Exploiting Sub-word Parallelism for Dependable Processors

TOSHINORI SATO
PRESTO, JST
Kyushu Institute of Technology
680-4, Kawazu, Iizuka, 820-8502
JAPAN
toshinori.sato@computer.org http://www.mickey.ai.kyutech.ac.jp/~tsato/cosmos/

Abstract: - This paper presents an approach for integrating fault-tolerance techniques into microprocessors by exploiting sub-word parallelism. Smaller transistors, higher clock frequency, and lower power supply voltage reduce reliability of microprocessors. In addition, they are used in systems which require high dependability, such as e-commerce businesses. Based on the trends, it is expected that the quality with respect to reliability will become important for future microprocessors. To meet the demand, we have proposed and evaluated a fault-tolerance mechanism, which is based on instruction reissue and utilizes time redundancy, and found severe performance loss. In order to mitigate the loss, this paper proposes to exploit sub-word parallelism. Redundant instructions are executed in parallel in the form of a SIMD instruction. Detailed simulations show that the performance loss in 4-way and 8-way superscalar processors is reduced to only 25.0% and 15.8%, respectively.

Key-Words: - Microprocessors, fault-tolerance, transient faults, sub-word parallelism, significant compression

1 Introduction

Distributed and web-based computing will be used for financial services and other e-commerce businesses in the near future. For these applications, reliability and dependability are very important [23, 24]. On the other hand, future microprocessors will become susceptible to transient faults [15], which constitute the vast majority of hardware failures [21], as transistors shrink in size with succeeding technology generations. Transient faults are random events which occur when various noise sources cause an incorrect result. For example, cosmic ray particles can alter the state of latches and dynamic logic, resulting in logic errors. In light of these, it is expected that the quality with respect to reliability will become important as well as performance and cost for future microprocessors.

Current fault-tolerance techniques utilized in commercial systems are based on redundancy [1, 21]. For example, error checking is implemented by duplicating processor cores and comparing outputs. Duplicate and compare is adequate for only error detection. The other example is parity and 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 [4, 21]. Hence, a 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 [16, 17, 19]. Originally, the instruction reissue mechanism is proposed for incorrect data speculation recovery [8, 18]. We modified and applied this mechanism for fault-tolerance. Since future microprocessors will utilize data speculation and include the instruction reissue mechanism [5, 7], 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 by means of dynamic instruction scheduling. However, unfortunately, we found significant performance loss [16]. In this paper, we will investigate the use of sub-word parallelism in order to mitigate the performance loss.

2 Related Work

There are a lot of papers on fault tolerant processors. Due to the lack of space, only very related works are mentioned. Detailed survey is found in [19].

Ray et al. [13] duplicate every instruction and detect faults by comparing two instructions which originate from a single instruction. A possible problem in their mechanism is that two executions belong to a single instruction might be effected by the same transient fault and thus it might not be detected. AR-SMT [14] processor, which is based on simultaneous multithreading (SMT) processors [22], utilizes time redundancy for detecting transient faults. 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 a fault by comparing two results. While AR-SMT processor exploits these characteristics, they only detect transient faults but

cannot 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 is not transparent. In contrast, our approach is based on single thread superscalar processors. Our approach is most similar to O3RS [10], which utilizes instruction reissue for fault detection. But it does not use instruction redundancy nor sub-word parallelism.

Instruction reuse [11, 20] is a technique, which eliminates redundant computations by utilizing instruction redundancy. Instruction redundancy 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. Sodani et al. and Molina et al. proposed the reuse buffer and the redundant computation buffer, respectively, for utilizing instruction redundancy and thus for boosting processor performance. In contrast, we exploit instruction redundancy to mitigate the overhead due to including fault tolerance [19]. Parashar et al. [12] proposed a similar technique.

Brooks et al. [2] and Loh et al. [9] found substantial amount of integer operands are less than half word. Thus, two integer operations might be executed in parallel in a single datapath if all operands for the both operations are less than half word. We exploit this characteristic to eliminate redundant execution of a single instruction in order to mitigate performance loss due to including fault tolerance mechanism we proposed. Inoue et al. [6] proposed a similar technique, but in this paper we also propose to exploit instruction redundancy as well as sub-word parallelism and to combine the two techniques.

3 Fault Tolerance Mechanism

This section describes the transient fault tolerance mechanism based on instruction reissue [8, 18]. While the described mechanism is applicable to any dynamic instruction schedulers, we will explain it using our proposed instruction reissue mechanism [18], which is based on register update unit (RUU).

3.1 Fault detection via instruction reissue

The proposed mechanism detects transient faults occurring in arithmetic and logical functions. Thus, the protected portion is depicted as the shaded box in Fig.1. We focus on these functions because they are unchecked in modern microprocessors [21], except for a few products [4]. Instruction cache, register files, and the RUU should be protected using parity or ECC, which is common for modern microprocessors.

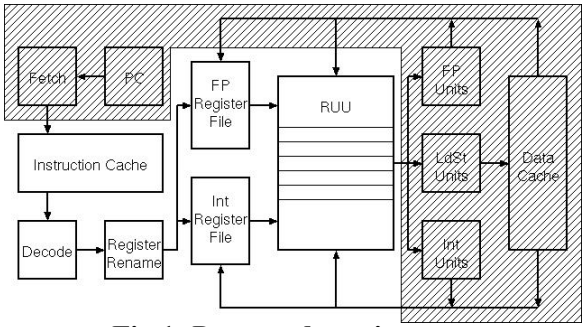


Fig.1: Protected portion

In order to detect transient faults, we propose to duplicate committed instructions and compare two results of a single instruction [16]. It is assumed the comparator is fault-free. This is possible by using large strong cells 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 the second time. If they do not match, a transient fault is detected. Thus, different from the SMT approach, our proposal requires no additional program counters.

There is an advantage of limiting only committed instructions to be reissued. When each instruction is ready for commitment, its all dependences have been resolved. Therefore, it can be unconditionally dispatched if an appropriate functional unit is free. This utilizes the 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 [14]. On the other hand, a possible disadvantage of the policy is the increasing of pressure on the effective instruction window capacity. We evaluated the influence on processor performance in [16] and found significant performance loss.

3.2 Transparent fault recovery

Even when fault detection is successful, a system provided with no scheme for recovery will usually hang, causing application down. Recovery can be handled by either OS or hardware. In this section, we

explain two transparent hardware-based recovery schemes. One uses the existing speculation recovery mechanism for mispredicted branches [13, 16], and the other is based on the instruction reissue mechanism for incorrect data speculation [16]. In this paper, we assume the failure is transient, and thus instruction retry is successful. That is, when a fault 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 when a transient fault is detected. All instructions following the fault instruction are squashed and thus the penalty is very large. However, it is expected that performance degradation will be small since the frequency of faults is low [13].

On the other hand, the fault recovery mechanism utilizing the instruction reissue exploits its ability of selective squashing. As the mechanism reissues only misspeculated instructions when a data dependence misprediction occurs, we reissue only instructions dependent upon a fault instruction. We have already known that the instruction reissue mechanism suffers fewer penalties from mispredicted data speculation than the instruction squashing mechanism, performance degradation will be further reduced.

3.3 Overhead reduction

From our previous studies, we have found that the fault-tolerance technique based on instruction reissue suffers considerable performance loss [16]. Hence, we are investigating ways to mitigate the overhead caused by the introduction of the fault-tolerance mechanism. First, we eliminated redundant cache read [17]. Then, we exploited instruction redundancy [19]. And in this paper, we propose to exploit sub-word parallelism.

3.3.1 Removing redundant memory accesses

Load instructions would diminish the performance of the proposed fault-tolerant processor due to the following reason. An execution of a load consists of address calculation and memory access, and thus the execution latency of a load instruction is longer than any others, while reissued load instructions may always hit in the L1 cache. This will put pressure on the RUU capacity. Based on these observations, we investigate a way to eliminate the possible bottleneck. We remove redundant memory accesses performed by reissued load instructions [17]. The original mechanism executes every load instruction twice. Both address calculation and memory access are performed twice. Since the data cache can be protected by parity or 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. While the data cache should be protected by parity or ECC, it is common for modern microprocessors. One of the possible demerits of the revise is that any faults on the data cache busses cannot be detected. We evaluated the modification in [17] and found there is still considerable performance overhead.

3.3.2 Exploiting instruction redundancy

Instruction reuse [11, 20] is a technique, which eliminates redundant computations by utilizing instruction redundancy. The reuse buffer keeps an outcome of every register-writing and store instruction. In the cases of load and store instructions, address calculation is the object of instruction reuse. When every instruction is executed, its result is registered in the reuse buffer regardless that it is speculative or not. After that, when another instruction is executed, the reuse buffer is referred in parallel. If the previous execution result is obtained from the reuse buffer, the instruction should not be reissued in order to check the occurrence of transient faults but its execution result from a functional unit is compared with the one from the reuse buffer, and its latency is significantly reduced so that the overheads due to fault tolerance is also reduced [19]. An advantage of utilizing the reuse buffer for fault tolerance is that it can be protected by using parity or ECC since the reuse buffer is a memory, and transient faults cannot affect the reuse buffer. While the reuse buffer is protected by parity or ECC, there may be possible transient fault at its logic and busses. Then, the instruction should be executed in the functional unit in parallel with referring the reuse buffer. It should be noted that branch instructions are not covered by the reuse buffer. While every branch should be executed twice, we expect that branch prediction helps to tolerate the latency. This technique requires the additional reuse buffer, while it is found that it efficiently reduces performance overhead [19].

3.3.3 Exploiting sub-word parallelism

Brooks et al. [2] and Loh et al. [9] found substantial amount of integer operands are less than half word. If both operands for an instruction are less than half word, the instruction and its copy can be executed in parallel in a single datapath. This type of execution resembles SIMD multimedia instructions found in modern

instruction set architectures, such as MMX. If the condition is satisfied, the instruction does not have to be executed redundantly via instruction reissue. Fig.2 explains this process. In order to detect narrow operands without severe latency at register operand fetch, an additional bit in every register file entry, which indicates a narrow value, is marked when each value is written in register file.

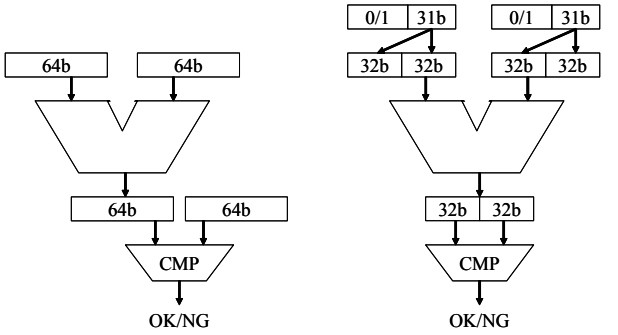


Fig.2: Sub-word parallelism

4 Evaluation Environment

We implemented a timing simulator using SimpleScalar/Alpha tool set (ver.3.0a) [3]. We evaluate 4- and 8-way out-of-order execution superscalar processors. Dynamic instruction scheduling is based on the RUU which has 64 entries. Each functional unit can execute any operation. The latency for execution is 1 cycle except in the cases 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. The data cache 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 that follows a store whose data address is unknown can be executed. 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 target buffer, a 4K-entry gshare-type 2-level adaptive branch predictor, and an 8-entry return address stack are used. The branch predictor is speculatively updated at the instruction decode stage. The number of the reuse buffer entry is 1K instructions.

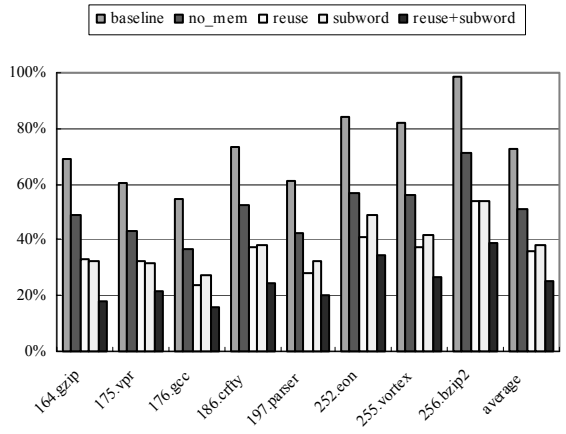
Eight programs from SPEC2000 benchmark suite is used for this study. We use the object files provided by the University of Michigan. They were compiled by DEC C V5.9-008 on Digital UNIX V4.0 (Rev.1229). For each program except for **252.eon**, 1 billion instructions are skipped before actual simulation begins. Each program is executed to completion or for 100 million instructions. We count only committed instructions.

5 Simulation Results

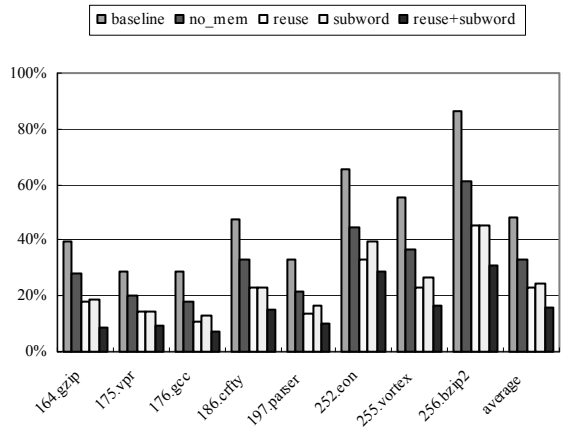
This section presents our simulation results. We use execution cycles for evaluating processor performance and the figures present the percent increase of the cycles. In this paper, we do not assume any transient faults occur but we only evaluate performance penalty caused by introducing the proposed mechanism. Therefore, the impact of the recovery from faults on performance is not evaluated.

For each group of five bars for every program, the first bar (see from left to right) labeled with **baseline** in Fig.3 shows the performance degradation due to the duplication of all committed instructions in the cases of 4- and 8-way superscalar processors. As can be easily seen, performance of the 4-way superscalar is diminished significantly, while it is smaller than the case in which the program is executed twice and the results are compared. It is reduced by an average of 64.5%. On the other hand, the performance degradation of the 8-way superscalar is moderate: an average of 47.9%. The major reason is the increasing latency of instructions until commitment [17]. Each reissued instruction stays in instruction window longer than in the case of the processor without fault tolerance. The goal of this paper is to mitigate the performance loss.

Fig.3 also explains how memory accesses affect performance. The second bar labeled with **no_mem** indicates the increase in execution cycles over the processor without fault tolerance when redundant memory accesses are eliminated. It is 43.6% and 33.0% 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.



(a) 4-way superscalar



(b) 8-way superscalar

Fig.3: %Increase in execution cycles

The middle bar labeled with **reuse** shows how efficiently instruction redundancy is utilized for overhead reduction. As can be easily seen, the increase in the 4- and 8-way superscalars becomes significantly small and is an average of 35.7% and 20.8%, respectively. This technique requires the additional 1K-entry reuse buffer as mentioned above, while it efficiently reduces the performance loss.

The next bar labeled with **subword** presents how performance loss is recovered by exploiting sub-word parallelism. The reuse buffer is not used in this evaluation. For most programs, it eliminates the overhead as efficiently as the instruction reuse does. The average loss for 4- and 8-way superscalars is reduced to 38.3% and 21.6%, respectively. It should be noted that this technique does not require large hardware such as the reuse buffer. Fig.4 explains why the loss is efficiently eliminated by exploiting sub-word parallelism. It shows the percentage of instructions which are candidate for SIMD operation (labeled with **subword_op**) and the percentage of instructions that SIMD operation is successfully applied (labeled with **subword_hit**). Because results are almost same for both processor models, only results for 8-way superscalar are presented. As you can see, approximately 25% of total instructions can be executed as SIMD operations. In other words, one forth of total instructions does not require redundant execution via instruction reissue for fault tolerance.

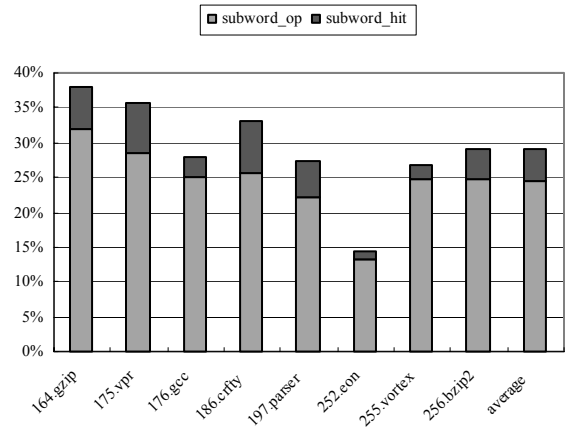


Fig.4: %Aggregated sub-word parallelism

The last bar labeled with **reuse+subword** in Fig.3 shows the synergy effect of utilizing instruction redundancy and exploiting sub-word parallelism. The combination reduces performance loss to 25.0% and 15.8% in the cases of 4- and 8-way superscalars, respectively. There is a possibility of the reuse buffer size reduction. Fig.5 shows how the reduction in the reuse buffer entry affects performance in the case of 8-way superscalar. For 4-way superscalar, we found the same tendency. For each group of six bars, first five bars from left to right indicate performance loss for processors which exploits sub-word parallelism with 4-, 16-, 64-, 256-, and 1K-entry reuse buffers, respectively. The last one is for the processor which does not exploit sub-word parallelism but only utilizes 1K-entry reuse buffer. As you can see, exploiting sub-word parallelism reduces required reuse buffer capacity. 4-entry reuse buffer with sub-word parallelism has the equivalent contribution with 1K-entry one without sub-word parallelism.

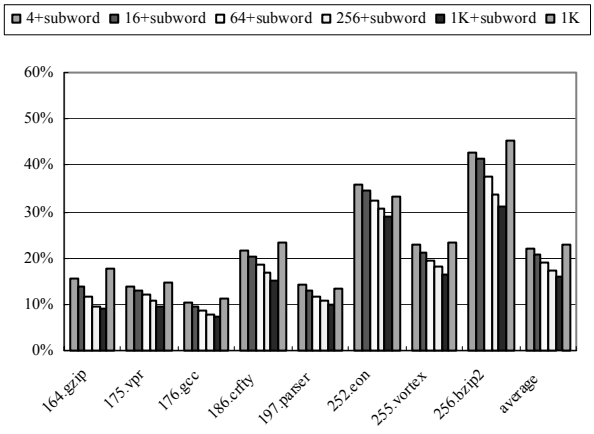


Fig.5: %Effect of reuse buffer capacity

6 Concluding Remarks

The investigation of fault-tolerance techniques for microprocessors is driven by two issues: One regards deep submicron fabrication technologies. Future semiconductor technologies could become more susceptible to soft errors. The other is the increasing popularity of mobile platforms. Cellular telephones are currently used for applications which are critical to our financial security. Such applications demand that computer systems work correctly. In light of this, we proposed the fault-tolerance mechanism for future microprocessors. It is based on the instruction reissue technique and utilizes time redundancy. Our previous evaluation showed that our proposed mechanism caused considerable performance loss. In this paper, we proposed to exploit sub-word parallelism to mitigate the loss, and to combine it with the reuse buffer. We evaluated our proposal using the timing simulator and found that the performance loss can be significantly reduced.

References:

[1] D. C. Bossen, A. Kitamorn, K. F. Reick, and M. S. Floyd, "Fault-tolerant design of the IBM pSeries 690 system using POWER4 processor technology," IBM Jour. Res. & and Develop., 46(1), 2002.

[2] D. Brooks and M. Martonosi, "Dynamically exploiting narrow width operands to improve processor power and performance," 5th Int. Symp. on High Performance Computer Architecture, 1999.

[3] D. Burger et al., "The SimpleScalar tool set, version 2.0," ACM SIGARCH Comp. Arch. News, 25(3), 1997.

[4] Fujitsu Ltd., "SPARC64 V microprocessor provides foundation for PRIMPOWER performance and reliability leadership," White paper, 2002.

[5] G. Hinton, et al., "The microarchitecture of the Pentium 4 processor," Intel Technical Journal, issue Q1, 2001.

[6] K. Inoue, et al., "A Dependable processor architecture exploiting spatial redundancy realized by datapath partitioning," Forum on Information Technology, 2004.

[7] R. E. Kessler, et al., "The Alpha 21264 microprocessor architecture," Int. Conf. on Computer Design, 1998.

[8] I. Kim and M. H. Lipasti, "Understanding scheduling replay schemes," 10th Int. Symp. on High Performance Computer Architecture, 2004.

[9] G. H. Lo, "Exploiting data-width locality to increase superscalar execution bandwidth," 35th Int. Symp. on Microarchitecture, 2002.

[10] A. Mendelson and N. Suri, "Designing high-performance & reliable superscalar architectures - the out of order reliable superscalar (O3RS) approach," 1st Int. Conf. on Dependable Systems and Networks, 2000.

[11] C. Molina and A. Gonzalez, "Dynamic removal of redundant computations," 13th Int. Conf. on Supercomputing, 1999.

[12] A. Parashar, et al., "A complexity-effective approach to ALU bandwidth enhancement for instruction-level temporal redundancy," 31st Int. Symp. on Computer Architecture, 2004.

[13] J. Ray, J. C. Hoe, and B. Falsafi, "Dual use of superscalar datapath for transient-fault detection and recovery," 34th Int. Symp. on Microarchitecture, 2001.

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

[15] P. I. Rubinfeld, "Managing problems at high speed," IEEE Computer, 31(1), 1998.

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

[17] T. Sato et al., "In search of efficient reliable processor design," 30th Int. Conf. on Parallel Processing, 2001.

[18] T. Sato, "Evaluating the impact of reissued instructions on data speculative processor performance," Microprocessors and Microsystems, 25(9), 2002.

[19] T. Sato, "Exploiting instruction redundancy for transient fault tolerance," 18th Int. Symp. on Defect and Fault Tolerance in VLSI Systems, 2003.

[20] A. Sodani et al., "Dynamic instruction reuse," 24th Int. Symp. on Computer Architecture, 1997.

[21] L. Spainhower et al., "IBM S/390 parallel enterprise server G5 fault tolerance: a historical perspective," IBM Jour. Res. & Develop., 43(5/6), 1999.

[22] D. M. Tullsen, et al., "Simultaneous multithreading: maximizing on-chip parallelism," 22nd Int. Symp. on Computer Architecture, 1995.

[23] R. Chachra et al., "Software reliability assessment of high volume web systems," 4th WSEAS International Conference on Information Science, Communications and Applications, 2004.

[24] E.-N. Ko, "Performance analysis of error detection system for distributed multimedia environment," 4th WSEAS International Conference on Information Science, Communications and Applications, 2004.