Evaluating Low-Cost Fault-Tolerance Mechanism for Microprocessors on Multimedia Applications

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

Abstract

In this paper, we evaluate a low-cost fault-tolerance mechanism for microprocessors, which can detect and recover from transient faults, using multimedia applications. There are two driving force to study fault-tolerance 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. Unfortunately, we found significant performance loss when we evaluated the proposal using SPEC2000 benchmark suit. In this paper, we evaluate it using MediaBench which contains more practical mobile applications than SPEC2000.

1 Introduction

Modern microprocessors have limited hardware error detection, especially for transient faults, which are the vast majority of hardware failures [26]. 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. While the frequency of the transient faults is currently low, 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 [1]. On the other hand, portable and mobile computer devices such as laptop and cell phone are becoming popular and will be one of the major platforms for world-wide distributed computing. For example, Java-2 MicroEdition (J2ME) works on cell phones [10,13]. We can download a game and play it on our cell phone. Travelers guide and flight ticket reservation are available. Furthermore, mobile banking and trading are also provided. For these applications, reliability and dependability are very important. From these considerations, it is expected that even embedded microprocessors should be fault-tolerant.

Current fault-tolerance techniques utilized in commercial systems such as IBM S/390 G5 [26] and Compaq NonStop Himalaya [5] 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 [26]. Hence, low-cost and simple fault-tolerance 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 [22]. Originally, the instruction reissue mechanism is proposed for incorrect data speculation recovery. We modify and apply the mechanism for fault-tolerance. Since some microprocessors include a kind of instruction reissue mechanisms [7,8], this fault-tolerance 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, we found significant performance loss when we evaluated our approach using SPEC2000 benchmark suit. Thus, in this paper we evaluate it using MediaBench suit [9] which is closer to practical mobile applications than SPEC2000.

The rest of this paper is organized as follows. Section 2 surveys related work. Section 3 describes our evaluation methodology. Section 4 explains the proposed fault-tolerance mechanism based on the instruction reissue. Section 5 discusses our simulation results. Finally, Section 6 presents our conclusions.

2 Related Work

Data speculation [6,11] 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 that recovers 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. [11] introduced 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 invalidated in parallel and reissued. However, they only proposed the concept of the instruction reissue. They do not present 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 proposed a practical implementation of the instruction reissue in [20,21].

IBM S/390 G5 [26] is a commercial fault-tolerant microprocessor. However, it relies on traditional fault-tolerance techniques and thus is not adequate for the mobile applications. HaL SPARC64 [23] and Intel Itanium [16] processors have fault detection mechanism for caches and memories but do not for functional units.

DIVA [2] is a fault-tolerant microprocessor based on space redundancy. A simple checker processor attached to its main 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 thus transparent. However, DIVA requires additional ports in register files and caches in order for the checker processor to share processor contexts with the main processor. This increases design complexity and circuit delay of the main processor. In [4], reducing hardware complexity is considered.

Rashid et al. [17] propose a time redundancy technique suitable for Multiscalar architecture [25]. 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 executes 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.

A similar idea of using time redundancy for detecting transient faults is utilized in AR-SMT [19], Slipstrean [15], and SRT [18] processors. They are all based on simultaneous multithreading (SMT) architecture [27]. 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, Slipstream, 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, Slipstream, or SRT is not transparent.

Our approach is most similar to O3RS (Out Of Order Reliable Superscalar) proposed by Mendelson et al. [12]. It also exploits the instruction reissue mechanism for fault detection. However, any impact on processor performance caused by introducing fault-tolerance has not been evaluated. We have evaluated the impact using a timing simulator in [22].

3 Evaluation Environment

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

Table 1. Benchmark programs

program		input set	#inst.s	#cycles	
				4-way	8-way
epicdecode	(ep-d)	test.image.pgm.E	9.6M	4.3M	3.3M
epicencode	(ep-e)	test_image.pgm	53.0M	19.6M	12.5M
g721decode	(g7-d)	clinton.g721	344.3M	120.4M	81.1M
g721encode	(g7-e)	clinton.pcm	362.1M	137.2M	98.8M
ghostscript	(gh)	tiger.ps	1,026.5M	389.7M	257.6M
gsmdecode	(gs-d)	clinton.pcm.run.gsm	77.4M	23.8M	18.9M
gsmencode	(gs-e)	clinton.pcm	255.1M	82.7M	48.6M

programs.

3.1 Processor model

We implemented a timing simulator using SimpleScalar/Alpha tool set (ver.3.0a) [3]. 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) [24], 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 branch tagret buffer, 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.

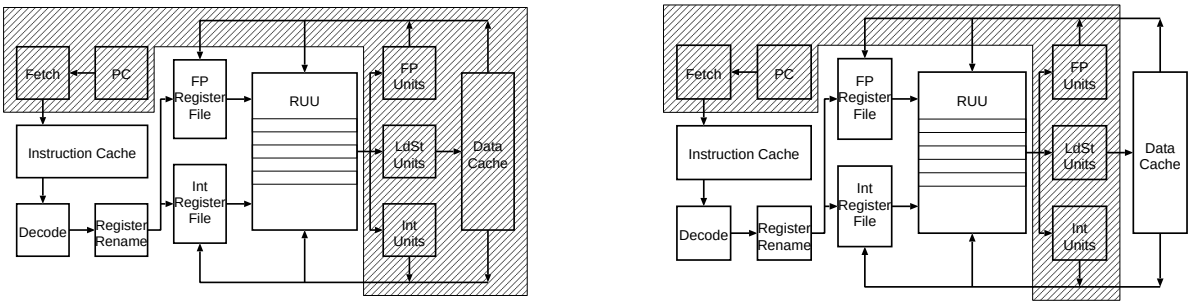
3.2 Benchmark programs

The MediaBench suit [9] is used for this study. The MediaBench is developed for used in the context of embedded, multimedia, and communications applications. It contains image processing, communications, and DSP applications. Table 1 lists the benchmarks and the input sets. The abbreviation for each program is given in the parenthesis. Each program is executed to completion and Table 1 also shows the number of instructions executed and execution cycles. We count only committed instructions. Evaluation using SPEC2000 benchmark suit can be found in [22].

4 Transient Faults Toleration via Instruction Reissue

In order to detect transient faults, we proposed to duplicate committed instructions and compare two results of a single instruction [22]. It is assumed the comparator is fault free. This is possible by using large strong cells [23] 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. Thus, the proposed mechanism detects transient faults occurred in datapaths and their control logics, and then protected portion is depicted as the shaded box in Figure 1(i). We focus on these functions because they are unchecked in modern microprocessors [26]. Instruction cache, register files, and the RUU should be protected using the parity or the ECC, that is common for modern microprocessors. Recovery from the fault will be described later in this section.

Load instructions would diminish performance of the proposed fault-tolerant processor since execution latency of a load instruction is longer than any others. An execution of a load consists of an address calculation and a memory access. This will have pressure on RUU capacity. Therefore, we investigate to eliminate the probable bottlenecks. That is, we remove redundant memory accesses performed by reissued load instructions. The original mechanism depicted in Figure



(i) Redundant memory accesses on load

(ii) No memory access on load reissue

Figure 1. Protected portion

1(i) executes every load instruction twice. Both the address calculation and the memory access are performed twice. Since data cache can be protected by the parity or the 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 protected portion is revised as depicted in Figure 1(ii). While data cache should be protected by the parity or the ECC, that is common for modern microprocessors.

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.

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 [20, 21]. 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, the fault instructions are regarded as misspeculated instructions and re-executed again.

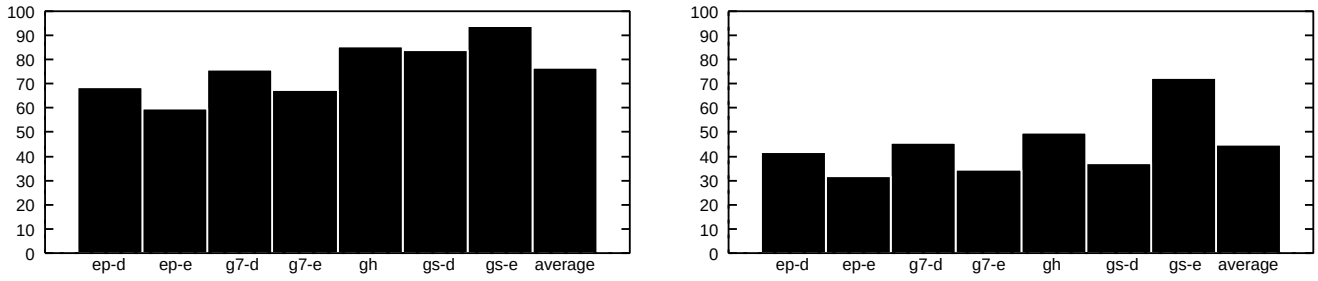
5 Simulation Results

This section presents our simulation results. First, processor performance is evaluated. And then, our approach will be compared with a space redundant technique based on a chip multiprocessor [14]. This paper assumes that we do not detect any transient faults but we only evaluate performance penalty caused by introducing the fault-tolerance mechanism. Therefore, the impact of the fault recovery mechanism on performance is not evaluated. This topic is remained for future study, while it is expected that performance degradation is small since the frequency of transient faults is low.

5.1 Processor performance

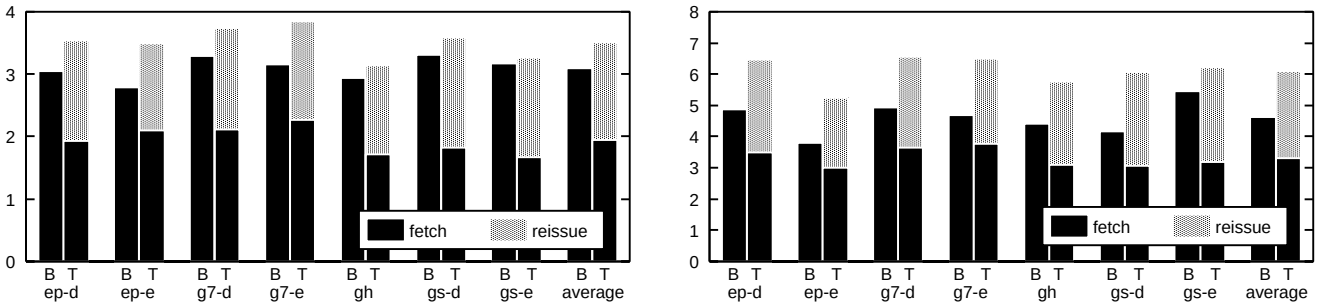
Figures 2(i) and (ii) show performance degradation because of duplicating all committed instructions in the cases of the 4- and 8-way superscalar processors respectively. We use execution cycles for evaluating processor performance and the figures present the percent increase of execution cycles. The primary observation is that performance of the 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. The number of cycles is increased by average of 76.5%. On the other hand, performance degradation of the 8-way superscalar is moderate, and the increase is average of 44.7%.

One of the reasons why the 4-way superscalar suffers severer impact on performance is considerable resource shortage. Figure 3 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-tolerance mechanism. The right (denoted as `T`) is for our approach, 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 the 4-way superscalar, duplicating committed instructions increases



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

Figure 2. %Increase of execution cycles



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

Figure 3. Number of dispatched instructions per cycle

the efficiency of dispatch by only 0.41 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, the 8-way superscalar improves the dispatch efficiency by 1.50 instructions per cycle since it has more functional units for the reissued instructions than the 4-way superscalar. And thus, the 8-way superscalar suffers less performance loss than the 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 reducing the effective capacity of the instruction window, resulting in that the freedom of dynamic instruction scheduling is reduced. That is, processor performance is diminished. Table 2 summarizes clock cycles where each instruction stays in the RUU. The commitment latency is significantly increased, especially for the 4-way superscalar. This increases the execution cycles of the processor models utilizing the proposed fault-tolerance mechanism.

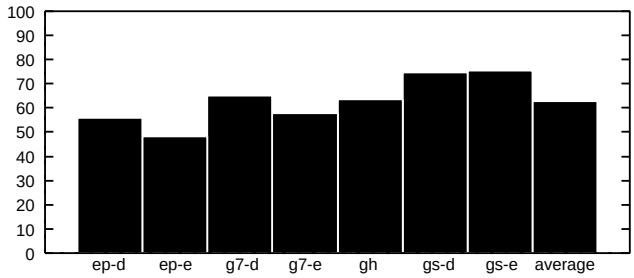
Figure 4 explains how memory accesses affect processor performance. It can be easily observed that the elimination of redundant memory accesses reduces the

Table 2. Commitment latency

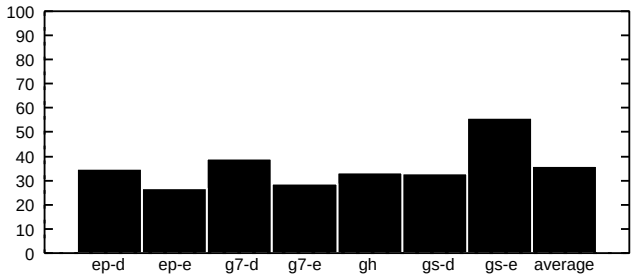
program	4-way latency (cycles)		8-way latency (cycles)	
	base	reissue	base	reissue
epicdecode	7.73	29.84	6.77	16.39
epicencode	12.59	22.85	10.09	16.04
g721decode	10.38	25.28	8.87	14.63
g721encode	8.82	21.18	8.16	12.81
ghostscript	12.93	32.99	8.56	17.42
gsmdecode	17.51	34.53	14.52	20.33
gsmencode	18.06	37.11	10.81	19.36

overhead by 17.8% and 19.8% on average for the 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. The observations can be confirmed by Table 3, which 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.

Figure 5 shows the number of dispatched instructions per cycle. Layout of Figure 5 is the same as for



(i)4-way superscalar



(ii)8-way superscalar

Figure 4. %Influence of removing redundant memory accesses

Table 3. Commitment latency (cycles)

program	4-way		8-way	
	redundant load	no redundancy	redundant load	no redundancy
epicdecode	29.84	27.30	16.39	15.38
epicencode	22.85	20.95	16.04	14.58
g721decode	25.28	23.34	14.63	13.54
g721encode	21.18	19.97	12.81	12.11
ghostscript	32.99	28.07	17.42	15.08
gsmdecode	34.53	32.65	20.33	19.67
gsmencode	37.11	33.31	19.36	17.43

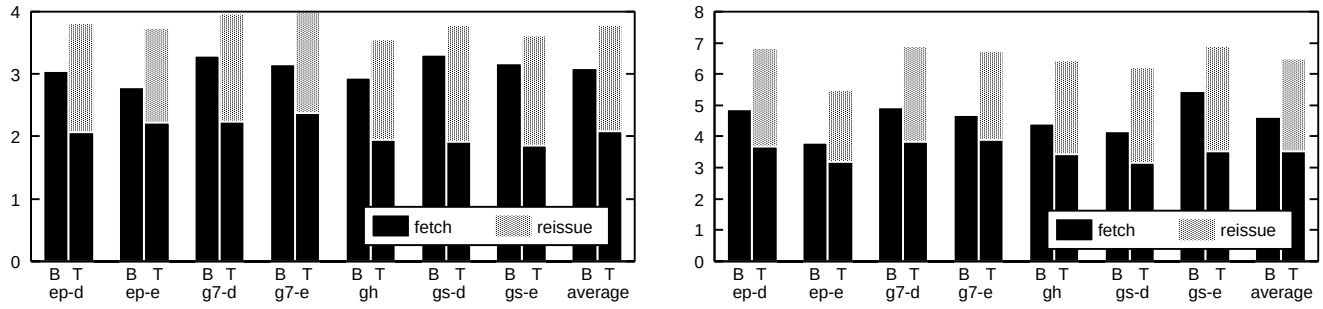
Figure 3. We can see that the dispatch bandwidth becomes more efficiently utilized than the model in Figure 3. In the case of the 4-way superscalar, duplicating committed instructions increases the efficiency of dispatch by 0.68 instructions per cycle. In addition, the dispatch bandwidth is almost fully utilized for most programs. On the other hand, the 8-way superscalar improves the dispatch efficiency by 1.88 instructions per cycle. Therefore, our approach exploits idle functional units efficiently.

5.2 Comparison with chip multiprocessor

Next, we compare our approach with a space redundant technique based on a chip multiprocessor [14]. 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 the 4-way superscalar with the chip multiprocessor consisting of two 2-way superscalars, and the 8-way superscalar with the one consisting of two 4-way superscalars. Hardware resources of the smaller superscalars are half of the baseline model. Thus, total hardware budgets of the time and space redundancy models are approximately same. Please note that only space redundant technique can detect errors in caches, register files, and the RUU. However, they can be protected using the parity or the ECC in the case of the time redundant technique. Figure 6 presents the results. 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 redundancy technique suffers less performance degradation than the space redundancy technique. Thus, our approach is one of the efficient mechanisms for fault-tolerance, while currently we can not conclude if its performance loss is reasonable for mobile platforms.

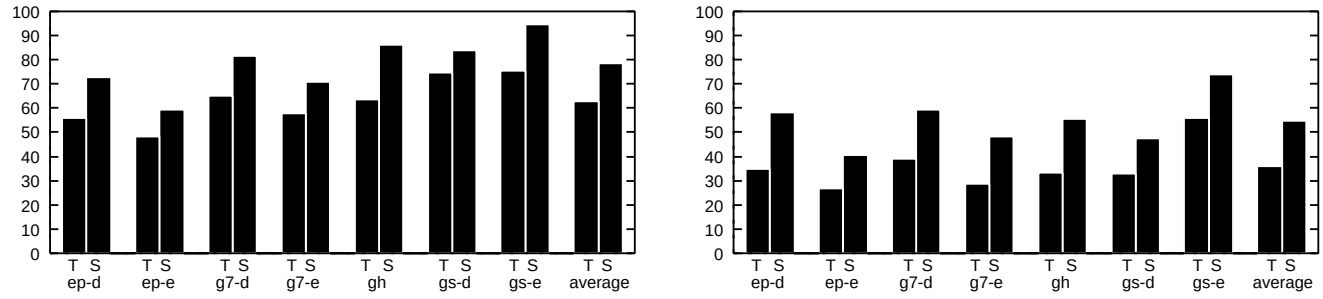
6 Concluding Remarks

There are two driving force to investigate fault-tolerance techniques for microprocessors. One is deep submicron fabrication technologies. Future semicon-



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

Figure 5. Number of dispatched instructions per cycle



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

Figure 6. %Time redundancy versus space redundancy

ductor 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-tolerance mechanism for future microprocessors. It is based on the instruction reissue technique for incorrect data speculation recovery and utilizes time redundancy. In this paper, we have evaluated the proposal using the MediaBench suit, which is one of the practical representatives for modern mobile applications, and found that our approach suffers approximately 35% performance loss from introducing fault-tolerance. While currently we can not conclude if the degradation is reasonable for mobile platforms, it is two times smaller than that in the chip multiprocessor. Therefore, we expect that our proposal would be the first step toward realizing any low-cost fault-tolerance mechanism for future microprocessors.

Acknowledgments

This work is supported in part by the grants from Japan Society for the Promotion of Science (No.13558030) and from the Okawa Foundation for Information and Telecommunications (No.01-13).

References

[1] L.Amghel and M.Nicolaidis, "Cost reduction and evaluation of a temporary faults detecting technique", Design, Automation and Test in Europe Conference, 2000.

[2] T.M.Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," 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] S.Chatterjee, C.Weaver, and T.Austin, "Efficient checker processor design," 33rd International Symposium on Microarchitecture, 2000.

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

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

[7] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the Pentium 4 processor," Intel Technical Journal, issue Q1, 2001.

[8] R.E.Kessler, E.J.McLellan, and D.A.Webb, "The Alpha 21264 microprocessor architecture," International Conference on Computer Design, 1998.

[9] C.Lee, M.Potkonjak, and W.H.Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," 30th International Symposium on Microarchitecture, 1997.

[10] M.Levy, "Java to go: part 1," Microprocessor Report, vol.15, archive 2, 2001.

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

[12] A.Mendelson and N.Suri, "Designing high-performance & reliable superscalar architectures—the out of order reliable superscalar (O3RS) approach," 8th International Conference on Dependable Systems and Networks, 2000.

[13] 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.

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

[15] Z.Purser, K.Sundaramoorthy, and E.Rotenberg, "A study of Slipstream processors," 33rd International Symposium on Microarchitecture, 2000.

[16] N.Quach, "High availability and reliability in the Itanium processor," IEEE Micro, vol.20, no.5, 2000.

[17] F. Rashid, K.K. Saluja, and P. Ramanathan, "Fault tolerance through re-execution in Multiscalar architecture," 8th International Conference on Dependable Systems and Networks, 2000.

[18] S.K.Reinhardt and S.S.Mukherjee, "Transient fault detection via simultaneous multithreading," 27th International Symposium on Computer Architecture, 2000.

[19] E.Rotenberg, "AR-SMT: a microarchitectural approach to fault tolerance in microprocessors," 29th Fault-Tolerant Computing Symposium, 1999.

[20] T.Sato, "Analyzing overhead of reissued instructions on data speculative processors," Workshop on Performance Analysis and its Impact on Design held in conjunction with 25th International Symposium on Computer Architecture, 1998.

[21] T.Sato, "Data dependence speculation using data address prediction and its enhancement with instruction reissue," 24th Euromicro Conference, Workshop on Digital System Design: Architectures, Methods and Tools, 1998.

[22] T.Sato and I.Arita, "In search of efficient reliable processor design," 30th International Conference on Parallel Processing, 2001.

[23] N. Saxena, C. Chen, R. Swami, H. Osone, S. Thusoo, D. Lyon, D. Chang, A. Dharmaraj, N. Patkar, Y. Lu, and B. Chia, "Error detection and handling in a superscalar, speculative out-of-order execution processor system," 25th Fault-Tolerant Computing Symposium, 1995.

[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] G.S.Sohi, S.E.Breach, and T.N.Vijaykumar, "Multiscalar processors," 22nd International Symposium on Computer Architecture, 1995.

[26] L.Spainhower and 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.

[27] D.M.Tullsen, S.J.Eggers, and H.M.Levy, "Simultaneous multithreading: maximizing on-chip parallelism," 22nd International Symposium on Computer Architecture, 1995.