

Design Trade-Offs and Deadlock Prevention in Transient Fault-Tolerant SMT Processors

Xiaobin Li

Jean-Luc Gaudiot

Abstract

Since the very concept of Simultaneous Multi-Threading (SMT) entails inherent redundancy, some proposals have been made to run two copies of the same thread on top of SMT platforms in order to detect and correct soft errors. This allows, upon detection of an error, for the rolling back of the processor state to a known safe point, and then a retry of the instructions, thereby resulting in a completely error-free execution. This paper focuses on two crucial implementation issues introduced by this concept: (i) the design trade-off between the fault detection coverage versus the design costs; (ii) the possible occurrence of deadlock situations. To achieve the largest possible fault detection coverage, we replicate the instructions fetched in order to generate the redundant thread copies. Further, we apply the SMT thread scheduling at the instruction dispatch stage so as to lower the performance overhead. As a result, when compared to the baseline processor, our simulation results show that by using our two new schemes, the performance overhead can be reduced down to as little as 34% on the average, down from 42%. Finally, in the fault-tolerant execution mode, since the two copied threads are cooperating with one another, deadlock situations could be quite common. We thus present a detailed deadlock analysis and then conclude that allocating some entries of ROB, LQ, and SQ for the trailing thread is sufficient to prevent such deadlocks.

1. Introduction

The multi-threading execution paradigm inherently provides the spatial and temporal redundancy necessary for fault-tolerance: the basic idea is to run two copies of the same threads on a Simultaneous Multi-Threading (SMT) platform [16, 15, 14, 9, 20]. The results of the two copied threads are then compared in order to detect transient faults. This is the temporal redundancy which is implemented by SMT: for instance, assume a soft error occurred in a functional unit (FU) when executing an instruction from

thread#1. Even though the FUs are typically shared between active threads, since the soft error is transient, as long as the same instruction from thread#2 is executed at a different moment, the results of the redundant execution from the two copied threads would not match. Furthermore, if any fault in the pipeline is detected, the checkpoint information can then be used to return the processor to a state corresponding to a fault-free point. After that, the processor can retry the instructions from the recovered point.

Nevertheless, a design trade-off is associated with adopting the above basic idea. Generally speaking, it requires the redundant execution to have appropriate fault detection coverage for a given processor component. Hence, the higher the fault detection coverage expected, the more redundant the execution required. However, redundant execution inevitably comes at the cost of performance overhead, added hardware, increased design complexity, etc. Consequently, how to trade the fault detection coverage off the added costs is essential for the practicality of the basic idea. Specifically, consider the need to generate redundant executing threads: given a general five-stage pipeline [5], containing instruction fetch, decode, issue, and retire stages, all stages can be exploited for that requirement [8, 13, 7, 18, 15, 14, 9, 20]. Take the fetch stage as an example, we can generate the redundant threads by fetching instructions twice [15, 14, 9, 20]. Since the instruction fetch stage is the first pipeline stage, the redundant execution would then cover all the pipeline stages, thus, the largest possible fault detection coverage could be achieved. However, allowing two redundant threads to fetch instructions would possibly end up halving the effective fetch bandwidth. Consequently, that halved fetch bandwidth would be an upper bound for the maximum pipeline throughput. Additionally, the redundant thread generated in the fetch stage would then compete not only for the decode bandwidth, the issue bandwidth, and the retire bandwidth, but also for IssueQ and ROB capacity, which are all identified as key factors that affect the performance of the redundant execution [18]. Conversely, we can re-issue the retired instructions from ROB back to the

functional units for redundant execution. In doing so, the bandwidth and spatial occupancy contention at IssueQ and ROB can be relieved, thus the performance overhead can be lowered [18]. However, this retire stage-based design comes at the price of a smaller fault detection coverage: only the EXE stage would be covered.

Given these trade-off considerations, we simply fetch the instructions once and then immediately copy the fetched instructions to generating the redundant thread. In doing this, there is no need for partitioning the fetch bandwidth between the redundant threads. Moreover, we can rely on the dispatch thread scheduling and redundant thread reduction to relieve the contention in the IssueQ and ROB. Both techniques lower the performance overhead (details can be found in Subsections 2.2 and 2.3).

Other than the design trade-off, another issue associated with the basic idea is the need to prevent deadlocks. In a fault-tolerant SMT design, two copies of the same thread are now cooperating with each other. Such cooperation could cause deadlocks [9]. We present a systematic deadlock analysis and conclude that as long as ROB, LQ, and SQ have allocated some dedicated entries for the trailing thread, the identified deadlock situations can be prevented. Based on this conclusion, we propose two ways for the prevention of any deadlock situations (more details in Section 3): (i) statically allocate entries in ROB, LQ, and SQ for the redundant thread copy; (ii) dynamically monitor for deadlocks.

The remainder of the paper is organized as follows: in Section 2, we will describe in detail our proposed fault-tolerant transient SMT designs. The deadlock analysis and prevention mechanisms are explained in Section 3. The simulation results to demonstrate our observations appear in Section 4. Section 5 will present related work on prior SMT transient fault-tolerant designs, while we conclude in Section 6.

2 Lowering the performance overhead

As discussed, to lower the performance overhead, we simply fetch the instructions once and then immediately copy the fetched instructions to generating the redundant thread. However, in doing so, faults in three major components in the fetch stage: I-cache, PC, and BPU might not be covered. In particular, any transient faults which would happen inside the I-Cache might not be detected. However, to protect the I-Cache, we can implement ECC-like mechanisms that are very effective in handling transient faults in memory structures [11, 2, 1]. Further, the fault occurring in the BPU will have no effect on the functional correctness of program execution [10]; however, the critical component PCs must also be protected by ECC-like mechanisms.

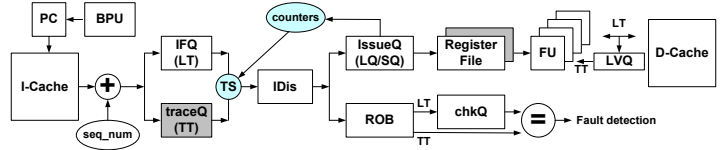


Figure 1. Functional diagram of our proposed fault-tolerant SMT data path

2.1 Copy fetched instructions to generate the redundant thread

As shown in Figure 1, our instruction copy operation is simple: just buffer the fetched instructions into two instruction queues, hence, the copy operation would not enlarge the pipeline cycle frequency, nor would another pipeline stage be added. To be specific, each instruction fetched is bound to a sequential number and a unique thread ID. For instructions that are stored in IFQ, “leading thread” (LT) is used as their thread ID, whereas for those stored in another IFQ, called trace queue (traceQ), “trailing thread” (TT) is used. It should be noted that traceQ also serves in the two performance overhead lowering techniques which will be described in detail in the following subsections.

2.2 Reduce TT to be lightweight

Focusing on our redundant execution mode, the key factors that affect the performance of redundant execution can be identified as: the bandwidth contention as far as issue, execution, and retire operations are concerned, as well as the capacity contention in IssueQ and ROB [8, 18]. This subsection addresses these types of resource contention by reducing TT to be as light as possible (remember that executing TT is merely for fault detection purposes). In doing so, the competition between IssueQ, ROB, and FUs could then be reduced.

2.2.1 Prevent TT mispredicted instructions from being dispatched

The number of dynamic mispredicted instructions might be a significant portion of the total fetched instructions, for example, Kang, *et al.* [6] observed that nearly 16.2% to 28.8% of the instructions fetched would be discarded from the pipeline even with a high branch prediction accuracy. Hence, if we could prevent mispredicted instructions in TT from being dispatched, the effective utilization of IssueQ, ROB, and FUs would be improved accordingly. Based on this observation, we leverage LT branch resolution results to completely prevent the mispredicted instructions in TT from being dispatched. It should also be noted that in our

designs neither a branch outcome queue [14] nor a branch prediction queue [20] is needed.

Specifically, when encountering a branch instruction in traceQ, the dispatch operation will check its prediction status: if its prediction outcome has been resolved by its counterpart from LT, we continue its dispatch operation; otherwise, the TT dispatch operation will be paused. In order not to pause the TT dispatch operation, LT must be executed ahead of TT. The LT ahead of TT execution mode is called staggered execution. (More on this in Subsection 2.3.2.)

To set up the TT branch instruction status (the initial status is set as “unresolved”), every completed branch instruction from LT will search traceQ to match its TT counterpart.¹ If the branch has been correctly predicted, the status of the matched counterpart TT branch instruction will be assured as “resolved.” Conversely, if the branch has been mispredicted, LT will perform its usual branch misprediction recovery, while at the same time it will flush all those instructions inside traceQ that are located behind the matched counterpart branch instruction. In other words, LT performs the branch misprediction recovery for both LT and TT, thus, TT does not recover any branch misprediction by itself. After recovery, the status of the TT branch instruction will be set up as “resolved.”

2.2.2 Adopt the Load Value Queue (LVQ) design

We adopt the LVQ design [14] and include it in our design as shown in Figure 1. Basically, when an LT load fetches data from the cache (or the main memory), the data fetched and the matching tag associated are also buffered into the LVQ. Instead of accessing the memory hierarchy, the TT loads simply checks and matches the LVQ for the data fetched. In doing so, TT might reduce the D-cache miss penalties and in turn improve its performance. It should be noted here that in order to fully benefit from the LT data prefetching, we must guarantee that LT is always ahead of TT, which requires a staggered execution mode.

2.3 Apply the dispatch thread scheduling

A well-known fact is that there are many idle slots in the execution pipeline. Hence, we must make sure that the redundant execution will exploit those idle slots as much as possible in order to circumvent the identified performance

¹The sequential numbers provide the mean for matching two redundant threads instructions. As we have seen, each instruction fetched is associated with a sequential number at first, then the fetched instruction is replicated to generate the redundant thread. In doing so, two copied instructions will have the same sequential numbers in different threads. It should also be noted that such a sequential number feature has been implemented, for example, in the Alpha and PowerPC processors [4].

affecting contentions [8, 18]. This subsection presents two such designs.

2.3.1 Modify ICOUNT policy and apply at the dispatch stage

To exploit the idle slots, we must ensure that whenever one thread is idle for any reason, the execution resources can be *promptly* allocated to another thread that can utilize those resources more efficiently. Based on this observation, the ICOUNT policy was proposed to schedule threads in order to fill IssueQ with issuable instructions, *i.e.*, restrict threads from clogging IssueQ [19]. However, we argue that it is the dispatch stage that directly feeds IssueQ with useful instructions, hence, scheduling threads in the dispatch stage level would react more promptly to the thread idleness in IssueQ.² Therefore, we modify the ICOUNT policy as follows (also see in Figure 1): at each clock cycle, we count the number of instructions that are still waiting in the IssueQ from LT and TT. A higher dispatch priority is assigned to the thread with the lower instruction count. More specifically, when the dispatch rate is eight instructions per cycle, the selected thread is allowed to dispatch as many instructions as possible (up to eight). If any dispatch slot is left from the selected thread, the alternate thread would consume the remaining slots. The above policy is denoted as “ICOUNT.2.8.dispatch.”

2.3.2 Slack dispatch scheme

While developing techniques to reduce TT to be lightweight as shown in Subsection 2.2, we found that a staggered execution mode [7, 14, 20] is beneficial for those techniques. To that end, the “slack dispatch” scheme is proposed: in the instruction dispatch stage, if the selected thread is TT, we check the instruction distance between LT and TT. If the distance is less than a predefined threshold, we skip the TT dispatch operation and continue buffering TT in traceQ. This means that the size of traceQ (the entry number of traceQ) must meet the following requirement:

$$\text{sizeof}(\text{traceQ}) > \text{sizeof}(\text{IFQ}) + \text{predefined distance} \quad (1)$$

Moreover, for purposes of fault-detection, all retired LT instructions and their execution results are buffered into the checking queue (chkQ), as shown in Figure 1. Hence, TT is responsible for triggering the result comparison.³ We further assume the register file of TT is protected by ECC-like mechanisms. This means that, if any fault is detected,

²In an on-going study, we plan to demonstrate that this dispatch stage thread scheduling concept is also applicable for the non-redundant multi-threading execution.

³The high chkQ bandwidth requirement could demand some special attention. To handle it, we would apply a similar design to that shown in [20].

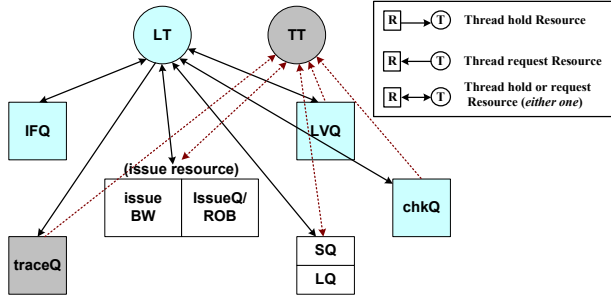


Figure 2. Resource allocation graph for deadlock analysis

the register file state of TT could be used to recover that of LT.

3 Deadlock analysis and prevention

As pointed out before, the two copies of a thread cooperate with each other for fault checking and recovering. However, if not carefully synchronized, such cooperation could result in deadlock situations where neither copy could make any progress [9]. To prevent this, a detailed analysis and appropriate synchronization mechanisms are necessary.

3.1 Deadlock analysis

Resource sharing is one of the underlying conditions of deadlocks [17]. Indeed, it should be noted that there is much resource sharing between the two thread copies. For example, IssueQ is a shared hardware resource and both thread copies contend for it. The availability of instructions being issued is another type of resource sharing: the issue bandwidth is dynamically partitioned between the two thread copies. Figure 2 shows the resource allocation graph. Take chkQ as an example, only if there is a free entry in chkQ could LT retire its instruction and back up the retiring instructions and execution results there. On the other hand, the entry in chkQ can only be freed by TT: only after an instruction has been retired and compared, can the corresponding entry in chkQ be released. Further, due to the similarity between dispatch and issue operations, we combine them under the term “issue resource” in the discussion which follows.

Based on Figure 2, we can list all possible circular wait conditions. However, some conditions obviously do not end up in a deadlock, for example, “LT → traceQ → TT → SQ → LT.” After exhausting the list, we describe all the possible deadlock scenarios as follows:

1. LT → chkQ → TT → issue resource → LT.

Scenario: When chkQ is full, LT cannot retire its

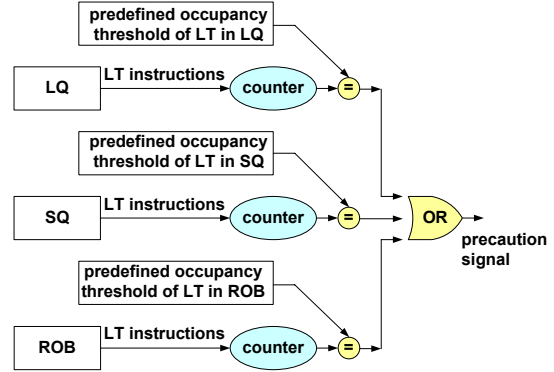


Figure 3. Dynamic monitoring for deadlock prevention

instructions. Then, those instructions ready to retire from LT are simply stalled in ROB. If that stalling ends with an ROB full of instructions from LT (the case in which ROB is full of instructions from LT could be exacerbated by the fact that LT is favored by the dispatch thread scheduling policy for the stagger execution mode), the instruction dispatch operation will be blocked, thus, TT will be stalled in traceQ. Consequently, no corresponding instructions from TT can catch up to release the chkQ entries and then a deadlock can happen. In summary, the condition for this deadlock situation is the following:

Observation 1 When chkQ is full and ROB is full of instructions from LT, a deadlock happens.

2. LT → LVQ → TT → issue resource → LT.

Observation 2 When LVQ is full and LQ is full of instructions from LT, a deadlock happens.

Similarly, the stalled load instructions could end up in a full ROB, thus, the instruction dispatch operation will be blocked. Hence, a deadlock observation follows:

Observation 3 When LVQ is full and ROB is full and no load instructions from TT in ROB, a deadlock happens.

3. LT → SQ → TT → issue resource → LT.

Observation 4 When SQ is full of instructions from LT, a deadlock happens.

3.2 Deadlock Prevention

Based on the above systematic deadlock analysis, we propose two mechanisms to handle the possible deadlock situations: static hardware resource partitioning and dynamic deadlock monitoring.

3.2.1 Static hardware resource partitioning

In static hardware resource partitioning, *i.e.*, each thread has itself allocated resources, the deadlock conditions identified can be broken such that we can prevent the deadlock.⁴ For example, we can partition the ROB in order to prevent the possible deadlock situation identified in Observation 1: if some entries of the ROB are reserved for TT, TT dispatch operations could continue since, when chkQ is full, the partitioned ROB cannot be full of instructions from LT. Subsequently, those dispatched TT instructions will be issued and their execution completed afterwards. After completion, they will trigger the result comparison and then free the corresponding chkQ entries if the operation was found to be fault-free. After some chkQ entries have been freed, LT is allowed to make progress.

Moreover, we find that only three hardware resources (ROB, LQ, and SQ) need to be partitioned in order to prevent all the deadlock situations that we identified: Partitioning the ROB to break the deadlock situation identified in Observation 1: ROB will never be full of instructions from LT such that TT will be dispatched and then chkQ will be released. Similarly, partitioning LQ to break the deadlock situation identified in Observation 2; Partitioning SQ to break the deadlock situation identified in Observation 4. Now considering Observation 3, when LVQ is full, an LT load instruction LD_k in LQ cannot be issued. However, since ROB is now partitioned between LT and TT, the stalled load instruction LD_k in ROB will only block LT from being dispatched. In other words, the TT dispatch operation will not be blocked by the stalled load instruction LD_k , thus, for example, another load instruction LD_i from TT will be dispatched which will then release the LVQ entry occupied by the counterpart load instruction LD_i from LT. Once free LVQ entries are made available, the stalled LT load instruction LD_k can be issued. In summary, we have the following observation:

Observation 5 *For each ROB, LQ, and SQ, allocating some dedicated entries for TT will prevent the deadlock situations identified.*

It should be noted, however, that static hardware resource partitioning has some performance impact on the SMT, particularly when partitioning ROB, LQ, and SQ (the instruction issue queues for load and store instructions) [12]. To mitigate this performance impact, we allocate some minimum number entries for TT to prevent deadlocks and the remainder of the queue is shared between LT and TT. Hence, the maximum available entry number for LT is the total queue entry number minus the reserved entry number whereas the maximum available entry number for

⁴The static hardware partitioning approach is similar to the deadlock avoidance design shown in [9]. However, based on our systematic deadlock analysis, we clarify the implementation details of the basic approach.

Algorithm 1: Dispatch thread scheduling policy (with dynamic deadlock monitoring)

```
Apply ICOUNT.2.8.dispatch policy;
if selected thread is LT AND IFQ not empty AND no
precaution signal then
    Dispatch from IFQ;
else if distance between LT and TT meets predefined
stagger execution mode AND traceQ is not empty
AND not an unresolved branch instruction then
    Dispatch from traceQ;
else
    nothing to be dispatched;
end
```

TT is the total queue entry number.

3.2.2 Dynamic deadlock monitoring

From the deadlock analysis, we can also conclude that if we could dynamically regulate the progress of LT such that neither ROB nor LQ and SQ can be filled with instructions only from LT, the deadlock situations identified can be prevented. As illustrated in Figure 3, we dynamically count the number of instructions from LT in ROB, LQ, and SQ, respectively, and then a caution signal is generated if at least one of the numbers of counted instructions exceeds the corresponding predefined occupancy threshold. Furthermore, as long as the caution signal is generated, the dispatch thread scheduling policy will pause LT from being dispatched.

To be specific, the comprehensive dispatch thread scheduling policy we developed is listed in Algorithm 1: first, we apply the ICOUNT.2.8. dispatch policy. If the selected thread is LT, we must then check whether IFQ is empty since no instruction can be dispatched in the case of an empty IFQ. Furthermore, we need to make sure no caution signal has been generated. If there is a such signal, we must stop dispatching from LT. On the other hand, if the selected thread is TT, we check the following conditions before dispatch TT: (1) the staggered execution mode requirement; (2) is traceQ not empty; (3) not encountering an unresolved branch instruction.

It should be noted that the dynamic deadlock monitoring approach offers a higher design flexibility than the static resource partitioning: by adjusting the predefined occupancy thresholds, we can manipulate the resource allocation between the cooperating threads. However, this flexibility comes at the cost of additional hardware as well as a more complicated thread scheduling policy.

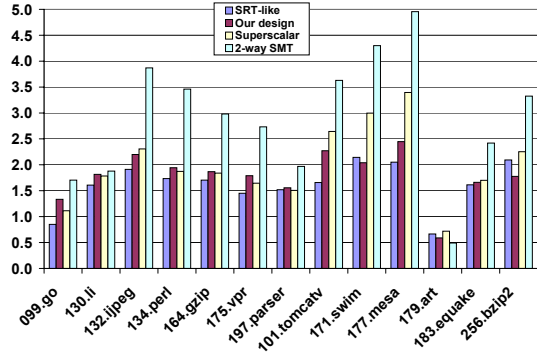


Figure 4. IPC comparison of two FT-SMT processors, superscalar, and two-way SMT

4 Fault-tolerant design overhead evaluation

This section will analyze the performance and hardware overhead associated with our fault-tolerant design presented in Section 2 and Section 3.

4.1 Performance overhead evaluation

To examine the performance overhead of a fault-tolerant SMT processor,⁵ we have developed SMT performance simulators based on the SimpleScalar toolset [3]. Moreover, two fault-tolerant SMT processor designs have been simulated: one is similar to that discussed in [14] whereas our design has been presented in Section 2 and Section 3. The major differences between two designs fall into three general categories: first, SRT-like design generates redundant threads by fetching twice whereas ours does so by copying fetched instructions (hence only one fetch). Secondly, we use ICOUNT.2.8.dispatch policy whereas ICOUNT.2.8 [19] is applied in an SRT-like design. Finally, to eliminate TT mispredicted instructions from being executed, an SRT-like design relies on a Branch Outcome Queue but we use our dispatch thread scheduling policy to prevent those mispredicted instructions from even being dispatched. Table 1 summarizes the key parameters of the processors simulated. The others microarchitectural configuration parameters such as fetch bandwidth, function units, L1 I- and D- caches, unified L2-cache, and main memory, are similar to the ones used in [14].

Moreover, we would like to point out that using IPC to evaluate the fault-tolerant SMT performance is appropriate. In the conventional multi-threading execution mode, threads are typically independent, thus, multiple tasks could be accomplished simultaneously. On the

⁵Note that we are simulating the performance overhead in fault-free cases.

Table 1. Simulated FT-SMT configurations

Component	Configuration
Processor	2-way SMT, 8-way out-of-order issue, 32-entry IFQ, 256-entry trace queue (256-entry IFQ when simulating superscalar), 64-entry issue queue, 128-entry ROB, 64-entry load queue, 64-entry store queue, 64-entry LVQ, 256-entry chkQ
FUs	6 Int ALU, 2 Int Multiply, 4 FP Add, 2 FP Multiply
Branch predictor	Hybrid: 8K-entry bimodal, 8K-entry gshare, 8K-entry 2-bit selector, 16-entry RAS, 4-way 1K BTB, 10-cycle misprediction penalty
L1 I- and D-cache	64KB, 32-byte block size, 4-way associative, 1-cycle hit latency
Unified L2 cache	1MB, 64-byte block size, 4-way associative, 6-cycle hit latency
Main mem	100-cycle latency

contrary, in the fault-tolerant multi-threading execution mode, two copies of the same thread actually carry out only single computation task. In another word, from the user’s viewpoint, two copies of the same thread function as if only one thread was executing. In this sense, the fault-tolerant multi-threading mode is functionally comparable to the superscalar model where IPC is conventionally used to evaluate the performance. It should also be emphasized that, in our fault-tolerant multi-threading simulations, we indeed compute IPCs as the number of *retired and then result-compared* instructions, counted from *only* the leading thread, divided by the total number of simulated clock cycles.

The simulation results are shown in Figure 4. Compared with the basic superscalar model (without fault-tolerance features), the performance overhead of our design is lowered to only 12% on average whereas an SRT-like design has 17% overhead. Further, if we choose the baseline as the two-way SMT without any fault-tolerance features (the baseline SMT processor is executing two identical threads), the performance overhead is 34% on average whereas the overhead of SRT-like design is 42%.

4.2 Hardware overhead evaluation

To evaluate how the size of traceQ and chkQ affects the fault-tolerant performance overhead, based on the performance simulators we developed, we varied the size of the queue and then observed the pipeline throughput to determine the design points.

4.2.1 Sizing traceQ

TraceQ functions as a buffer holding TT instructions to support the staggered execution mode. To be sufficient

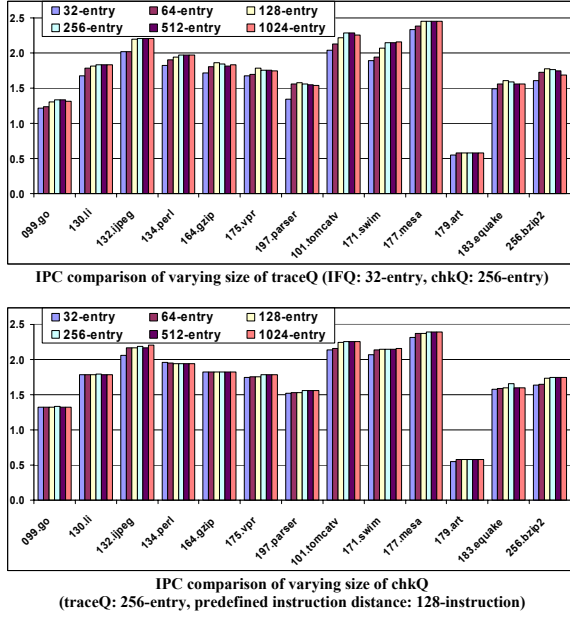


Figure 5. Hardware overhead evaluation

for that support, however, Equation 1 requires that the minimum size of traceQ must be higher than IFQ size plus the predefined instruction distance. Consequently, when simulations set traceQ size as 32-entry, the staggered execution mode is not supported there. However, when the traceQ is 64-entry or 128-entry, we use 32-instruction as the predefined instruction distance in order to enable staggered execution mode. Likewise, with traceQ larger than 256-entry, 128-instruction distance is used. Figure 5 shows that when traceQ is small in size, the benefit of the staggered execution mode cannot be fully exploited. IPCs are indeed degraded: compared with a 256-entry traceQ, IPCs in smaller traceQ cases are lowered by 9% on average. On the other hand, continually increasing traceQ, the IPC improvement is limited: on average, no significant IPC improvement is observed. Given these, we choose traceQ with 256-entry size as the design point. Furthermore, each entry of traceQ stores an instruction word (32-bit as typical size), a thread ID (1-bit), and branch resolve status bit (1-bit). Hence, the storage core of traceQ is : $256 \times (32+1+1) = 1088$ bytes.

4.2.2 Sizing chkQ

Figure 5 also shows the simulation results when we vary the size of chkQ. Given traceQ size is 256-entry and the predefined instruction distance is 128-instruction, we can find out that increasing chkQ size has diminishing throughput return: beyond a 64-entry chkQ, no significant performance improvement can be observed. The reason

for that is because chkQ is buffering those LT instructions which have retired however are waiting for comparison. Because of being reduced, TT can quickly catch up LT after dispatch stage and then promptly complete chkQ comparisons. Hence, chkQ is not frequently full such that we can choose a chkQ with only 64-entry. Furthermore, each entry of chkQ holds sequential numbers (10-bit, for instance), instruction word (32-bit), execution results (32-bit, for instance), and status bit (1-bit). Hence, the storage core of chkQ is: $64 \times (10 + 32 + 32 + 1) = 600$ bytes.

5 Related work on SMT transient fault-tolerant designs

The possibility of using the inherent redundancy included in multi-threading architectures to implement fault-tolerant features has been examined by a number of researchers [8, 13, 7, 18]. AR-SMT [15] uses SMT to execute two copies of the same program for fault-tolerance reasons. The original thread, called Active-stream, goes through the processor as it would normally. When it commits, its results are also enqueued into the delay buffer for Redundant-stream to perform the fault checking. The R-stream committed results are compared with those in the delay buffer. A fault is detected if the comparison fails, and furthermore, the committed state of R-stream can be used as a checkpoint for recovery. Simulations show that AR-SMT increases the execution time by only 10% to 30% over that of a single thread. However, AR-SMT requires twice as much memory space: one half for each stream since committed store instructions of A-stream will corrupt memory if any one of them is faulty. This is obviously expensive.

Simultaneous and Redundantly Threaded (SRT) processor proposed by Reinhardt, *et al.* [14] also compares two redundant threads execution results for fault detection. Leading thread (LT) stores branch outcomes into the branch outcome queue (BOQ) while Trailing Thread (TT) just consumes BOQ when encountering branches. Load Value Queue (LVQ) is implemented to guarantee that the two threads have the same input data. SRT is an improvement over AR-SMT because of two optimizations: (i) the slack fetch scheme; and (ii) the fault checking scheme: it only checks store instructions when the SoR includes the register files. Simulations show that when an SMT processor simultaneously runs two copies of the same program but without fault detection features, it is on average 32% slower than one which is only running one thread [9]. Additionally, a follow-up work [9] explores design options of Alpha 21464 processors for fault detection via multi-threading.

By not allowing any leading thread instructions to commit before fault checking, the Simultaneously and Redundantly Threaded processor with Recovery (SRTR)

proposed by Vijaykumar, *et al.* [20] relies on traditional precise interrupts mechanisms to rollback processor states, such that re-execution from a detected faulty instruction is possible. Moreover, SRTR performs within 1% and 7% of SRT for SPEC95 integer and floating-point programs, respectively.

6 Conclusions

The very concept of SMT provides the redundancy to detect faults. Therefore, we can run two copies of the same thread on top of SMT platforms in order to detect and correct soft errors. This paper has focused on two crucial implementation issues introduced by this scheme: (i) the design trade-off between the fault detection coverage versus design costs; (ii) the possible occurrence of deadlock situations. To achieve the largest possible fault detection coverage, we replicate the instructions fetched to generate the redundant thread copies. Further, we apply the SMT thread scheduling at the level of the instruction dispatch stage to lower the performance overhead. As a result, when compared to the baseline processor, our simulation results show that by using our two new schemes, the performance overhead can be reduced down to 34% on average, down from 42%. Finally, in the fault-tolerant execution mode, since the two copied threads are cooperating with each other, deadlock situations could be quite common. We thus demonstrated a detailed deadlock analysis and then concluded that allocating some entries of ROB, LQ, and SQ for the trailing thread is sufficient to prevent such deadlocks.

References

- [1] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama. A 1.3-GHz Fifth-Generation SPARC64 Microprocessor. *IEEE Journal of Solid-State Circuits*, November 2003.
- [2] D. C. Bossen, J. M. Tendler, and K. Reick. Power4 System Design for High Reliability. *IEEE Micro*, March-April 2002.
- [3] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, University of Wisconsin-Madison Computer Sciences Department, June 1997.
- [4] Compaq Computer Co., Massachusetts. *Alpha 21264/EV68CB and 21264/EV68DC Hardware Reference Manual*, 1.1 edition, June 2001.
- [5] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., Third edition, 2002.
- [6] D. Kang and J.-L. Gaudiot. Speculation Control for Simultaneous Multithreading. In *18th International Parallel and Distributed Processing Symposium*, April 2004.
- [7] S.-C. Lai, S.-L. Lu, and J.-K. Peir. Ditto Processor. In *International Conference on Dependable Systems and Networks*, June 2002.
- [8] A. Mendelson and N. Suri. Designing High-Performance & Reliable Superscalar Architectures: The Out of Order Reliable Superscalar (O3RS) Approach. In *International Conference on Dependable Systems and Networks*, June 2000.
- [9] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *29th International Symposium on Computer Architecture*, June 2002.
- [10] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *36th International Symposium on Microarchitecture*, December 2003.
- [11] N. Quach. High Availability and Reliability in the Itanium Processor. *IEEE Micro*, September-October 2000.
- [12] S. E. Raasch and S. K. Reinhardt. The Impact of Resource Partitioning on SMT Processors. In *12th International Conference on Parallel Architectures and Compilation Techniques*, September 2003.
- [13] J. Ray, J. C. Hoe, and B. Falsafi. Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery. In *34th International Symposium on Microarchitecture*, December 2001.
- [14] S. K. Reinhardt and S. S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *27th International Symposium on Computer Architecture*, June 2000.
- [15] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *29th International Symposium on Fault-Tolerant Computing*, June 1999.
- [16] N. R. Saxena and E. J. McCluskey. Dependable Adaptive Computing Systems - The ROAR Project. In *1998 IEEE International Conference on Systems, Man and Cybernetics*, October 1998.
- [17] A. Silberschatz, P. B. Galvin, and G. Gagne. *Applied Operating System Concepts*. John Wiley & Sons, Inc., First edition, 2000.
- [18] J. C. Smolens, J. Kim, J. C. Hoe, and B. Falsafi. Efficient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitectures. In *37th International Symposium on Microarchitecture*, December 2004.
- [19] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *22nd International Symposium on Computer Architecture*, June 1995.
- [20] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-Fault Recovery Using Simultaneous Multithreading. In *29th International Symposium on Computer Architecture*, June 2002.