Easing Instruction Queue Competition among Threads in RMT

Jie Yin, Jianhui Jiang
Department of Computer Science and Technology, Tongji University, Shanghai, China
E-mail: jieyin2008@gmail.com, jhjiang@tongji.edu.cn

***Abstract*—As chip feature size decreases, processors are getting more and more sensitive to soft errors. To find cheaper reliability solutions has attracted the attention of many researches. SMT (Simultaneous Multithreading) processor permits multiple issues from different threads at the same time, which provides nature support for fault-tolerance by executing threads redundantly. Many RMT (Redundant Multithreading) architectures have been proposed. In those architectures, IQ (Instruction Queue) is a critical resource that affects the performance obviously. This paper proposed DDDI (Delay Dispatching Dependent Instructions) strategy which can use IQ more efficiently. In DDDI, instructions that dependent on load instructions that encounter cache miss can't be dispatched in to IQ until the load instructions get values from L2 cache or main memory. Experiments show that DDDI can avoid the threads that encounter cache miss blocking IQ resources, and not only IQ, but also the whole pipeline can be used more efficiently. Performance is boosted outstandingly.**

Index Terms—**Instruction Queue, SRT, cache miss, RMT, rename register file**

I. INTRODUCTION

For many years, most computer architecture researchers have pursed one primary goal: performance. Recently, however, as chip feature size decreases, the number of transistors integrated on the chip increased dramatically, and chip supply voltage is getting lower and lower. On the one hand it improves the processor performance; on the other hand it makes processors increasingly sensitive to soft errors [1].

Radiation-induced errors are termed “soft error” since the state of one or more bits in silicon chip could flip temporarily without damaging the hardware. Soft errors arise from energetic particles, such as alpha particles from packaging material [2] and neutrons from atmosphere [3].

Some studies show that under nano-technology, soft error has become the main cause of chip failure [4]. Almost 80-90% of system failures are caused by soft error [5] [6].

At present, only machines that run critical mission consider the prevention of soft errors. However, according to Intel’s prediction, in the next few years, even the most low-end processors also need to be strengthened in this regard.

Architecture design for soft error is an efficient scheme to reduce SER (Soft Error Rate). Computer architecture has long coped with various types of faults, including soft errors. The most commonly used architecture-based approaches are spatial redundancy, information redundancy and time redundancy.

Spatial redundancy duplicates a module and makes the two replicas compare their results. Due to high hardware cost and power and energy consumption, this approach is mainly used in high-end processors, such as IBM S/390 [7], and Compaq Nonstop Himalaya [8].

The basic principle of information redundancy is to add redundant bits to a datum to detect or tolerant an error [9]. The method is used primarily for storage components (such as Cache, Memory, and Register, etc.), and some transmission components (e.g. Bus), which is not practical for logical components.

Temporary redundancy performs an operation twice (or more times), one after another, and then compare the results. It can cause some damage to the performance. Hardly any processor can run at full speed, and the re-execution can use spare resource, which will mitigate the performance damage. Because of the low hardware cost (relative to spatial redundancy) and high fault coverage (relative to information redundancy), temporary redundancy is a cost-effective solution and has attracted great attention of computer architects.

In last century, temporary redundancy is implemented primarily through instruction re-execution in super-scalar architecture [10]. Each instruction is issued twice, and results between them are compared. This scheme only detect faults in execute stage, with other pipeline stage unprotected.

Later, SMT (Simultaneous Multithreading) architecture was proposed in 1995 by Tullsen [11]. SMT is capable of executing multiple instructions from different threads simultaneously [12] [13]. Through the excavation of the instruction-level parallelism within a single-threaded and Inter-thread-level parallelism, SMT reduced the horizon waste and vertical waste. SMT improves the resource utilization through sharing them among threads. SMT allows multiple threads run concurrently, which provides nature supports for fault-tolerance by executing threads redundantly. Fault-tolerance through redundant execution based on SMT is called RMT (Redundant Multi-threading).

AR-SMT [14] is the firstly proposed SMT-based fault-tolerance architecture, which copies the thread into two copies (master and slave threads). Results from master threads are stored in a buffer to wait for the corresponding results from slave threads for comparison.

doi:10.4304/jcp.6.7.1394-1401

However, AR-SMT needs to allocate additional memory space for the slave threads and isn't transparent to the operating system.

SRT [15] proposed by Mukherjee is a simple and practical fault-detection architecture, which has become a lot of follow-up research infrastructure [16]. SRT detects faults by comparing the corresponding store results between master threads and slave threads.

Later in SRTR [16], function of restoration is added in SRT. In SRT, only store instructions are compared, while other instructions can be committed before comparing, when architecture register file polluted by some faulty instruction results, the processor can't restore to a correct state. While in SRTR, each instruction must be compared before committing, and no faults can spread to architecture register file, when meeting faults, only need to re-fetch the faulty instructions from I-Cache and re-execute them.

In our previous research, IQ (Instruction Queue) is proved to be a critical resource that affects the performance obviously. In RMT, Slave threads can use IQ more efficiently than master threads. That's because master threads may encounter D-cache miss, while slave threads may not. Slave threads can get the Load results from master threads. In master threads, the instructions dependent on Data-cache-miss-load instructions may stay in IQ for a period of time. In the paper, the instructions of master threads that dependent on unresolved Load that encountered cache miss can't be dispatched into IQ until the corresponding load instructions get results from L2 Cache or main memory. The method is named DDDI (Delay Dispatching Dependent Instructions). Experiments show that the method is effective for improving the IQ utilization and processor performance

The rest of the paper is organized as follows. Section 2 discusses the background and provides the motivation for our techniques. Section 3 discusses the DDDI technique. Section 4 and 5 presents the experimental results and analysis. Section 5 introduces some related works. Finally in section 6, we conclude.

II. BACKGROUND

1) Improve the performance of RMT

Nowadays, to improve the performance of RMT is a hot topic. The primary goal of RMT optimizations is to reduce the performance degradation of a single program caused by the redundant thread.

There exist two methods to improve the performance of RMT. First, reduce the number of instructions need to be executed. Second, speed up the flow rate of instructions in the pipeline, which is realized mainly through preventing some threads blocking key resources.

Reducing the number of instructions that needs executing has been researched a lot in superscalar architecture; many methods have been extended to RMT. There have been a number of proposals that exploit this:

a) Instruction Reuse:

Sodani and Sohi proposed instructions reuse to accelerate the execution of a single program [17]. Sodani and Sohi created an instruction reuse buffer that tracks one or more instruction's input and output values. If an instruction or a sequence of instructions is executed again and can be matched against the instructions present in the reuse buffer, then the pipeline can simply obtain the output of the instructions without executing them [17].

In an RMT implementation, the contents of the reuse buffer can be updated after the output comparator certifies that an instruction is fault free. In subsequent executions, the trailing thread can use values being passed to it by the leading thread (e.g., via an RVQ) to probe the reuse buffer, obtain the result values in case of a hit, and thereby avoid executing the instruction itself. Parashar et al. [18] and Gomaa and Vijaykumar [19] have explored variants of this scheme.

b) Avoid execution of dynamically dead instructions

In the course of program execution, many dead instructions are operated. The instructions whose results will never be used are termed "dead instructions"[20], such as:

```
R1=R2+R3    (1)
R4=R2*R3    (2)
R2=R4-R3    (3)
R1=R4+R2    (4)
R5=R6+1     (5)
```

In the instruction sequence, since the result of the first instruction will never be used by other instructions, so they are dead instructions and faults in the instruction have no effect on the thread correctness.

In RMT, dead instructions are detected during the execution of master threads, and execution of these instructions may be omitted in slave threads.

c) Turn slave threads off in high IPC regions.

There exist some instruction regions that need huge executing resources and the reliability is not important to the user. Such as some streaming media, we only need to guarantee its performance, slight errors are permissible. Therefore, these regions can be ignored in slave threads.

Another method to improve the performance of RMT is speed up the flow rat of instructions in the pipeline. We must avoid some instructions blocking key resources. For example, some instructions may stay in the key resources for a long time without any progress, while the resources may be used by other threads.

In SRTR, in order to avoid faults spread to architecture registers, each instruction must be compared before committing. The instructions of master threads often need to wait for the corresponding instructions of slave threads arriving the comparing stage. The waiting of master threads will delay the release of resources that occupied by the master threads. And in SRT, only store instructions need comparison, which is why it can gain better performance than SRTR.

The ability of every resource that affects the performance is analyzed quantitatively in our previous research. Rename Register File (RRF) and Instruction Queue (IQ) are two most important resources.

Further studies revealed that, load instructions that encounter cache miss may block the two key resources easily. Meeting cache miss, load instructions and the follow-up instructions dependent on them will stay in the

key resources for a period of long time without any progress. Reference [21] found that when running some workloads, the IQ is on average 97 occupied when at least on L2 miss is outstanding, but only 62% occupied at other times.

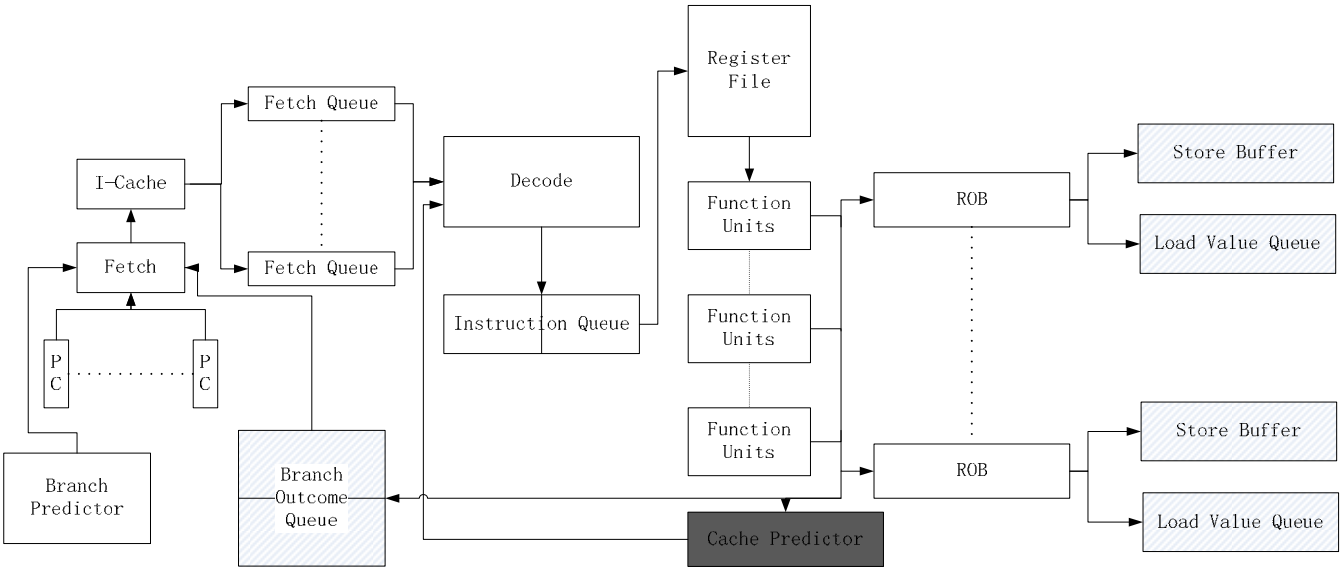

Figure 1. Structure of SRT with DDDI

2) SRT (Simultaneous Redundant Threading)

The purpose of DDDI is to prevent above situation from happening. It can be used in any RMT architecture. We selected the simple and practical SRT as infrastructure. This section mainly introduces SRT:

SRT is simple and practical fault-detection architecture, which has become a lot of follow-up research infrastructure. SRT detects faults by comparing the corresponding store results between master threads and slave threads.

In [15], to aid the analysis, the authors firstly proposed the concept of sphere of replication. All activity and state within the sphere is replicated, either in time or in space. Values that cross the boundary of the sphere of replication are the outputs and inputs that require comparison and replication, respectively.

In order to detect faults, store instruction results produced by master threads must be stored into STB (Store Buffer) and wait for corresponding results from slave threads for comparison. Fault may propagate to the follow-up store instructions and that can't propagate out have no effect on the program results. So checking the store instruction results is enough. Only instructions of master threads should be committed, while those of slave threads just for comparison.

For load instructions, the authors propose LVQ (Load Value Queue) to ensure that master threads and slave threads can load the same value. Load instructions results of master threads are stored into LVQ, and are used by slave threads later. So, slave threads mayn't encounter D-cache miss.

The authors also proposed BOQ (Branch of Queue) and SF (Slack Fetch) to enhance the performance. Through BOQ, master threads will store the branch instruction results to BOQ, which may be used by the corresponding slave threads later. Slack Fetch refers to keeping a certain distance between the two threads so that slave threads can get Load instruction results produced by master threads. So slave threads may not encounter miss-predicted branches and D-cache-miss-load.

As the original SRT only supports single thread redundantly execution, the paper firstly extend the structure to support the execution of multiple independent threads. Figure 1 is the SRT implemented in the paper:

The architecture can support eight independent threads simultaneously. In fault-tolerance mode, four different threads are performed simultaneously and each of them is copied to two. The shaded sections are added relative to original SMT.

Like previous studies, this paper mainly studies the performance in fault-free case. The single-fault detection rate was 100%.

III. MAIN IDEA

In SMT, D-cache-miss degrades the performance significantly. When load instruction encounters cache miss, it may stay in the pipeline for almost hundreds of clock cycles (when encountering L2 cache miss), and the instructions dependent on the instructions will be clogged in IQ, too, while the IQ slots may be used by other threads. The main idea of this paper is to delay the time the instructions which are dependent on unresolved D-cache-miss load instructions entering IQ.

Figure 1 is the structure of SRT with DDDI. When load instruction encounters cache miss, set the new added bit of the destination rename register to 1 (see Figure 2). When load instructions is completed or squashed from the pipeline due to wrong branch prediction, the corresponding bits of the destination rename register is set to 0.

During dispatch stage, if we find the bit of any source operand register of the instruction is 1, which indicates that it is dependent on a load instruction that encounter cache miss, and the load instructions has not been resolved yet. So, stop dispatching instructions from the thread and select the next thread.

Some scholars have suggested that the threads should be stopped when cache miss detected. We believe that this is a rather extreme approach. Even load instructions encounter cache miss, there may exist no dependency relationship between them and the follow-up instructions. We have verified the method before and found that the performance actually declined. Because it may degrades the instruction parallelism.

Detecting whether load instructions will encounter cache miss needs a period of time. However, the cache access prediction mechanism has become a common structure yet. Like branch prediction, it can predict whether the cache access will hit or miss. The structure is implemented in many general processors, such as the Alpha 21464. This paper also used the prediction mechanism. When Load instruction leaves IQ for execution, predicting whether it will encounter cache miss, and the corresponding bit can be set timely.

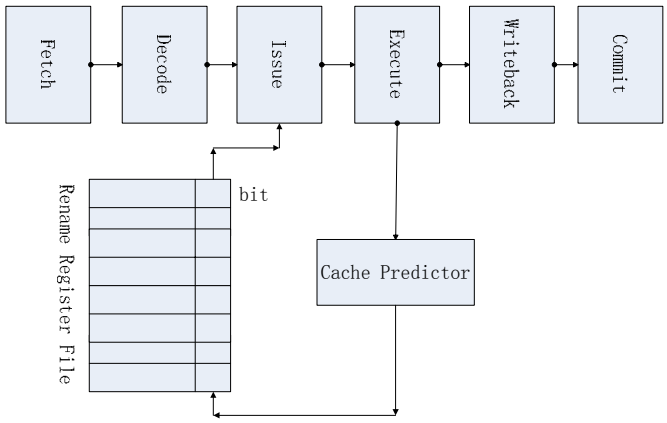

Figure 2. Pipeline of SRT with DDDI

IV. EXPERIMENT SETUP

The original SRT only supported single-thread case. The author has not studied the redundant multithreading performance thoroughly, because there isn't multiple threads compete for resources in single-thread case. This paper firstly extended the architecture to support multiple threads.

In original SRT, Slack size is set to 256, and BOQ, LVQ and STB are set to 64. We modified the configurations, because the original configurations may cause deadlock in multithreading case. When some master threads which has occupied certain entire key resources may encounter BOQ full or LVQ full, and the threads will be halted, while corresponding slave threads may be also halted because they can't obtain the key resources. And then, no BOQ or LVQ entry will be freed by slave threads. Finally, all threads cannot make forward. So in our experiments, the Slack is set to 128, followed by three parameters set to 160. The structures are all FIFO, which cause low hardware overhead.

In this paper, we use simulator M-SIM [19], which is expansion of simple scalar which is widely used in computer architecture research. The difference is that M-SIM uses ROB instead of RUU. The simulator supports simultaneous multi-threading. M-SIM simulates private fetch queue, ROB, LSQ, and architecture registers for each thread. All threads share IQ, and rename registers, while other simulators simulate shared fetch queue, ROB, LSQ and so on, in order to further share these resources. M-SIM is closer to the real situations [9] [10], in practical case, these structures are often private for each thread to reduce the hardware complexity. Because sharing these components, would make the hardware complexity greatly improved, which in turn increase the processor frequency constraints. Table 1 shows the detailed simulator configuration:

TABLE 1. Processor parameters used in simulator

Parameter	Value
Machine Width	8-wide fetch, 8-wide issue, 8-wide commit
Rename Register	64
Instruction Queue Size	32
ROB Size	96
LSQ Size	48
Function Units and Latency	8 Int ADD(1), 2Int Mult(3)/Div(20), 4Load/Store, 8 FP ADD(2), 2 FP Mult(4)/Div(12)
L1 I-Cache	32 KB, 2-way set-associative, 512 sets, 32 bsize, 1 cycles hit time
L1 D-Cache	32 KB, 4-way set-associative, 256 sets, 32 bsize, 1 cycles hit time
L2 Cache Unified	4 MB, 8-way set-associative, 1024 sets, 64 bsize, 6 cycles hit time
I_tlb	256 KB, 4-way set-associative, 16 sets, 4096 bsize
D_tlb	512 KB, 4-way set-associative, 32 sets, 4096 bsize
Tlb_miss_lat	30
Mem_bus_width	8
Branch Predictor	Bimod bimod_size 2048, btb_sets 512, btb_assoc 4, retstack_size 8

Experiments selected some SPEC2000 programs, which is pre-compiled by the M-SIM team. To ignore the impact of initialization, each thread have neglected the front part of the programs according to paper [23] [24], and then simulate the next 100 million instructions, in a multi-threaded environment, when a thread has committed 100million instructions, the simulation is ended.

According to [23], we first categorized the SPEC benchmarks into either high-ILP or memory-intensive programs, labeled "ILP" and "MEM," respectively. In order to make the study representative, we considered a variety of load combinations.

TABLE 2. Multi-programmed workloads used in the experiments.

name	Loads
ILP (4 ILP 0 MEM)	Gzip/ Bzip/ Gcc-i/ Fma3d
MIX1 (3 ILP 1 MEM)	Gcc-i/ Bzip/ Vpr/ Gzip
MIX2 (2 ILP 2 MEM)	Gzip/ Gcc-i/ Swim/ Applu
MIX3 (1 ILP 3 MEM)	Gzip/ Equake/ Swim/ Applu
MEM (0 ILP 4 MEM)	Art /Swim/Applu/Equake

V. RESULT ANALYSIS:

1) The impact of cache-miss-load instruction on performance

Firstly, the impact of cache-miss-load instructions on IQ is researched. Figure 3 denotes the average residence time stayed in IQ of each instruction from ILP workload. Dark gray column represents the master threads, and light gray column represents the slave threads.

Figure 3 reveals that instructions of each slave thread stayed in IQ about 4 cycles, while instructions of master threads may stay in IQ much longer.

The difference of instructions between master threads and slave threads is that, master threads may encounter load-cache-miss, and the thread will be blocked in IQ for many cycles, even hundreds of cycles (when meeting L2 Cache miss)，while, slave threads may get load value from LVQ. So, instructions of slave threads can pass through IQ quickly.

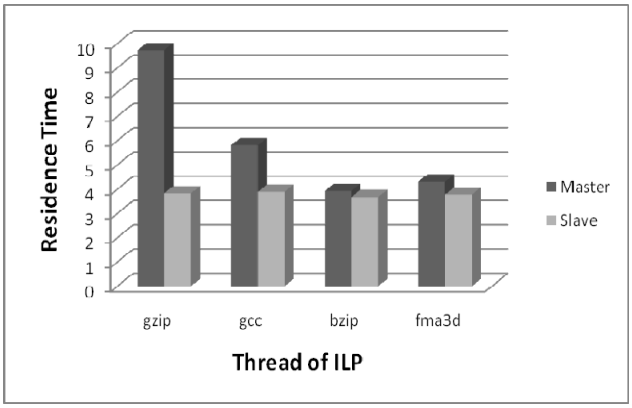

Figure 3 The residence time of instructions in IQ for each thread

In RMT, there exists vicious competition among threads, that is some thread may occupy almost the whole key resources and cause the other threads can't obtain appropriate key resources at some time. IQ often encounters such situation. When some master threads meet cache miss, above situation will probably happen. Table 3 shows the probability that one thread occupied the whole IQ. Form the table, it can be found that about 2.09% the whole running period IQ is occupied by the master thread of gzip. At that time, no other thread can make progress, due to the lack of IQ resources.

TABLE 3. The probability that one thread occupied the whole IQ

thread	Master	Slave
gzip	0.0209	0.0001
gcc	0.0044	0.0001
bzip	0.0001	0.0001
ma3d	0.0001	0

The previous paragraph studied the impact of cache-miss-load instructions among 8 threads of workload ILP. When cache miss taking place, flow rate of instructions in pipeline will be slowed down. Table 4 represents the relationship between cache-miss-load instructions and the flow speed in the pipeline of different workload. In table 4, L1 Miss refers to the proportion of the L1-Cache-Miss load instructions in all instructions. Residence time in pipeline indicates the average time stayed in pipeline of one instruction from different workload.

From table 4, it can be concluded that when meeting cache miss more frequently, instructions will stay in the pipeline longer.

TABLE 4. Residence time in pipeline Vs. Cache miss

Workload	L1 Miss	L2 Miss	Residence time in pipeline
ILP	0.008749	0.00447	30.187
MIX1	0.011572	0.007463	42.7699
MIX2	0.023611	0.015703	67.4112
MIX3	0.013946	0.008055	48.349
MEM	0.021938	0.016602	56.1714

2) Performance improvement：

Since NDDI avoids instructions dependent on D-cache-miss-load staying in IQ too long without any performance contributions, IQ is used more efficiently to gain great performance improvement. Figure 4 shows that the method is effective in increasing processor performance.

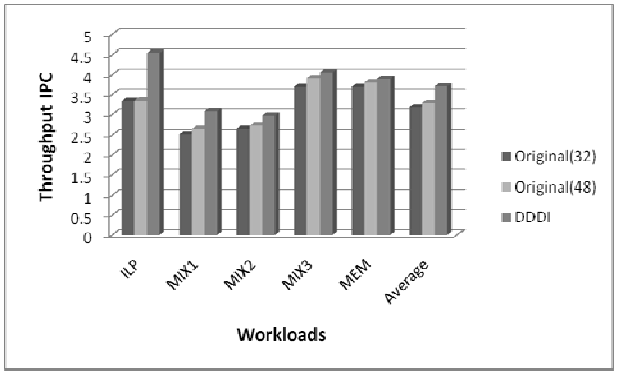

Figure 4 Throughput Improvement

In figure 4, original (32) represents the original SRT which IQ size is 32, and IQ size is 48 for original (48). From figure 4, it can be concluded that DDDI is more efficient than increasing the size of IQ.

Throughput just indicates the whole performance of all programs. In order to measure the improvements of the single thread, this paper also evaluates the weighted speedup of DDDI, namely, weighted speedup [18], which is defined as follows:

$$weightedspeedup = \frac{1}{N}\sum_{i=1}^{N}\frac{IPCi'}{IPCi}$$

Which IPCi and IPCi ' indicate the original IPC and IPC using DDDI for each thread respectively. Figure 5 indicates that the weighted speedup for the proposed strategy has been improved, too.

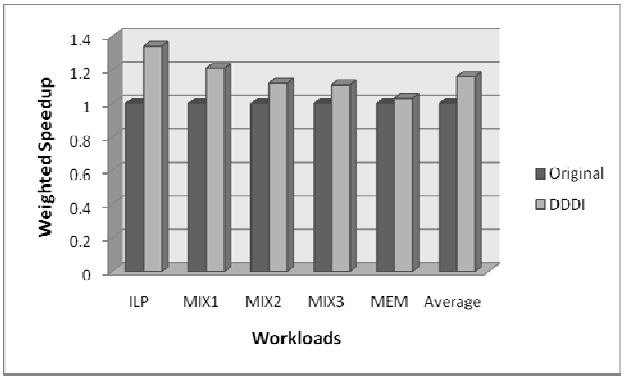

Figure 5 Weighted Speedup Improvement

3) The residence time of instructions in IQ:

In original SRT, when load instruction encounters cache miss time, the follow-up dependent on them will stay in IQ for a long time. When using DDDI, the instructions can enter IQ only if the load instructions obtain data from storage. Therefore, this method can reduce average residence time in IQ, which can speed up the flow speed of instructions to improve processor performance. Figure6 shows changes of residence time for instructions in IQ:

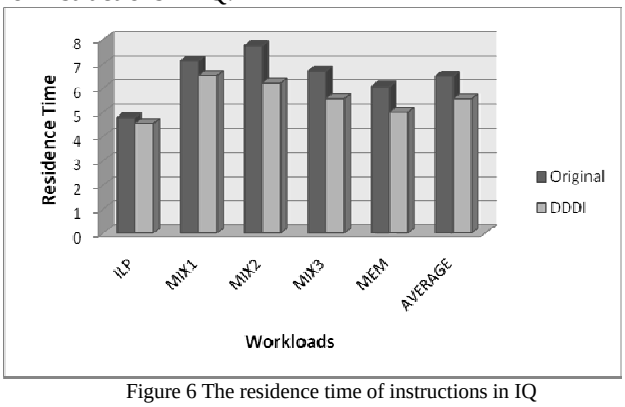

Figure 6 The residence time of instructions in IQ

Experimental results show that using this method, the residence time of instructions (including the mis-predicted instruction) in the IQ significant decreased. The average residence time is dropped by 14.23%.

In our research, we studied the distribution of instruction queue utilization in detail (see Figure 7 and 8). In original SRT, during 28.13% of the running period, IQ is full used which means no IQ entry is free. While in SRT with DDDI, the proportion is 36.69%. However, in original SRT, 9.17% is occupied by one thread, which means that no other thread can obtain the IQ resources. And in SRT with DDDI, the proportion is dropped to 4.73%.

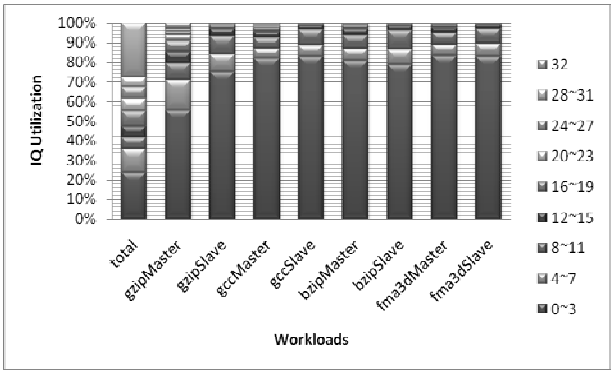

Figure 7 Distribution of Instruction Queue Utilization of Original SRT

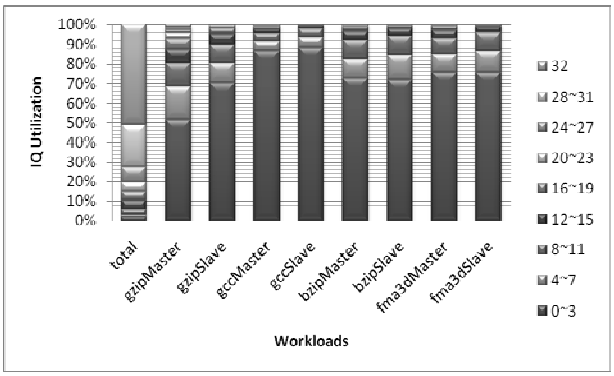

Figure 8 Distribution of Instruction Queue Utilization of SRT with DDDI

4) The residence time of instructions in pipeline

DDDI not only speedup the flow rate of instructions in IQ, but also another key resource: rename register file and the whole pipeline. Because in original SRT, when master threads encountering cache miss, they will block the IQ resource, and the flow rate of the whole pipeline will be slowed down.

Figure 9 illustrates the average residence time of instructions stay in rename register file. It can be found that when using DDDI, instructions will stay in rename register file more short-term.

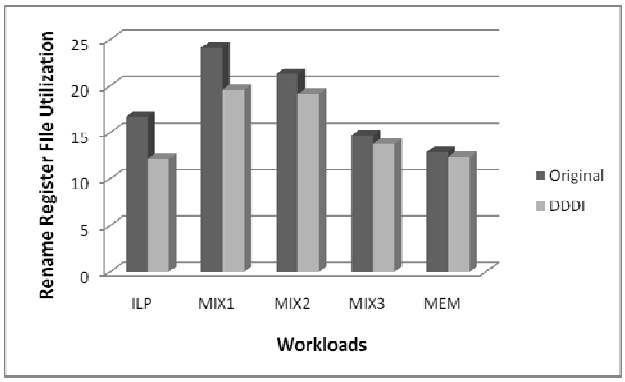

Figure 9 The residence time of instructions in rename register file

The throughput is affected by the flow rate of instructions in the pipeline. Figure 10 denotes the average residence time of instructions in the pipeline. The average residence time is dropped by 14.88%.

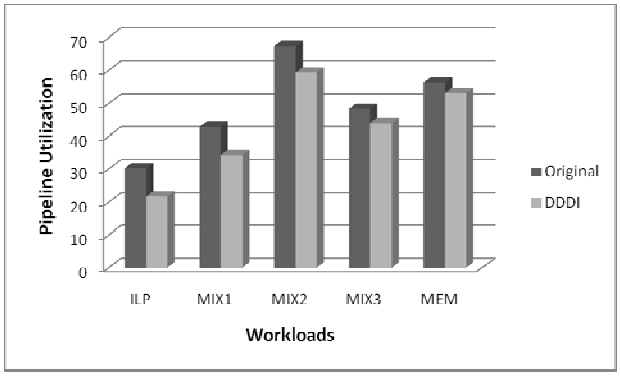

Figure 10 The residence time of instructions in pipeline

VI. RELATED WORK

In our previous research, it is found that rename register file and instruction queue is the most important resources that play an important role in performance.

Many researches focused on finding better solutions to make the usage of rename register file more efficiently. In SMT, to avoid the vicious competition among threads, static partition [25] and dynamic partition [26] [27] of rename register file are proposed.

Static partition allocates the rename register file to each thread evenly, and dynamic partition allocates the rename register file according the real-time performance of each thread. Threads that executes more quickly can get more rename register file resources.

In RMT, RRF is a key factor that affects performance. How to make better usage of RRF need further research. Existing techniques in SMT may be not practical for RMT. Because RMT is a special SMT. in RMT, the requirements for rename register resources between master threads and slave threads vary widely, so static partition strategy is not applicable for RMT. Master threads and slave threads make progress in the same speed, but don't require same amount of RRF resources, thus dynamic partition is not suitable for RMT, too.

VII. CONCLUSIONS:

According previous research, Instruction Queue affects the performance obviously. In RMT, IQ may be blocked by threads that encounter cache-miss easily. Just like coins have two sides, resources sharing among threads improve the resource usage on one hand and on other hand, and it may cause vicious competition.

In any processor, D-cache-miss load instructions make damage to performance. In redundant multi-threaded architecture, due to slave threads never encounter D-cache-miss, only master threads need special treatment.

This paper proposed DDDI, instructions of master threads that dependent on unresolved D-cache-miss-load instructions can't be dispatched into IQ until the load instruction is completed. Experiments show that the method is effective in improving the utilization of IQ, the processor's throughput has been significantly improved, and the weighted speedup used to measure single thread performance gain great increase, too.

ACKNOWLEDGMENT

The work is supported by the National Natural Science Foundation of China under Grant No. 60903033 and the National Basic Research and Development (973) Program of China (No. 2005CB321604).

REFERENCES

[1] Daniel Sorin, *Fault Tolerant Computing Architecture[M]*. Morgan&Claypool publishers. 2009.

[2] T. C. May and M. H. Woods, "Alpha-Particle-Induced Soft Errors in Dynamic Memories," *IEEE Transactions on Electronic Devices*, vol. 26, Issue1, pp: 2-9, January 1979.

[3] J. F. Ziegler and W. A. Lanford, "The effect of Cosmic Rays on Computer Memories," *Science*, vol. 206, No. 776, 1979.

[4] S. Mitra, N. Seifert, et,al, "Robust system design with built-in soft-error resilience," *IEEE Computer*, 2005, 38(2): 43-52.

[5] Karlsson, J., Liden, P., Dahlgren, P., Johansson, R., Gunneflo, U., "Using heavy-ion radiation to validate fault-handling mechanisms," *Micro, IEEE* Volume 14, Issue 1, Feb. 1994 Page(s):8 - 23

[6] Sosnowski,J., "Transient fault tolerance in digital systems," *Micro, IEEE*, Volume 14, Issue 1, Feb. 1994 Page(s):24 - 35

[7] D. C. Bossen, A. Kitamorn, K. F. Peick, and M. S. Floyd, "Fault-Tolerant Design of the IBM pSeries 690 Using POWER4 Processor Technology," *IBM Journal of Research and Development*, vol. 46, No. 1, pp:77-96, 2002.

[8] A. Wood. Data integrity Concepts, "Features and Technology," White paper, Tandem Division, Compaq Computer Corporation.

[9] W. W. Peterson and E. J, Weldon. Error-Correcting Codes, MIT Press, 1961.

[10] A Mendelson and N Suri, "Designing high-performance & reliable superscalar architectures: The out of order reliable superscalar (O3RS) approach," In: *Proc of IEEE/IFIP Int'l Conf on Dependable Systems and Networks*, New York, 2000, 473-481

[11] D. M. Tullsen, S. J. Eggers and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism". In: *Proc. of 22nd Annual International Symposium on Computer Architecture*, Santa Marguerite Liguria, Italy, 1995, pp: 392-403.

[12] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture [J]. Intel Technology Journal, Vol. 6, No. 1. February 2002, pp: 4-15.

[13] R. P. Preston et al, "Design of an 8-Wide Superscalar RISC Microprocessor with Simultaneous Multithreading," In: *Proc. Of IEEE International Solid-State Circuits Conference*, San Francisco, USA, February 2002, pp: 334 – 335.

[14] Eric Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," In: *Proc. Of 29th International Symposium on Fault-Tolerant Computing*, Madison, Wisconsin, 15-18 June, 1999, pp: 84-91.

[15] S. K. Reinhardt and S. S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," In: *Proc. Of the 27th International Symposium on Computer Architecture*, Vancouver, British Columbia, Canada, 10-14 June, 2000, pp: 25-36.

[16] T.N. Vijaykumar, Irith Pomeranz and Karl Cheng, "Transient-Fault Recovery Using Simultaneous

Multithreading," In: *Proc. Of the 29th Annual International Symposium on Computer Architecture*, Anchorage, Alaska, 25-29 May, 2002, pp: 87-98.
[17] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse," In: *Proceedings of 24th Annual International Symposium on Computer Architecture (ISCA)*, Denver, Colorado, USA, June 1997, pp: 194-205.
[18] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam, "A Complexity-Effective Approach to ALU Bandwidth Enhancement for Instruction-Level Temporal Redundancy," In: *31st Annual International Symposium on Computer Architecture (ISCA)*, pp. 376 – 386, June 2004.
[19] M. A. Gomaa and T. N. Vijaykumar, "Opportunistic Fault Detection," In: *32nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 172 – 183, Madison, Wisconsin, USA, June 2005
[20] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam, "SlicK: Slice-Based Locality Exploitation for Efficient Redundant Multithreading," In: *12th Annual International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 95 – 105, October 2006
[21] Dean M. Tullsen and Jeffery A. Brown, "Handling Long-latency Loads in a Simultaneous Multithreading Processor," In: *Proc. Of the 34th IEEE International Symposium on Microarchitecture*, Austin, USA, 1-5 Dec 2001, pp: 318-327.
[22] J. Sharkey, "M-Sim: A Flexible, Multi-threaded SimulationEnvironment," Tech. Report CS-TR-05-DP1, Department of Computer Science, SUNY Binghamton, 2005.
[23] Seungryul Choi and Donald Yeung, "Learning-Based SMT Processor Resource Distributing via Hill-Climbing," In: *Proc. of the 33rd International Symposium on Computer Architecture*, Boston, MA, USA, 17-21 June, pp: 239-251.
[24] T. Sherwood, et al, "Automatically Characterizing Large Scale Program Behaviour," In: *Proc. of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, 5-9 October, pp: 47-57.
[25] Steven E. Raasch and Steven K. Reinhardt, "The impact of resource partitioning on SMT processors ," In: *Proc of the 12th International Conference on Parallel Architecture and Compilation Techniques*, New Orleans, Louisiana, 2003 , pp: 15-25.
[26] Francisco J. Cazorla, Alex Ramirez, Mateo Valero et al, "Dynamically controlled resource allocation in SMT processors ," In: *Proc of the 37th International Symposium on Microarchitecture*, Portland, 2004, pp: 171-182.
[27] Hua Yang, Gang Cui and Xiaozong Yang, "Eliminating inter-thread interference in register file for SMT processors,". In: *Proc of the 6th International Conference on Parallel and Distributed Computing, Applications and Technologies*, Dalian, 2005, pp: 40-45.

Jie Yin, born in 1981. She received the B.S. degree in computer science and technology from the Tong Ji University in 2004, Now he is a Ph D candidate of Tongji University. His current research interests include fault-tolerant computing and microprocessor architecture.

Jianhui Jiang, born in 1964, Ph D, professor. He is currently Chairman of Department of computer science and technology at Tong Ji University. His research interests include fault-tolerant computing, software reliability engineering, microprocessor architecture, digital system design and testing, performance evaluation of computer systems. He is a senior member of Chinese Computer Federation (CCF).