Simulation of Fault Effect on Redundant Multi-Threading Execution

Huan-yu Tu and Sarah Tasneem

ABSTRACT – *One of the most cost-effective fault-tolerant schemes in commercial computing systems is using redundant multi-threading executions. These systems detect faults by redundantly executing more than one (usually two) instances of a program, and comparing the architected states of these instances. In these systems, though a fault might decrease the overall performance but will not affect the correctness of the execution. In this paper, through simulations we investigate the fault effects of redundant multi-threading executions by injecting transient faults to redundant computing systems. The results are presented and discussed here.*

Keywords – simulations, redundant, multi-threading, fault-tolerant, lock-stepping

1. INTRODUCTION

Redundant execution has been the most widely used scheme in commercial fault tolerant systems. Most redundant systems use a technique called lock-stepping in which two processors execute the same instructions and compare the results. When the results of the two operations are not identical, indicating the failure of one of the processors, the system has to perform a series of tests to determine which one is faulty. Easy-implementation and cost-effectiveness are major advantages of lock-stepping. Examples of such systems include IBM S/390 mainframes [1, 2] and Tandem Himalaya systems [3].

In simple core systems, lock stepped execution can be easily achieved by comparing the architected state of the machine. However, modern processors use speculative mechanisms which involve many non-architected states. The decisions taken by these mechanisms are validated. They may improve performance, but can never affect the correctness of the program. They take much more area in processor cores than the architected state, therefore are more likely to be affected by transient or permanent faults. When two processing elements have different internal speculative states, a non-deterministic output signal timing issues could make the lock-stepping scheme problematic without a full processor state reset. For example, a fault could flip a bit in the branch prediction table. This faulty entry could mislead the fetch unit of one processor. The processor pairs then start fetching and executing instructions in different paths of a branch, falling out of lock-stepped execution. This kind of non-deterministic timing for a modern processor has been discussed in [4].

Recently, simultaneous multi-threading (SMT) has gained popularity, and processors implementing SMT are becoming more prevalent. SMT shares resources such as, fetch bandwidth, instruction window, register file, caches between multiple threads. This fine-grained sharing, leads to effective utilization of resources between multiple threads, and hence higher throughput. SMT has led to micro-architectural fault-tolerant technique such as AR-SMT, RMT (Redundant Multi-threading), SRT (Simultaneous Redundant Threading). These techniques run the same program as independent threads on SMT processor, and compare their outputs. Some techniques compare committed stores (as in von Neumann machine there is a total-load-store order); while other approaches compare the end of a dependence chain. A mismatch during comparison indicates a fault which may or may not be recoverable depending on the approach.

Redundant execution on SMT is not lock-stepped execution. However, the non-determinism introduced by speculative techniques is the same. Motivated by: (a) non-determinism in modern processor cores; (b) prevalence and use of SMT for fault-tolerance, we study the effect of faults in branch predictor, branch target buffer, and return address stack of a SMT processor executing a redundant copy of the program. We also study the effect on error detection, and the slack it introduces between the two threads. We are aware of no other work that studies the effect of faults on structures maintained by speculative techniques.

In section 2, we discuss some typical fault detection techniques and its relevance to this paper. In section 3 we discuss our experimental setup and simulation model. In section 4, we present and discuss our simulation results. Conclusion is given in section 5.

2. FAULT DETECTION MECHANISMS

Most redundant systems use a technique called AR-SMT [7] for multi-threading fault detection. In AR-SMT, two threads, namely active thread (A-thread) and redundant thread (R-thread), are generated for each application thread. The A-thread leads and passes its result to the corresponding R-thread via a micro-architecture extension called delay buffer. Information passed via delay buffer includes control flow and data flow information e.g. register values, load and store

Manuscript received June 28, 2008.

Huan-yu Tu is with Eastern Connecticut State University, Willimantic, Connecticut 06226 USA (phone: 860-465-5050; e-mail: tuh@easternct.edu).

Sarah Tasneem is with Eastern Connecticut State University, Willimantic, Connecticut 06226 USA (e-mail: tasneems@easternct.edu).

addresses, etc. On the other hand the R-thread trails and exploits the branch outcomes and pre-fetching side effect of the A-thread, hence runs more efficiently. R-thread checks its results with those from A-thread. In case of discrepancy, R-stream committed state is used (the results agreed upon by both threads) to recover both A and R threads. AR-SMT does not have a strict timing requirement. However, the size of the delay buffer determines the acceptable range of slack between the two threads; otherwise, one of the threads will be blocked due to delay buffer.

SRT, SRT/R and SRT with Recovery are the recent work based on AR-SMT [8]. SRT/R proposes enhanced error recovery mechanism based on SRT. In SRT, a leading instruction may commit before the trailing thread checks for a fault. In contrast, SRT/R must not allow any leading instruction to commit before checking occurs since a faulty instruction cannot be undone once the instruction commits any result out of the sphere of recovery. To support a short slack, SRT/R's leading thread provides the trailing thread with branch predictions while SRT's leading thread provides branch outcomes.

Redundant Simultaneous Multi-Threading [9] studies redundant threading error detection mechanism in a commercial-grade SMT processor i.e. DEC Alpha 21464 processor. They uncovered some subtle implementation complexities and realized that RMT can be of more significant burden to single processor devices than indicated by previous studies.

In all of the redundant threading schemes as mentioned above, except SRT/R, trailing threads have much better instruction delivery streams than normal prediction and speculation mechanism can provide because the rarely faulty leading thread functions as an oracle for the corresponding trailing thread. It provides nearly perfect control, value prediction and data pre-fetching for its trailing peer. This fact motivated the slipstream processor design [5, 6], which is more performance oriented.

Compared with lock-stepped execution, all of the redundant multi-threading schemes have a relaxed, though limited, timing requirement i.e. slack between the leading and trailing threads. The slack is usually constrained to the buffering mechanism between the two threads for efficient execution of both threads; otherwise one of the threads would be blocked for synchronization. The coupling and buffering mechanisms between the redundant threads are not as straightforward as lockstep as the RMT paper indicates. There are many important implications to be well understood before such schemes can be applied in real commercial fault tolerant systems. An advantage of redundant multi-threading over lock-stepped execution is that it allows two threads to fall out synchronization, as long the lag between them would not be big enough to trigger watchdog timers.

In DIVA micro-architecture [10], there are two processor cores. One processor core is an aggressive, complex, and out-of-order superscalar core that would be fabricated with aggressively deep submicron technology. The other one is a conventional in-order "checker" processor core that would be fabricated with conventional technology that tolerates transient hardware faults by itself or by designer's assumption. Completed instructions are passed to the checker core for dynamic verification. Once the aggressive core finishes execution, all source operands for an instruction are available before checker processor executes that instruction. Hence, there is no dependence amongst the instructions inside the checker core. When an error is detected, the order of the instruction streams only helps in identifying the first faulty instruction. DIVA claims to be able to detect all of the transient and permanent errors, including the design errors.

3. SIMULATION MODEL

In this section, we discuss the simulators, benchmarks and fault injections.

3.1 Simulator

Two simulators, Sim-multi [11] and PharmSim [12], are used to study execution behavior under faulty branch prediction state. Since most simulators do not take fault-tolerance into account, we implemented fault injection mechanisms and emulated lockstep execution in our simulators to perform the study.

Front End	A 64KB instruction cache, 64KB YAGS branch predictor, 32 KB cascading indirect branch predictor, 64-entry return address stack. The front end is assumed to be perfect, i.e. it can fetch past taken branches, BTB is assumed for providing target addresses, which are available at decode, for direct branches.
Caches	The first-level data cache is 2-way set associative 64KB cache, with 64 byte lines, 3 cycle access latency. L2 is 4-way set associative, 2MB unified cache with 128-byte lines and 6-cycle access. Caches are write-back and write-allocate.
Execution Core	Two threads, 8-wide machine has 256-entry window, 4 load and store ports, all integer units are fully pipelined. The pipeline depth is 14 stages.

Table 1: Simulation Parameters for Sim-multi

Sim-multi is a complete timing simulator of the Alpha architecture loosely based on SimpleScalar toolkit was used for simulation. The simulated SMT-processor parameters are shown in Table 1. The machine is capable of executing two thread contexts simultaneously by fine-grained sharing of the resources. Two copies of the same benchmark program was executed on the two thread contexts. The fetch unit fetches instructions from the cache for the two thread contexts based on Icount fetch policy [13]. The policy counts the number of instructions from active threads that are currently in the instruction buffers, and fetches instructions from the thread

that has the fewest instructions. The assumption is that this thread is moving instruction through the processor quickly, and hence, marking most efficient use of the pipeline. Faults are detected by comparing the committed stores, that are buffered by each thread in their respective the committed stores in "store queues".

The experiments were performed on SPEC2000 integer benchmark programs. The programs were compiled with Compaq C compiler with peak optimization. All simulations were run on train inputs.

PharmSim is a combination of SimOS-PPC with SimpleMP. SimOS-PPC is a full system simulator that is capable of booting real PowerPC ISA based OS such as IBM AIX. SimpleMP is the multi-processor version of SimpleScalar simulator. PharmSim combines them together by dynamically translating PowerPC ISA into PISA ISA, which is actually supported by SimpleScalar. However, in our simulation, we do not utilize most of the features in PharmSim. It is chosen by its availability.

The branch predictor in PharmSim consists of two branch direction predictors. One is the bi-modal predictor that uses an array to hold 2-bit saturation counters for individual branch instructions. Another is the 2-level adaptive branch predictor with a global history pattern and a level two table. Bi-modal predictor has high prediction accuracy when branches are not correlated, while 2-level adaptive predictors are better when the correlation between the branches is more important. There is a higher level predictor that decides which prediction from these two predictors should be used.

As PharmSim is a precise but slow simulator, we configure it to run 200 million cycles for each of the SPEC2000 integer benchmarks. Faults are injected only during the first 100 million cycles. We do not inject faults in the second 100 millions cycles. However, we continue the measurement of the effect of fault, by measuring the latency of fault manifestation.

3.2 Branch Prediction

In section one, we introduced non-determinism which used by many speculative techniques. However, in this paper, we limit our study to an important micro-architectural technique: branch prediction and its related structures. We characterize how faulty branch prediction state affect processor execution behavior. Faults are injected into branch predictor table, branch target buffer and return address stack.

Branch prediction tables and Branch Target Buffer (BTB), are big arrays in a processor core. The BTB provides the target address of a branch, and branch prediction tables store history to decide whether a branch is taken or not taken. Both have an arrangement very similar to the arrangement of tag arrays in caches. RAS or Return address stack is smaller in area compared to the branch predictor and BTB tables. It is used to predict the return address for return instructions.

3.3 Fault Injection

Faults occur unpredictably in real hardware both in terms of time and space. In a simulator, especially one designed for architecture study, there is no direct mapping between simulator source code and real transistors or wires. Even the simulation time is not exactly the same as physical time. In our simulation, we only model transient faults in speculative state arrays; no effort was spent on logic, as there is no straightforward way to do so. Faults are injected into simulation "randomly" based on either simulation cycle time or the number of committed conditional branch instructions. Faults are primarily flips of either the decision of branch predictor, or a bit in the prediction structures. We talk more about this, in the next section on simulation results.

4. SIMULATION RESULTS

In this section, we present results from our simulations. The results in subsection 4.1 are from PharmSim, and are used to study the behavior of the faults. The subsection presents the percentage of faults that are manifested, for two different branch predictor configurations. The results in subsection 4.2 are from Sim-multi, and it talks about effect of faults on slack between main and redundant threads, performance, and branch prediction accuracy.

4.1 Fault Manifestation of Branch Predictors, BTB and RAS

Fault Injection. A random number is generated every 1024 cycles. If the number is divisible by 1024, a fault is injected. Fault injection in PharmSim was done by generating a random position in the predictor arrays and flipping that bit.

A fault is considered excited, when the predictor looks up an entry into which a fault has been injected. For branch predictor, we also count the number of unique manifestations, as a fault could be manifested multiple times, before it is removed. For BTB and RAS, once a fault is excited, it will be detected and removed shortly by the instruction commit stage. A fault is considered removed, when the branch predictor updates an entry into which a fault was injected.

	Config. 1	**Config. 2**
Bimod predictor table size	8192	2048
2-level predictor L2 size	8192	2048
2-level predictor History size	8	8
Combining predictor table size	8192	2048
BTB configure: sets, assoc.	8192 by 4	2048 by 4
RAS table size	64	32

Table 2: Two different branch prediction configurations

To perform the study on how the injected faults are actually excited and its latency for excitation, we pass a current simulation cycle token when injecting a fault. This token allows us to find latency between injection of a fault and its excitation or removal. The fault excitation/removal characteristics depend on both the configuration of the branch predictor and the runtime behavior of the benchmarks. Our simulations were run on two different branch predictior

configurations to see how array sizes change the faulty behavior. The two branch prediction configurations are summarized in the following Table 2.

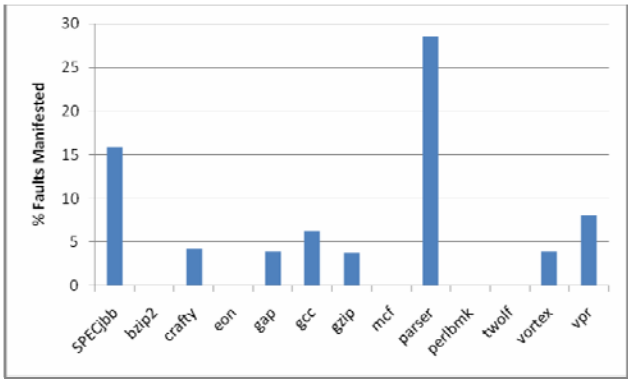

Figure 1: Percentage of faults injected into RAS that are manifested

RAS faulty behavior is captured in the Figure 1 for 64-entry (configuration 1) RAS (results are not presented for configuration 2 due to similarity except for larger numbers), the percentages of injected faults that are excited later are very small, usually well less than 10%. Taking into account that RAS is a stack and accesses are push/pop operations, one may conclude that in our simulation, the RAS array is empty most of the time, i.e. faults are injected onto unused entries by randomness. RAS faults are removed later with RAS push for a procedure call operation. A fault introduced into a valid RAS entry is manifested by a later RAS pop operation that returns a procedure's predicted return address to instruction fetch unit. Simulation results show that if a fault is injected onto RAS array, it will be either removed or excited immediately. Removal takes thousands of cycles while excitation is usually less than 1000 cycles (we do not present this result for brevity of this report). The results indicate that RAS operations are very frequent during execution because of the highly frequent procedure call/return instructions in the benchmark programs. Unlike BTB or branch predictor, all injected RAS faults are either removed or excited and removed shortly.

As expected, when we shrink RAS array size from 64 down to 32, the percentage of excited faults increases because more RAS entries are used on average. However, as the array size shrinks, the probability that it will be affected by transient faults also decreases. We summarize the characteristics for RAS as that, first, it's not very likely affected by transient faults due to it relative small size in the processor core; second, even if faults occur, they will most likely be destroyed or excited immediately. The effect on performance is negligible.

Faults injected into BTB, are manifested or removed after longer time. Thousands or even millions of cycles may have elapsed before a fault is actually excited. As only a small portion of the injected faults were excited or removed during our 200 million cycle simulation (this can be seen by adding the percentages of fault removed and excited in our simulation results), we can speculate that many faults could get excited or removed only after more than 100 million cycles. Figure 2 shows the fault manifestation and removal percentage for the first configuration.

Fault excitation data for level 2 branch predictor is shown in Figure 3. Next subsection will talk about this in great detail.

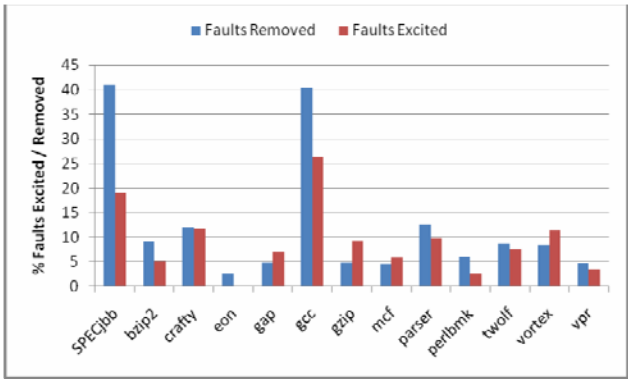

Figure 2: Percentage of faults excited and removed when faults are introduced into BTB

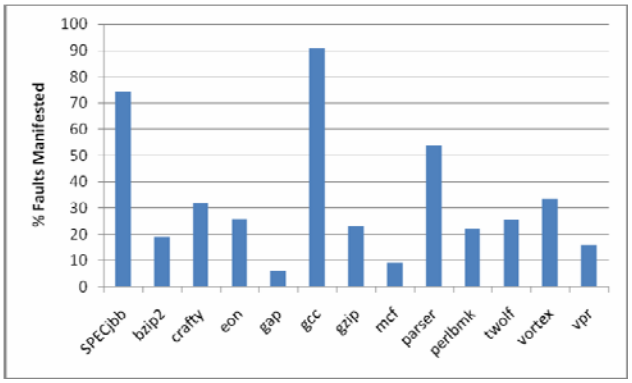

Figure 3: Percentage of faults excited when faults are introduced into branch predictor table

4.2 Faulty Behavior of Processor Pipeline

Two facts are observed: 1. Faults occur rarely; 2. Detailed timing simulator incurs a slowdown of a factor of 10,000 from native execution.

Hence, detailed execution of programs and injecting faults at the rate at which it occurs in real world is an impossible task at hand. We (like other computer architects) acknowledge *fact 1*, but ignore it for the study. This allows us to study the worst case behavior of faults on branch prediction structure, and its impact on performance and slack, when injected at various points in the execution of the program. The study, as far as possible, tries to reduce the interfering effect of multiple injected faults.

Base Case: The base case is a SMT processor with parameters described in the previous section. The two copies of the program executing are called: "main" thread, and "redundant" thread respectively. Stores committed by the two threads are queued separately, and compared. The two threads are not synchronized. The stores are not bound to commit at the same clock cycle, because of the finegrained sharing of

resources between the two threads. Resources such as read and write ports, scheduling and selection logic, fetch bandwidth are some of the important factors that will lead to slack between main and redundant threads. The committed stores are queued in a FIFO store queue (due to in-order commits), and their results compared. For this structure to be off-the critical path of the program, there should be enough entries in the queue, to buffer committed stores. If the queue is full, no more instructions (head of which is a store instruction) are allowed to commit.

Performance and Branch Prediction Accuracy: To study the effects that faults in branch prediction hardware could introduce, we inject faults into the branch predictor. We do this by flipping the branch predictor's taken or not taken decision. In the first part, we flip the decision of a branch that is picked randomly in a window of 1,000 conditional branches executed by the redundant thread. Only decisions of the redundant thread are flipped. The flipped decision is returned to the front-end, which continues fetching from taken or not-taken path depending on the decision. The faulty decision could be

1. **Misprediction.** The branch predictor predicts the branch correctly, but the decision is flipped because of a fault. The misprediction is detected later, when the branch is executed. The wrong path instructions that are in-flight and/or executing, are squashed, and the front-end starts fetching the correct path. Hence, this faulty decision reduces the branch prediction accuracy, and may degrade performance. It also puts pressure on the committed stores queue (Reason will be explained later.)
2. **Correct prediction.** The branch predictor mispredicted the branch. But, the fault flipped the decision. This boosts branch prediction accuracy, and may improve performance.

On average, every branch out of 1,000 branches (i.e. 0.1% of the total executed branches) has a decision that conflicts with branch predictor's decision. The fault-injection rate, might not allow a fault's effect to die down, before another fault is introduced. However, the results are for merely demonstrating the worst-case hit in performance because of faults. The store queue is assumed to infinite to ignore the effect of stalls when the queue is full. Hence, the results are the effect of faulty-decisions only (besides finegrained resource sharing). Figure 4 shows the percentage relative change in IPC (Instructions per cycle) when a fault is injected every 1000 conditional branches executed (w.r.t base case). Here, positive percentages indicate improved performance. Store queue size was assumed to be infinite. In Figure 4, the IPC of the execution of redundant thread with 0.1% faulty branch prediction decisions relative to the execution without any faults. `crafty`, `gap`, `parser` and `vpr` have no impact. `gcc` has improved performance, and the rest have degraded performance. The reason for a change in IPC is shown in Figure 5. Figure 5 shows the percentage relative change in branch prediction accuracy when a fault is injected every 1000 conditional branches executed (w.r.t base case). Positive percentages indicate reduced branch prediction accuracy. Store queue size was assumed to be infinite. In Figure 5, the increase in mispredictions of a processor with faulty predictor relative to one with no faults. `eon` has the highest increase of 0.84% in mispredictions. This increase in misprediction, leads to around 2% degraded performance. `gcc` is the interesting aspect of this result. The benchmark has 0.05% improvement in its prediction accuracy (hence the negative bar). This leads to 5% improvement in performance. `vpr`, `parser`, `gap` have no degraded performance, even with a reduced prediction accuracy (i.e. increase in mispredictions). For most benchmarks, with the assumption of *infinite sized store queue*, faults do not affect performance significantly.

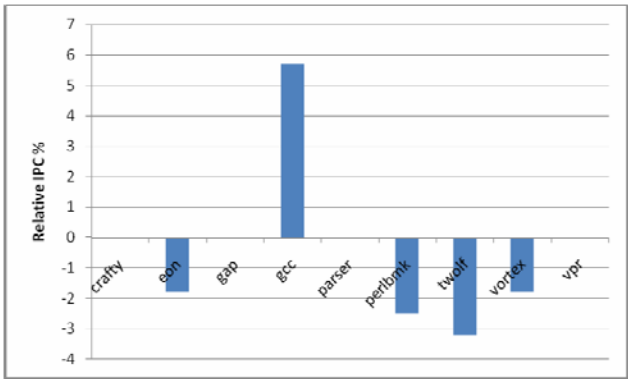

Figure 4: Percentage relative change in IPC when a fault is injected every 1000 conditional branches executed.

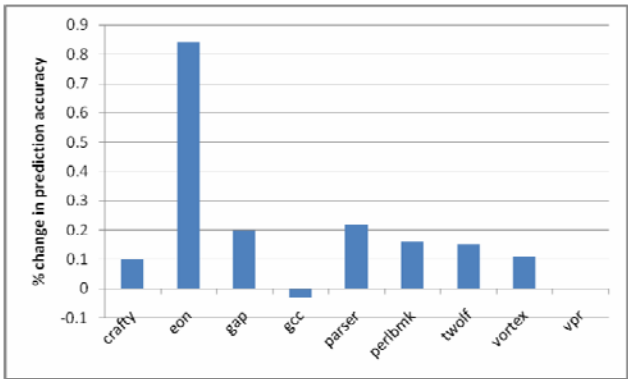

Figure 5: Percentage relative change in branch prediction accuracy when a fault is injected every 1000 conditional branches executed.

Slack: Branches are control transfer instructions. They allow us to jump to different instructions in a sequential program, and continue execution. Since branches are big performance impediments (inability to fetch beyond branches is the biggest problem), they are predicted by the front-end, which continues its fetch from the predicted PC. The branch is later executed, and prediction is checked. Correct predictions can lead to big boost in performance. However, as previous results showed, branches and performance are not directly correlated. The reason for this: all branches are not equally important [14] (we do not wish to describe this in any more detail, as it requires background in processor micro-architecture). This, along with the fact that, resources are

shared at a fine-grained level in SMT processor, could lead to enormous pressure on store queue. The thread going down the wrong-path due to faulty decision will get to committing the store which is in the head of the other thread's queue much later. Many other stores could have been committed by the other thread by then, which will lead to store queue getting full, and stalling the thread.

To demonstrate and study the slack caused by the above problem, we consider the following case: the branch predictor flips one decision every 10 million conditional branches executed by the redundant thread. The reason for reducing the probability of fault, by increasing the distance between two faults it to filter out the "snow-balling" effect that a fault might have (i.e. effect of one fault affecting subsequent fault). Note that faults are introduced successively to study the effect of it at various points in the program.

	% Stores Lead	% Stores Lag	Lag By (cycles)
crafty	27	73	100
eon	0	100	20,000
gap	4	96	400,000
gcc	30	70	5,000
parser	2	98	1,480,000
perlbmk	0	100	837,000
twolf	0	100	339,000
vortex	0	100	21,000
vpr	39	61	250

Table 3: Percentage of committed stores in the redundant thread that lag or lead with respect to the main thread. Fault is introduced every 10 million conditional branches executed.

	Unique Fault Manifestation (%)	Fault Manifestation (%)
crafty	7.7	71
eon	2.3	16
gap	9.6	366
gcc	43.8	465
parser	9.8	63
perlbmk	7.5	549
twolf	4.9	41
vortex	15	52
vpr	1.7	10

Table 4: Percentage of faults introduced into the branch predictor table are manifested as (unique) faults

Results for this study are listed in Table 3. The model detects faults by comparing the stores that have been retired by the main and redundant thread. Columns 2 and 3 in Table 3 list the percentage of committed stores in the redundant thread that lead and lag respectively with respect to the main thread. A committed store in the redundant thread leads, when it commits earlier than the main thread. This lead slack between the main thread and redundant thread was found to be negligible and hence not reported in the table. The lag slack is shown in column 3 of the table. The number of cycles varies drastically from 100 to 1 million cycles. The large slack is due to the following reasons:

1. YAGS [15] branch predictor that we used in the simulator has a high prediction accuracy (around 95%) on SPEC2000 benchmarks. Hence, the flipped branch prediction decision introduced because of a fault misleads the redundant thread into executing wrong path instructions. The speculative execution of instructions waste cycles, and introduces slack between the two threads. The slack depends on how long it takes for the redundant thread to resolve the branch, squash wrong-path instructions, fetch, execute and commit instructions from the right path.
2. Fine-grained sharing of resource across multiple threads in a SMT processor. SMT processors are sensitive to resource usage, and accordingly direct the front-end to fetch from different threads based on fetch policy. This could severely penalize the poorly-performing redundant thread, leading to large slack between main and redundant thread.

Fault Manifestation: So far, we have been studying the effect of injecting faults by flipping the decision of the branch predictor. We conclude this section by studying the effect of injecting faults into the branch predictor table. Most modern branch predictors are very sophisticated. They store information about the branch's direction in its previous instances, and also the path that lead to the branch. For this study, we randomly pick an entry in the branch predictor table and inject fault, by changing the confidence counters. The fault that has been introduced will be considered "manifested" only when the faulty entry is read for taking a decision. It is not "manifested" when a faulty entry is destroyed (i.e. if the entry is thrown out of the table, or if it is over-written). A faulty entry can be read multiple times, before it is destroyed. Table 4 lists the fault manifestation rate for various benchmarks. Column 2 of table 4 shows the percentage of unique fault manifestation. Column 3 of the table shows the percentage of "all" fault manifestations. For `gcc`, 43% of faults introduced in the branch predictor table, are read for taking a decision. However, a fault that is introduced is read more than 10 times (hence, 465% in column 3) before it is destroyed. An entry that was used for taking a decision for a branch (at fetch time), is updated (i.e. overwritten) when the branch commits. A faulty entry being read more than 10 times before being updated, leads us to the following observations: (a) there are many in-flight instructions and, (b) many instances of the same branch are in-flight before they are committed. More in-flight instructions is due to large instruction window (256 entries). More instances of the same branch is because of the characteristic of the benchmark. `gcc` has short loops, and the conditional branch that forms the loop is executed many times, before the first instance of it is committed. `gap` and `perlbmk` are pointer intensive benchmarks, looping on complex data structures.

5. CONCLUSION

Redundant lock-stepped execution is a popular technique for designing fault-tolerant computing systems. In this paper, we studied the non-determinism in a branch predictor. We randomly introduced faults into branch predictor table, branch target buffer, and return address stack, and see how it impacts, (a) performance, (b) branch prediction accuracy, (c) slack (in cycles) between the main and redundant thread, and (d) fault manifestation.

We conclude that, deep speculation in future processors will complicate the design of fault-tolerant systems significantly. Speculative techniques maintain large structures, most of which are on the critical path of the execution of the program. Hence, these structures cannot be protected by error correcting codes. Unfortunately, the probability of transient faults in these structures in submicron designs is very high. Deep speculation can improve performance; however, faults in these structures could also adversely affect performance.

REFERENCES

[1] M. Mueller et al. Ras strategy for IBM S/390 G5 and G6. In *IBM journal of Research and Development. VOL 43. NO 5/6*, Sept/Nov 1999.

[2] L. Spainhower and T. A. Gregg. IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective. In *IBM journal of Research and Development. VOL 43. NO 5/6*, Sept/Nov 1999.

[3] Online. Tandem nonstop himalaya system. In *WEB*, page http://nonstop.compaq.com/, 2002.

[4] Andrea C. Arpaci-Dusseau Remzi H. Arpaci-Dusseau. Fail-stutter fault tolerance. In *The Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII)*, pages pages 33–38, May 2001 Schloss Elmau, Germany.

[5] Z. Purser K. Sundaramoorthy and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[6] K. Sundaramoorthy Z. Purser and E. Rotenberg. A study of slipstream processors. In *33rd International Symposium on Micro Architecture*, Dec. 2000.

[7] Eric Rotenberg. AR-SMT: A micro-architecture approach to fault tolerance in microprocessors. In *Proceedings of the 29th International Symposium on*
Fault Tolerant computing systems, June 1999.

[8] Karl Cheng T. N. Vijaykumar, Irth Pomeranz. Transient fault recovery using simultaneous multithreading. In *Proceedings of ISCA*, 2002.

[9] Steven K. Reinhardt Shubhendu S. Mukherjee, Michael Kontz. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of ISCA*, 2002.

[10] Todd M. Austin. Diva: A reliable substrate for deep submicron micro-architecture design. In *Proceedings of the 32nd Annual IEEE/ACM international Symposium on Micro-Architecture*, Nov, 1999.

[11] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, Multiscalar Processors, In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 414 - 425, 1995.

[12] PharmSim. http://www.cdl.edu/pharmsim/index.html.

[13] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing onchip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[14] C.B. Zilles and G.S. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proc. 27th International Symposium on Computer Architecture*, June 2000.

[15] A. N. Eden and T. Mudge. The YAGS branch prediction scheme. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 69 - 77, November 1998.