transient fault is performed by comparing two execution results associated to the single instruction. In the case that the fault is not detected, no action is performed since the following instructions have obtained their source operands when the instruction finishes for the first time. Otherwise, the fault instruction and its dependent instructions should be invalidated and be reissued again if the corresponding dispatched bit is set. The corresponding dispatched and executed bits are reset, and the reissued bit is set. When the reissued instruction finishes, the mispredicted signal is asserted. Namely, the mispredicted signal indicates if the fault or reissued instructions finish. Please note the difference between the to-reissue and reissued bits. The reissued bit means that the associated instruction is fault or reissued so that its dependent instructions should be reissued.

[1] This is because the signal is originally used for data prediction [13,14], and a fault is regarded as a misspeculation.



(i)4-way superscalar (ii)8-way superscalar

Figure 4: %Increase of execution cycles



(i)4-way superscalar (ii)8-way superscalar

Figure 5: Number of dispatched instructions per cycle

In contrast, the to-reissue bit means the associated instructions executed only once and thus it should be reissued. The following process is as same as above. The instructions dependent on the reissued instruction also have to be reissued. In this manner, all instructions dependent upon the fault instruction are searched and reissued serially. This scheme introduces only one signal per completed instruction into the dynamic instruction scheduling mechanism. The signal decides the action resulted from the associative search which examines if the source and destination tags match. The signal is asserted when a fault or reissued instruction finishes. If the signal is not asserted, the dynamic instruction scheduling mechanism performs usual process. Otherwise, it detects instructions which have to be reissued. This mechanism is very simple, and hence it requires only slight hardware overhead.

4.2 Performance overheads

Figures 4(i) and (ii) show performance degradation because of duplicating all committed instructions in the cases of 4- and 8-way superscalar processors respectively [15]. We use execution cycles for evaluating processor performance and the figures present the percent increase of execution cycles. This paper assumes that we do not detect any transient faults but we only evaluate performance penalty caused by introducing the fault-tolerant mechanism. Therefore, the impact of the fault recovery mechanism on performance is not evaluated. As can be easily seen, performance of 4-way superscalar is diminished significantly, while it is smaller than the case where the program is executed two times and the results are compared. It is reduced by an average of 68.2%. On the other hand, performance degradation of 8-way superscalar is moderate, and an average of 44.2%.

One of the reasons why 4-way superscalar suffers severer impact on performance is considerable resource shortage. Figure 5 presents how the number of dispatched instructions per cycle increases. Please note that the number includes not only committed instructions but also squashed instructions due to mispredicted branches. For each group of two bars, the left (denoted as `B`) indicates the number of dispatched instructions per cycle for the baseline processor, which does not utilize the fault-tolerant mechanism. The right (denoted as `T`) is for our proposal, and is divided into two parts. The lower part indicates the number of fetched instructions and the upper indicates that of reissued instructions. While the fetched instructions consist of the committed and squashed instructions, the reissued instructions only include the committed instructions. In the case of 4-way superscalar, duplicating committed instructions increases the efficiency of dispatch by only 0.43 instructions per cycle. In other words, pressure on functional units becomes serious since the fetched and reissued instructions struggle for the limited resources. On the other hand, 8-way superscalar improves the dispatch efficiency by 1.14 instructions per cycle since it has more functional units for the reissued instructions than 4-way superscalar. And thus, 8-way superscalar suffers less performance loss than 4-way superscalar.

The other reason is increasing latency of instructions until commitment. Each reissued instruction stays in the instruction window longer than in the case of the baseline processor. One of the influences of the delay is degrading

Table 2: Branch prediction accuracy and commitment latency

program	4-way				8-way			
	%br pred		latency (cycles)		%br pred		latency (cycles)	
	base	reissue	base	reissue	base	reissue	base	reissue
164.gzip	93.23	92.87	14.70	30.49	93.14	92.89	11.63	17.99
175.vpr	80.43	79.96	9.25	19.66	79.66	79.14	7.72	11.86
176.gcc	82.54	81.41	7.95	17.21	81.97	81.14	6.39	10.37
186.crafty	86.82	86.36	11.00	23.68	86.67	86.16	7.67	13.33
197.parser	88.00	87.35	10.33	21.50	87.64	87.22	8.03	12.71
252.eon	93.22	92.57	13.38	27.49	92.96	92.78	8.55	15.74
255.vortex	91.82	92.20	13.65	29.90	92.07	91.87	9.06	16.50
256.bzip2	99.89	99.89	21.50	43.83	99.89	99.89	10.97	22.06

branch prediction accuracy because the state of branch prediction tables is updated late. Table 2 summarizes branch prediction accuracies and clock cycles where each instruction stays in the RUU. The commitment latency is significantly increased especially for 4-way superscalar, resulting in the slight decrease of branch prediction accuracy. It is generally known that even slight decrease of branch prediction accuracy affects modern superscalar processor performance seriously. This increases the execution cycles of the processor models utilizing the proposed fault-tolerant mechanism.

4.3 Efficient fault-tolerance design

In this section, we investigate to mitigate overhead caused by introducing the fault-tolerant mechanism. We focus on two classes of instructions. One is branch instruction and the other is load instruction.

As mentioned in the previous section, branch prediction accuracy is diminished due to late update of the prediction tables. When we update the tables speculatively at the earlier pipeline stages, this problem may be solved. Instead of updating at the commit stage, we propose to update the tables at the decode or writeback stages. At the decode stage, any branch outcome has not been known, and thus the update is based on the current predictions of branches. If branch prediction accuracy is considerably low, this speculative update causes further degradation of prediction accuracy. On the other hand, all branch instructions are resolved at the writeback stage. Only squashed branches affect the prediction tables wrongly. However, the update is significantly late. We will evaluate these speculative updates using the timing simulator in the following section.

Load instructions would diminish performance of the proposed fault-tolerant processor due to the following two reasons. First, there may be resource shortage of data cache ports. For example, while 8-way superscalar has the ability to dispatch eight instructions per cycle, it has only four data cache ports, for which the fetched and reissued instructions might struggle. Second, execution latency of a load instruction is longer than any others, while reissued load instructions may always hit in the L1 cache. An execution of a load consists of address calculation and memory access. This will have pressure on RUU capacity. Based on these observations, we investigate to eliminate the probable bottlenecks. First, we evaluate how doubling data cache ports affects processor performance. Second, we remove redundant memory accesses performed by reissued load instructions. The original mechanism executes every load instruction twice. Both address calculation and memory access are performed two times. Since data cache can be protected by parity or ECC, it is not necessary to detect faults in data cache using the instruction reissue. We reissue only the address calculation but perform the memory access once. Thus, the sphere of replication is revised as depicted in Figure 6. While data cache should be protected by parity or ECC, that is common for modern microprocessors. One of the possible demerits of the revise is that any faults on data cache busses can not be detected.

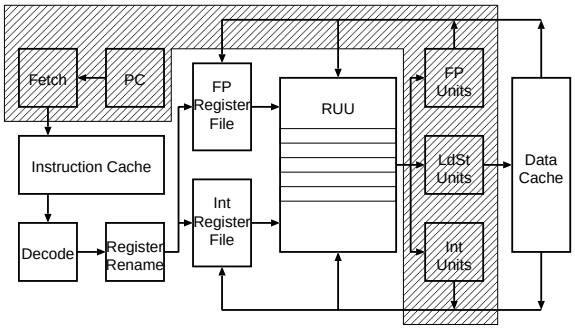


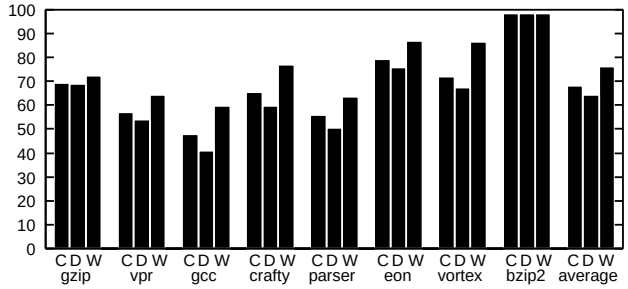
Figure 6: Sphere of replication for revised mechanism

5 Simulation Results

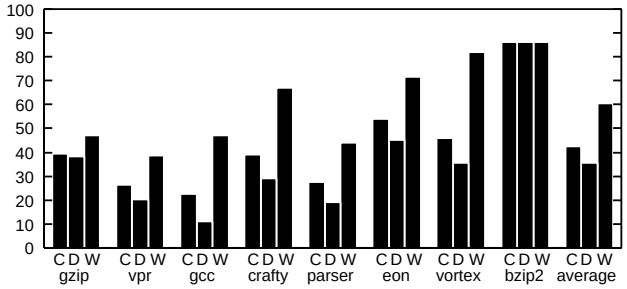
This section presents our simulation results. First, design alternatives explained in the previous section is evaluated. And then, our proposal will compared with a space redundant technique based on a chip multiprocessor [8].

5.1 Considering branch instructions

First, we evaluate the speculative update of branch prediction tables. Figure 7 presents simulation results. Layout and subject matter of the figure are the same as for Figure 4. For each group of three bars, the first (denoted as `C`) indicates the increase of execution cycles over baseline model, which does not utilize fault-tolerant mechanism, when the prediction tables are updated at the commit stage, and the remaining two bars (denoted as `D` and `W` respectively) indicate those where they are updated at the decode and writeback stages respectively. As can be easily observed, the speculative update at the decode stage mitigates the overhead by 5.5% and 16.5% on average for 4- and 8-way models

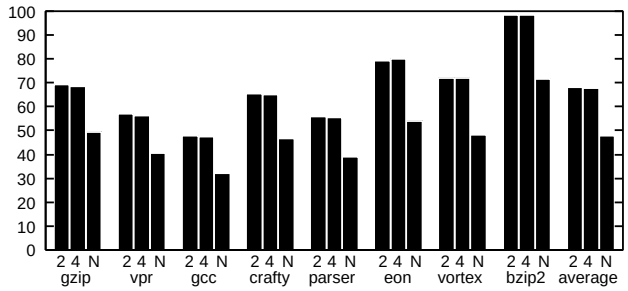


(i)4-way superscalar

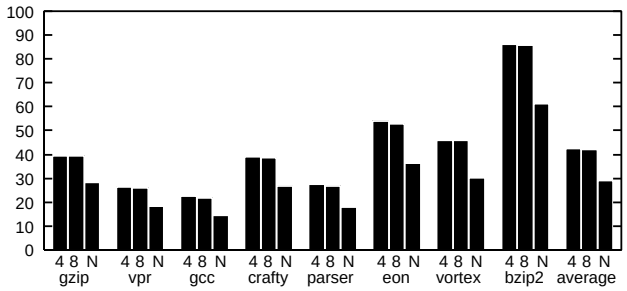


(ii)8-way superscalar

Figure 7: %Influence of speculatively updating branch predictors



(i)4-way superscalar



(ii)8-way superscalar

Figure 8: %Influence of increasing cache ports and removing redundant memory accesses

respectively. This means that fast update of the prediction tables overwhelms the possible degradation of prediction accuracy due to the speculative update. On the other hand, the speculative update at the writeback stage increases the average overhead by 18.2% and 41.9% for 4- and 8-way models respectively. Since the update at the writeback stage is significantly late and still speculative, processor suffers both two demerits.

Table 3 shows branch prediction accuracies in the cases of the speculative updates. Different from our expectation that the branch prediction accuracies might be diminished, they are always improved if the prediction tables are updated at the decode stage. On the other hand, the speculative updates at the writeback stage degrades the accuracies significantly. The results confirm the observations above.

In summary, the speculative updates of the branch prediction tables at the decode stage is efficient for mitigating overhead caused by introducing the fault-tolerant mechanism.

5.2 Considering memory accesses

Figure 8 explains how memory accesses affect processor performance. Layout and subject matter of the figure are the same as for Figure 4. For each group of three bars, the first (denoted as `2` or `4`) indicates the increase of execution cycles over baseline model for the original reliable processors, the second one (denoted as `4` or `8`) indicates that when data cache has twofold ports of the former processor models, and the last (denoted as `N`) indicates that when redundant memory accesses are eliminated. First, doubling data cache ports

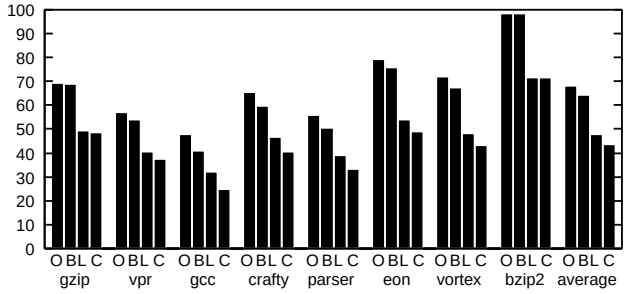
Table 3: %Branch prediction accuracy

	4-way		8-way	
program	ID	WB	ID	WB
164.gzip	93.67	89.55	93.63	87.85
175.vpr	83.86	76.05	83.80	73.70
176.gcc	87.97	74.34	87.28	72.26
186.crafty	91.66	78.41	91.33	75.41
197.parser	91.34	83.79	90.99	82.41
252.eon	97.29	88.38	97.41	85.58
255.vortex	96.21	84.12	95.80	82.89
256.bzip2	99.89	99.89	99.89	99.82

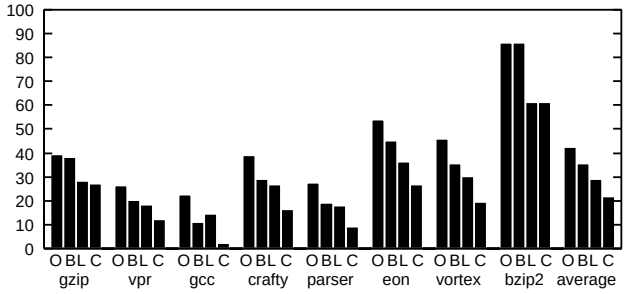
hardly affects processor performance. This means that there are almost no resource shortage for data cache ports in the original models. The results are desirable since doubling the ports needs high hardware cost. Further reduction of the ports is remained as the future study. Second, the elimination of redundant memory accesses reduces the overhead by 30.0% and 31.3% on average for 4- and 8-way models respectively. This means that the long execution latency is one of the main sources of the overhead. In other words, the pressure on the RUU capacity should be mitigated.

Table 4 summarizes clock cycles where each instruction stays in the RUU. By comparison with Table 2, it can be seen that the elimination of redundant memory accesses reduces the latency significantly while the doubling has no effects on the latency. The results confirm the observations above.

In summary, reducing redundant memory accesses of the



(i)4-way superscalar



(ii)8-way superscalar

Figure 9: %Influence of combination of two techniques

Table 4: Commitment latency (cycles)

program	4-way twofold ports	4-way no redundancy	8-way twofold ports	8-way no redundancy
164.gzip	30.14	26.11	18.01	16.23
175.vpr	19.60	16.93	11.78	10.60
176.gcc	17.03	14.17	10.22	08.97
186.crafty	23.57	20.16	13.27	11.63
197.parser	21.37	18.34	12.65	11.19
252.eon	27.32	23.10	15.88	13.54
255.vortex	29.97	25.35	16.43	14.14
256.bzip2	43.83	37.82	22.01	19.05

reissued load instructions is efficient for mitigating overhead caused by introducing the fault-tolerant mechanism.

5.3 Combination of two techniques

Figure 9 presents synergetic effect on processor performance when the speculative update at the decode stage and the elimination of redundant memory accesses are combined. Layout and subject matter of the figure are the same as for Figure 4. For each group of four bars, the first (denoted as `O`) indicates the increase of execution cycles over baseline model for the original reliable processors, the next two bars (denoted as `B` and `L` respectively) indicate those when either the speculative update or the elimination of redundant memory accesses is applied, and the last (denoted as `C`) indicates that both techniques are applied. It is found that the combination is always better than each single technique. It reduces the overhead by 36.0% and 48.6% on average for 4- and 8-way models respectively.

Based on the considerations in the previous section and the evaluation in this section, we can mitigate the overhead caused by the proposed fault-tolerant mechanism significantly, and the average increases of execution cycle time over the baseline model which does not have fault-tolerant capability are only 43.6% and 21.9% in 4- and 8-way superscalar processors respectively.

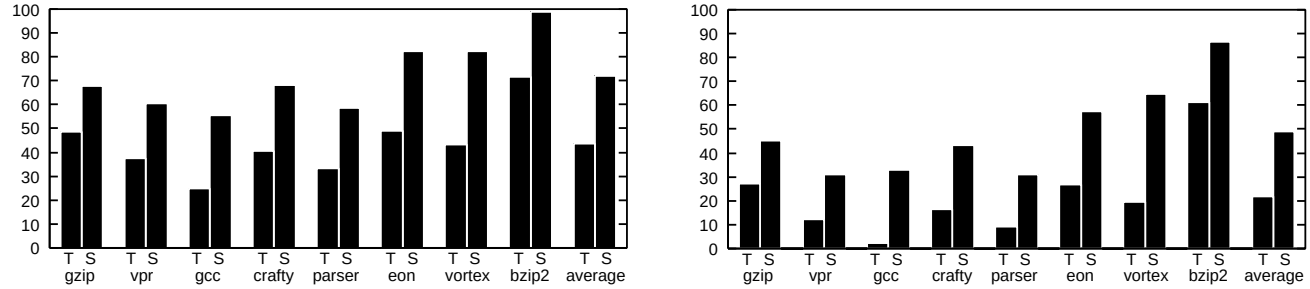
5.4 Comparison with chip multiprocessor

Next, we compare our proposal with a space redundant technique based on a chip multiprocessor [8]. The chip multiprocessor is a complexity-effective solution for large scale microprocessors in the future. In this evaluation, we use the chip multiprocessor consisting of two smaller superscalar processors. They execute an identical program respectively and two outcomes of a single instruction is compared. We compare 4-way superscalar with the chip multiprocessor consisting of two 2-way superscalars, and 8-way superscalar with the one consisting of two 4-way superscalars. Hardware resources of the smaller superscalars are half of the baseline model. Please note that only space redundant technique can detect errors in caches, register files, and the RUU. However, they can be protected using parity or ECC in the case of the time redundant technique. Figure 10 presents the results. Layout and subject matter of the figure are the same as for Figure 4. The execution cycles are used for the comparison. For each group of two bars, the left (denoted as `T`) is for the proposed model utilizing time redundancy, and the right (denoted as `S`) is for the chip multiprocessor utilizing space redundancy. Please note that the penalties caused by synchronizing two component processors in the chip multiprocessor are not considered. Hence, the simulation results are favorable for the chip multiprocessor. It can be easily observed that the proposed time redundant technique suffers less performance degradation than the space redundant technique.

6 Concluding Remarks

There are two driving force to investigate fault-tolerant techniques for microprocessors. One is deep submicron fabrication technologies. Future semiconductor technologies could become more susceptible to alpha particles and other cosmic radiation. The other is increasing popularity of mobile platforms. Recently cell phones are used for applications which are critical to our financial security, such as flight ticket reservation, mobile banking, and mobile trading. In such applications, it is expected that computer systems will always work correctly. From these observations, we have proposed a fault-tolerant mechanism for future microprocessors. It is based on the instruction reissue technique for incorrect data speculation recovery and utilizes time redundancy. In order to mitigate overhead caused by including fault-tolerant facility, we have evaluated some alternative designs and found that speculatively updating branch predictors and removing redundant memory accesses are very effective.

One of the future studies regarding the fault-tolerant mechanism for microprocessors is evaluating the impact of the transparent fault recovery mechanism on processor performance. We should construct a transient faults model for



(i)4-way superscalar (ii)8-way superscalar

Figure 10: %Time redundancy versus space redundancy

mobile environment. We should also investigate more practical mechanisms than that proposed in this paper, since the RUU is not always reasonable design to implement. Currently, we are studying on simple instruction reissue mechanism.

Acknowledgments

The authors are grateful to anonymous reviewers whose comments and suggestions helped to improve the quality of this paper. This work is supported in part by the Grant-in-Aid for Scientific Research #13558030 from Japan Society for the Promotion of Science.

References

[1] T.M.Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," Int. Symp. on Microarchitecture, 1999.

[2] D.Burger, T.M.Austin, "The SimpleScalar tool set, version 2.0," ACM SIGARCH Computer Architecture News, vol.25, no.3, 1997.

[3] Compaq Computer Corp., "Data integrity for Compaq NonStop Himalaya servers," White paper, 1999.

[4] F.Gabbay, "Speculative execution based on value prediction," Technical Report #1080, Department of Electrical Engineering, Technion, 1996.

[5] Intel Corp., "Inside the NetBurst micro-architecture of the Intel Pentium 4 processor," White paper, 2000.

[6] M.H.Lipasti, C.B.Wilkerson, J.P.Shen, "Value locality and load value prediction," Int. Conf. on Architectural Support for Programming Languages and Operating Systems, 1996.

[7] NTT DoCoMo Inc., "NTT DoCoMo to launch new i-mode service based on Java technology," `http://www.nttdocomo.com/new/contents/01/whatnew0118a.html`, 2001.

[8] K. Olukotun, B.A. Nayfeh, L. Hammond, K. Wilson, K. Chang, "The case for a single-chip multiprocessor," Int. Conf. on Architectural Support for Programming Languages and Operating Systems, 1996.

[9] F.Rashid, et al., "Fault tolerance through re-execution in Multiscalar architecture," Int. Conf. on Dependable Systems and Networks, 2000.

[10] S.K.Reinhardt, S.S.Mukherjee, "Transient fault detection via simultaneous multithreading," Int. Symp. on Computer Architecture, 2000.

[11] E.Rotenberg, "AR-SMT: a microarchitectural approach to fault tolerance in microprocessors," Fault-Tolerant Computing Symp., 1999.

[12] P.I.Rubinfeld, "Managing problems at high speed," IEEE Computer, vol.31, no.1, 1998.

[13] T.Sato, "Analyzing overhead of reissued instructions on data speculative processors," Workshop on Performance Analysis and its Impact on Design held in conjunction with Int. Symp. on Computer Architecture, 1998.

[14] T.Sato, "Data dependence speculation using data address prediction and its enhancement with instruction reissue," Euromicro Conf., Workshop on Digital System Design: Architectures, Methods and Tools, 1998.

[15] T.Sato, I.Arita, "Tolerating transient faults through an instruction reissue mechanism," Int. Conf. on Parallel and Distributed Computing Systems, 2001.

[16] N.Saxena, et al., "Error detection and handling in a superscalar, speculative out-of-order execution processor system," Fault-Tolerant Computing Symp., 1995.

[17] G.S.Sohi, "Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers," IEEE Transactions on Computers, vol.39, no.3, 1990.

[18] G.S.Sohi, S.E.Breach, T.N.Vijaykumar, "Multiscalar processors," Int. Symp. on Computer Architecture, 1995.

[19] L.Spainhower, T.A.Gregg, "IBM S/390 parallel enterprise server G5 fault tolerance: a historical perspective," IBM Journal of Research and Development, vol.43, no.5/6, 1999.

[20] D.M.Tullsen, S.J.Eggers, H.M.Levy, "Simultaneous multithreading: maximizing on-chip parallelism," Int. Symp. on Computer Architecture, 1995.