Efficient Transient-Fault Tolerance for Multithreaded Processors Using Dual-Thread Execution

Yi Ma and Huiyang Zhou
School of Electrical Engineering and Computer Science
University of Central Florida
{yma, zhou}@cs.ucf.edu

Abstract—**Reliability becomes a key issue in computer system design as microprocessors are increasingly susceptible to transient faults. Many previously proposed schemes exploit simultaneous multithreaded (SMT) architectures to achieve transient-fault tolerance by running a program concurrently on two threads, a main thread and a redundant checker thread. Such schemes however often incur high performance overheads due to resource contention and redundancy checking.**

In this paper, we propose dual-thread execution (DTE) for SMT processors to efficiently achieve transient-fault tolerance. DTE is derived from the recently proposed fault-tolerant dual-core execution (FTDCE) paradigm, in which two processor cores on a single chip perform redundant execution to improve both reliability and performance. In this paper, we apply the same principles as in FTDCE to SMT architectures and explore fetch policies to address the critical resource-sharing issue in SMT architectures. Our experimental results show that DTE achieves an average of 56.1% speedup over the previously proposed simultaneously and redundantly threaded processor with recovery (SRTR). More impressively, even compared to single-thread execution, DTE achieves full-coverage transient-fault tolerance along with an average of 15.5% performance improvement.

Index Terms—**Fault tolerance, microprocessors, multi-threaded architectures, redundant systems.**

I. Introduction

CURRENT trend in semiconductor technology features faster transistors, higher integration density, and lower supply voltages. While this trend improves processor performance and reduces transistor-power consumption, it results in increased susceptibility to transient faults for modern microprocessors.

To protect program execution against transient faults, one promising approach is redundant execution using leader/follower architectures [1],[7],[11],[12],[13],[14],[15],[18],[21], in which a program is redundantly executed by a leading thread and a trailing thread. The execution results from the two threads are compared to detect transient faults. If the execution results are committed only after redundancy checking, fault tolerance can also be achieved by rewinding both threads to the most recently committed execution state when an error is detected. The leading thread and the trailing thread may run on different processor cores in chip multiprocessors (CMPs) [1],[7],[15] or share a single processor with simultaneous multithreading (SMT) [12],[13],[18].

In most leader/follower schemes for fault tolerance, the leader is the main thread while the follower serves as a checker thread providing redundant execution. Although the leading thread results, such as load values and branch outcomes, can be exploited to speedup the trailing checker thread, the overall execution may still suffer from significant performance overheads. The main reasons include (a) delayed instruction commitment in the main thread as instructions such as stores can only commit after redundancy checking, which results in the increased pressure on critical resources like the load/store queue (LSQ); and (b) resource contention between the two threads, especially when they share a SMT processor. In this paper, we propose a new scheme, named dual-thread execution (DTE), to efficiently achieve transient-fault tolerance for SMT architectures.

DTE is derived from the recently proposed fault-tolerant dual-core execution (FTDCE) paradigm [21]. In FTDCE, a program is executed by two processor cores on a single chip, the front processor and the back processor. The front processor speculatively executes instructions by invalidating long-latency cache-missing loads and their dependent instructions. The back processor re-executes the instructions retired by the front processor to ensure correctness and provide redundancy checking. In this way, both significant performance enhancement and fault tolerance can be achieved simultaneously (a more detailed background on FTDCE is described in Section 2.2). In this paper, we extend the same principles of FTDCE to SMT architectures. In other words, in DTE, a program is executed redundantly by a front thread and a back thread on a SMT processor and the threads process instructions in a similar way to the front and back processors in FTDCE.

As DTE is built upon SMT architectures, a key issue to address in DTE is resource contention between the two threads. In this paper, we propose and evaluate various resource sharing policies. Our experiments show that with a carefully designed policy, DTE achieves an average 56.1% speedup over the previously proposed Simultaneously and Redundantly Threaded processor with Recovery (SRTR) [18]. Compared to single-thread execution, DTE achieves both transient-fault

1-4244-9707-X/06/$20.00 ©2006 IEEE

tolerance and performance improvement (up to 91% and 15.5% on average) at the same time.

The remainder of the paper is organized as follows. Section 2 discusses related work and reviews FTDCE as the background for DTE. Implementation of DTE including the resource sharing policies is addressed in Section 3. Section 4 presents the simulation methodology and Section 5 discusses the experimental results. Section 6 summaries the paper.

II. Related Work and Background

A. Related Work

AR-SMT [13] is one of the first leader/follower schemes to exploit SMT architectures for fault tolerance. The leading stream (A-stream) executes the program, and conveys its results to the redundant stream (R-stream) through a delay buffer. The results from the A-stream provide perfect control and data flow prediction to speedup the R-stream. Validation of the predictions from the A-stream in the R-stream is used to detect potential transient faults. Slipstream processors [15] extend the AR-SMT concepts for higher performance efficiency. In slipstream processors, the leading A-stream runs a distilled version of the program by removing ineffectual instructions while the R-stream serves as the main thread re-executing the complete program. The A-stream results are used as both predictions and redundancy checking for the R-stream. To our knowledge, besides fault-tolerant dual-core execution (FTDCE) [21], slipstream processors are the only other scheme that can achieve both performance improvement and fault tolerance simultaneously. Compared to slipstream processing, FTDCE as well as the proposed DTE is more complexity effective and achieves full redundancy coverage along with higher performance improvement [21].

In DIVA [1], a separate in-order checker is used to validate out-of-order execution of the main processor. Although DIVA takes advantage of the leading main processor results to simplify the checker design, delayed instruction commitment incurs performance overheads. In SHREC [14], the checker shares a pool of functional units with the main processor to further reduce the resource overhead of the DIVA design. However, it suffers from the same delayed instruction commitment problem as in DIVA.

In SRT [12], a program is executed by two threads on SMT architectures, a leading thread and a trailing thread similar to the AR-SMT scheme. The leading thread executes instructions and forwards the values of load instructions and branch outcomes to the trailing thread through a load value queue (LVQ) and a branch outcome queue (BOQ) when committing the instructions. The retired store instructions are kept in a store buffer (StB) until they have been verified by the trailing thread. SRTR [18] extends SRT to support fault recovery. In SRTR, all instructions of the leading thread are not allowed to commit until they are verified by the trailing thread. The two threads communicate values for redundancy checking through the LVQ, BOQ, StB, and register value queue (RVQ). CRT [11] provides transient-fault detection for CMPs, and CRTR [7] extends CRT to provide fault recovery. All these schemes, however, suffer from performance overheads as discussed in Section 1. In [9], the execution results from the leading thread is further utilized to reduce the resource requirements of the trailing thread in SMT processors.

All the above-mentioned fault-tolerance schemes except slipstream processors trade performance for system robustness. In comparison, a recently proposed fault tolerant architecture named fault-tolerant dual-core execution (FTDCE) [21] achieves both transient-fault tolerance and significant performance improvement simultaneously for CMPs. As DTE is derived from FTDCE, next we review FTDCE as the background for DTE.

B. Background: Fault-Tolerant Dual-Core Execution (FTDCE)

FTDCE is built upon two superscalar processors (called the front core and the back core) coupled with a hardware FIFO queue (called the result queue), as show in Fig. 1. The front and back processors have separate L1 caches while share a unified L2 cache.

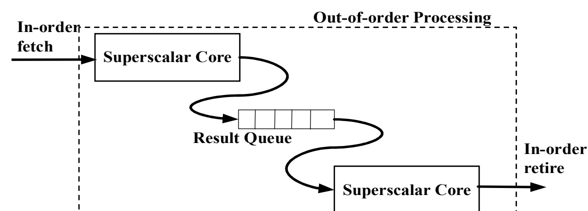

Fig. 1, An overview of Fault-Tolerant Dual-Core Execution

How it works -- In FTDCE, the front processor executes instructions in its normal way expect for long-latency cache misses (i.e., L2 misses). When the front processor detects any L2 miss load, it uses an invalid (INV) value to substitute the data that are being fetched from memory, similar to runahead execution [6],[10]. The INV flag is propagated through register data dependency and memory data dependency to invalidate the dependant instructions of the cache-missing loads.

Instructions except stores and those raising exceptions are retired by the front processor in its normal way. When a store instruction retires, it does not update data cache. Instead, it updates a structure called run-ahead cache to communicate the store value to subsequent loads in the front processor. Exception handling is disabled as the back processor maintains the precise execution state. The retired instructions with their execution results from the front processor are forwarded into the result queue.

The back processor fetches instructions from the result queue, re-executes all the instructions to ensure correctness, and performs redundancy checking by comparing its execution results with those carried from the front processor. When a discrepancy is detected, it is simply treated as a branch

misprediction and is recovered by rewinding both processors to the currently committed state, i.e., squashing all the instructions in the back processor, the results queue, and the front processor, invalidating the run-ahead cache, and copying the back processor's current architectural states to the front processor.

Since only the instructions that are not invalidated by the front processor have valid results to be checked against transient faults, those invalidated instructions are not under protection. To achieve full redundancy coverage, the back processor fetches those invalidated instructions twice, one for normal execution and the other for redundancy check, using a simple renaming scheme as described in [21].

How it improves performance -- By invalidating long-latency cache-missing loads, the front processor runs with a virtually ideal L2. As a result, it runs very fast and far ahead of the back processor. The cache misses in the front processor then become prefetches for the back processor. In addition, the front processor promptly resolves the branch mispredictions that are independent on those invalidated instructions, which helps the back processor reduce the time wasted on wrong paths. Overall, as shown in [20], the collaboration between the two processors forms a very large instruction window and effectively hides memory access latencies. The execution paradigm of FTDCE is similar to Flea-Flicker two-pass pipelining [2]. The key difference is that the execution results of the run-ahead pipeline are reused in the Flea-Flicker design while FTDCE relies on re-execution to relieve the correctness requirement of the front core. The re-execution eliminates the complexity associated with centralized memory order bookkeeping and enables redundancy check for fault tolerance.

How it achieves transient-fault tolerance – In FTDCE, every instruction is redundantly executed and the results are checked before committing to the ECC protected architectural states. Any discrepancy due to soft errors will be transparently repaired by rewinding both processors to the currently committed state using the existing mispeculation-recovery scheme.

III. Implementation of Dual-Thread Execution

DTE can be viewed as an SMT implementation of the FTDCE concept with the two separate processor cores being replaced by two threads, namely *the front thread* and *the back thread*, on a single processor. In Section 3.1, we describe the hardware changes required on a typical SMT processor to support DTE. In Section 3.2, we deal with the inherent resource-contention issue with SMT, and explore fetch policies to judiciously allocate resources between the front thread and the back thread.

A. DTE Architectures

To support DTE, there are four hardware changes that we need to make on a typical SMT processor, as highlighted in Fig. 2.

First, we need to add a run-ahead cache and a result queue. The run-ahead cache is only accessed by the front thread. All the committed stores in the front thread write the data to run-ahead cache instead of D-cache, the same way as in FTDCE. When a block is replaced from the run-ahead cache, it is simply dropped without being written to L1 D-cache. The front thread retires instructions with the execution results to the result queue.

Second, we need to include INV bits in the register file (RF) and the load-store queue (LSQ) for instruction invalidation. The front thread executes instructions in the same way as the front processor in FTDCE, i.e., a long-latency cache-missing load is invalidated by setting the INV flag of the destination register(s). Dependent instructions of those invalidated long-latency loads are invalidated through INV propagation, except branches and stores. If a branch instruction uses an INV register, its prediction will be used as the resolved branch target. A store instruction becomes a nop if its address is invalid. If the value of a store instruction is invalid, the corresponding LSQ entry's INV bit is set. The store-load forwarding will then propagate the INV bit to the subsequent load(s) accessing the same address. When a store with an invalid value retires, it sets the INV bit of the byte(s) in the run-ahead cache.

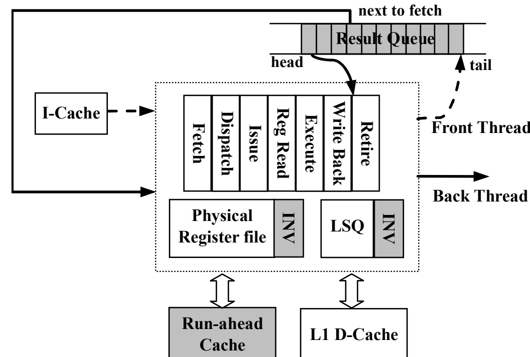

Fig. 2, DTE architecture

Third, we need to change the fetching and renaming mechanism of the back thread. The back thread fetches instructions from the result queue rather than the instruction cache. An invalidated instruction is also duplicated when it enters the instruction fetch buffer. The duplicated instructions will access an additional rename map table as described in [21] so as not to interfere with normal execution. Upon completion, all instructions have to be checked before being retired from the back thread by comparing the results with either the execution results carried from the front thread or the redundant results computed at the back thread.

Fourth, we need to include a mispeculation/fault recovery mechanism for the back thread. When the back thread detects a branch misprediction or a redundancy checking failure, all the instructions in the front thread, the back thread, and the result queue are squashed, the run-ahead cache is invalidated, and the architectural register file of the back thread is copied to the front thread.

Similar to FTDCE, DTE achieves fault tolerance through redundant execution. The normal execution results in the back thread are used to compare with the redundant results, which are either carried from the front thread or produced by the back thread itself, to detect transient faults during instruction execution. Here, we assume that the architectural state of the processor, including the PC, the architectural register file, as well as the memory hierarchy are protected with information redundancy techniques such as ECC. Then, when a discrepancy is detected, both threads are rewound to the current architectural state using the existing mispeculation recovery mechanism.

Besides transient-fault tolerance, DTE also has the potential to achieve performance enhancement as the two threads form a large instruction window to hide memory access latencies, the same as FTDCE.

B. Fetch Policies for DTE

In DTE, both the front thread and the back thread share a single out-of-order (OOO) execution core and the cache hierarchy. Compared to FTDCE, such close integration offers more efficient inter-thread communication (i.e., lower cost of mispeculation recovery at the back thread) while incurring the potential resource-contention problem. Therefore, efficient resource sharing is the key to overall performance. Next, we examine various resource-sharing policies and discuss their pros and cons for DTE.

ROUND-ROBIN (RR) fetch policy – This scheduling policy is commonly used because of its fairness in allocating resources to individual threads. As discussed in [17], the basic RR policy can be extended to overcome the fetch-block fragmentation and thread shortage problem. In other words, in each cycle, the selected thread is allowed to use up all the fetch bandwidth. If it does not have enough instructions, the remaining bandwidth can be used by the other thread. This fetch bandwidth sharing mechanism is used in all the scheduling policies presented in this paper.

The RR policy is relatively easy to implement but fails to consider the resource requirement of each thread. An L2-cache miss in one thread, for example, can stall the entire pipeline by reserving most/all of the available resources including the issue queue, LSQ, the re-order buffer (ROB), and free physical registers. The other thread is thereby blocked although there may exist abundant instruction-level parallelism (ILP).

ICOUNT fetch policy – In order to improve resource utilization, this scheduling policy gives higher priority to the thread with fewer instructions in the decode, rename, and issue stages [17]. ICOUNT shows good results for high ILP threads while still suffers from the resource over-allocation problem due to L2-cache misses. Further improvements upon the ICOUNT policy, such as STALL [16], FLUSH [16], and FLUSH++ [5], are proposed to address this problem by either stalling or flushing (i.e., de-allocating resource) the offending thread (i.e., the thread having the L2-cache miss). Those approaches including ICOUNT, however, do not fit well with DTE since only the back thread has L2-cache misses. As the front thread already runs much faster with the virtually ideal L2 cache, granting more resource to the front thread simply reduces the resource available to the slower back thread. As the front thread will be stalled when the result queue is full, slowing down the back thread usually undermines the overall performance.

SLACK fetch policy – This policy is used to improve the performance of SRT [12] and SRTR [18]. It aims to maintain a target slack (i.e., a fixed number of instructions), between the leading thread and the trailing thread. The fetch unit always starts fetching from the leading thread until the target slack is reached. Then, the ICOUNT policy is used to direct the fetch unit. SLACK works well with SRT and SRTR since the performance bottleneck in these schemes lies in the leading thread (i.e., the main thread). In contrast, the trailing thread (i.e., the back thread) in DTE is the main thread and usually runs slower. Therefore, SLACK is less effective for DTE as it tends to give higher priority to the front thread.

Back-First (BF) fetch policy – Since the back thread is often the performance bottleneck in DTE, this scheduling policy always gives higher priority to the back thread. Doing so, however, presents a potential problem that the front thread could be starved for resources and fails to run sufficiently far ahead to warm the caches and fix branch mispredictions for the back thread.

Queue-Occupancy (QO) fetch policy – This scheduling policy solves the problem with the BF policy and it is based on the observation that the occupancy of the result queue indicates the difference in execution speeds between the two threads. A full result queue suggests that the front thread runs too fast while an empty queue shows that the back thread is the one using up too much resource. Therefore, in this scheduling policy, we try to keep the occupancy around 50% (i.e., the number of instructions in the result queue is half of the queue size) using a simple threshold mechanism. If the result queue occupancy is lower than 50%, the front thread gets higher priority. Otherwise, the back thread is the one to be fetched first.

In order to avoid the scenario where one thread monopolies all the resources, a static threshold is also introduced in the abovementioned scheduling polices upon the maximum resources, such as ROB and issue queue entries, that can be allocated to either thread. In this paper, the maximum is set as 120 for a 128-entry instruction window processor.

IV. Simulation Methodology

Our simulation environment is developed from the SimpleScalar [4] toolset while our execution-driven timing simulator is completely rebuilt to model MIPS R10000 architecture with SMT support. The cache modules in our simulator model both data and tag stores. Wrong-path events are also faithfully simulated.

The processor configuration is listed in Table I. For DTE, the default result queue size is 512 entries and we assume 1-cycle delay of the result queue. The run-ahead cache is 4kB, 4-way associative with a block size of 8 bytes. A latency of 8 cycles is assumed for copying the architectural register values from the back thread to the front thread. The Queue-Occupancy (QO) policy is the default fetch policy for DTE and we examine impact of fetch policies in Section 5.2.

The same set of SPEC2000 benchmarks and the simulation points are used as in [20].

TABLE I. Configuration of the processor

Pipeline	3-cycle fetch stage, 3-cycle dispatch (decode and dispatch) stage, 1-cycle issue stage, 1-cycle register access stage, 1-cycle retire stage. Min. branch misprediction penalty = 9 cycles
Instruction Cache	Size=32 kB; Assoc.=2-way; Repl.=LRU; Line size=16 instructions; Miss penalty=*10* cycles.
Data Cache	Size=32 kB; Assoc.=2-way; Repl.=LRU; Line size = 64 bytes; Miss penalty=10 cycles.
Unified L2 Cache	Size=1024 kB; Assoc.=8-way; Repl.=LRU; Line size=128 bytes; Miss penalty=300 cycles.
Br. Predictor	64k-entry G-share; 32k-entry BTB
Superscalar Core	Reorder buffer: 128 entries; Dispatch/issue/retire bandwidth: 8-way superscalar; 8 fully-symmetric function units; Data cache ports: 8. Issue queue: 128 entries. LSQ: 128 entries. Rename map table checkpoints: 32
Execution Latencies	Address generation: *1* cycle; Memory access: *2* cycles (hit in data cache); Integer ALU ops = 1 cycle; Complex ops = MIPS R10000 latencies
Memory Disambiguation	Perfect memory disambiguation
HW prefetcher	Stride-based stream buffer prefetch

V. Experimental Results

A. Performance Impact of DTE

As discussed in Section 3, DTE has the potential to achieve both transient-fault tolerance and performance enhancement at the same time. In this experiment, we examine the performance impact of DTE as shown in Fig. 3. For comparison, we also include the results of SRTR, a previously proposed transient-fault tolerance scheme for SMT processors. In SRTR, the leading thread is the main thread and the trailing thread is the checker thread. The SLACK policy is used in SRTR with the target slack set as 64 since it achieves the best performance among the slacks of 32, 64 and 80. The execution time reported in Fig. 3 is normalized to single-thread execution (i.e., the single thread running on a SMT processor without redundancy checking).

From Fig. 3, it can be seen that DTE achieves significant performance improvement over single-thread execution for the benchmarks *mcf*, *art*, *equake*, and *swim*. These benchmarks are memory-intensive workloads featuring a large number of L2-cache misses. Effective prefetching of the front thread in DTE reduces the number of cache misses and enables more computation overlapping in the back thread. DTE achieves similar performance to single-thread execution for *parser*, *twolf*, *vpr*, and *ammp*, suggesting that the benefits from the front thread prefetching are offset by additional redundancy checking in the back thread for these benchmarks. For computation-intensive benchmarks, *gap* and *bzip2*, DTE is less effective as there are too few L2-cache misses being invalidated to make a difference. On average, DTE achieves transient-fault tolerance along with an average of 15.5% speedup over single-thread execution. Compared to SRTR, DTE outperforms it on every benchmark, with speedups up to 144% (swim) and an average of 56.1%. The reason is that DTE eliminates the problem of delayed instruction commitment (since the front thread does not wait for redundancy checking) and builds a much larger instruction window to hide memory access latencies.

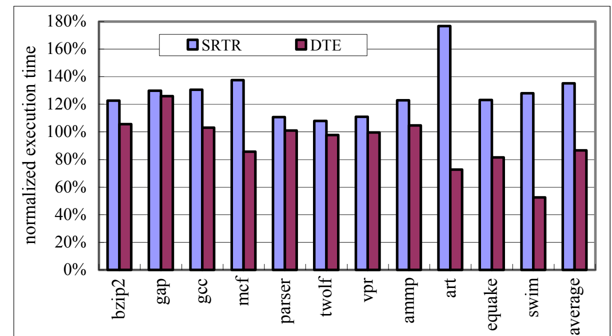

Fig. 3, Normalized execution time relative to single-thread execution.

B. Exploring Fetch Policies

In this experiment, we examine the impact of different fetch policies for DTE addressed in Section 3.2 and the results shown in Fig. 4 are the execution time normalized to single-thread execution.

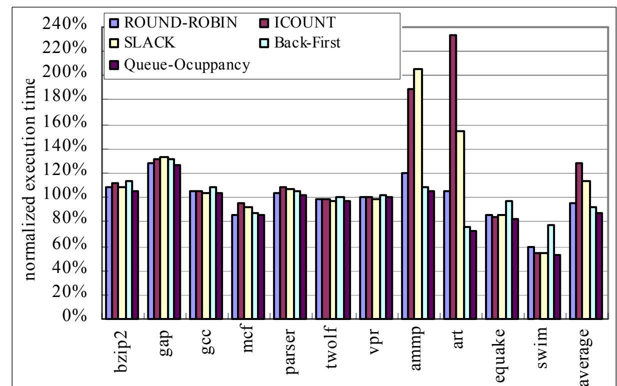

Fig. 4, Performance impacts of fetch policies for DTE.

Among various fetch policies used for DTE, ROUND-ROBIN works reasonably well and achieves an average of 4.7% speedup over single-thread execution. ICOUNT favors the front thread since all its L2 cache misses are invalidated. As discussed in Section 3.2, ICOUNT does not fit well with DTE and it results in a 28.4% performance overhead on average. SLACK has the similar problem to the ICOUNT policy. Both ICOUNT and SLACK incur pathologic behavior for the benchmarks *ammp* and *art* as the front thread consumes too much resource. Such pathologic behavior can be easily avoided by reducing the maximum amount of resource that can be allocated for the front thread.

By prioritizing the back thread, the Back-First (BF) policy solves the problem with ICOUNT and SLACK and achieves an average of 8.6% speedup over single-thread execution. For the benchmarks *equake* and *swim*, however, BF performs much worse than all the other fetch policies. The reason lies in the slow progress of the front thread as most resources are occupied by the back thread. Therefore, the front thread fails to run sufficiently ahead to warm up the caches for the back thread. The Queue-Occupancy (QO) fetch policy effectively overcomes these problems and it achieves the highest performance among all the fetch policies studied in this experiment.

C. *Impact of Pipeline Bandwidth*

In this experiment, we vary the processor pipeline bandwidth to study how well DTE works with different resource limitations. Fig. 5 shows the average IPC (Harmonic mean) of all 11 benchmarks for 4-way, 6-way, and 8-way superscalar processors with SMT. In both DTE and SRTR, the IPCs are calculated based on the number of instructions retired by the main thread, i.e., the redundant instructions are not included. QO is used for DTE and SLACK is used for SRTR in this experiment.

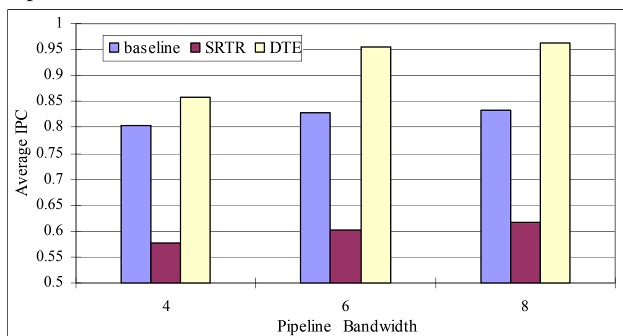

Fig. 5, Performance impact of pipeline bandwidths.

As shown in Fig. 5, DTE achieves 7.1%, 15.3%, and 15.5% speedup over single-thread execution on average for pipeline bandwidth as 4, 6, and 8, respectively. In comparison, SRTR incurs 28%, 27.2%, and 26% performance overhead in order to achieve transient-fault tolerance. DTE utilizes the increased pipeline bandwidth more effectively than SRTR and single thread. The reason is that the memory wall problem limits resource utilization in SRTR and single-thread execution while the large instruction window formed with DTE effectively hides memory access latencies. Furthermore, the higher pipeline bandwidth, the less resource contention exists between the front thread and the back thread in DTE.

Another observation from Fig. 5 is that redundant execution in DTE does not introduce too much pressure on pipeline resources. DTE running on a 6-way superscalar achieves similar performance to DTE running on an 8-way superscalar processor, suggesting that resource shortage does not present a problem with DTE when pipeline bandwidth is beyond 6.

D. *Energy Efficiency of DTE*

To evaluate energy efficiency of DTE, we imported WATTCH [3] and Hotleakage [19] into our simulator to examine both dynamic and static energy consumption. In this experiment, we use the 70nm technology with a clock frequency of 5.6GHz and assume linear clock gating [3]. Fig. 6 shows the normalized energy consumption relative to single-thread execution for both DTE and SRTR. For DTE results, we also incorporated the power consumption of the result queue and the run-ahead cache. Compared to SRTR, DTE consumes more dynamic energy since the back thread in DTE needs to duplicate the instructions that are invalidated by the front thread. The advantage of DTE lies in its highly reduced execution time, which results in much less static/leakage energy consumption even compared to single-thread execution. With current trend of technology scaling, static/leakage energy will become more dominant in overall energy consumption, which makes DTE a better fit for future technologies.

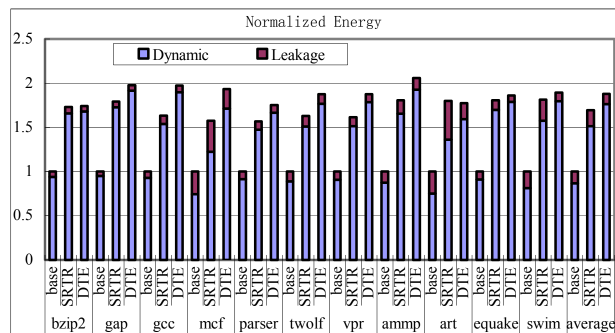

Fig. 6, Normalized energy consumption relative to single-thread execution.

In the next experiment, we use energy-delay product (EDP) [8] to evaluate energy efficiency of DTE and SRTR. Fig. 7 shows the normalized EDP relative to single-thread execution. Compared to SRTR, DTE achieves similar energy efficiency for the benchmarks, *gcc*, *parser*, *twolf*, *vpr*, and *ammp*. For remaining benchmarks, DTE has much better energy efficiency than SRTR with only one exception, *gap*, for which DTE achieves little performance improvement (2.8%) over SRTR while incurring an energy overhead of 10%. On average, DTE reports much higher energy efficiency than SRTR (1.63 vs. 2.29).

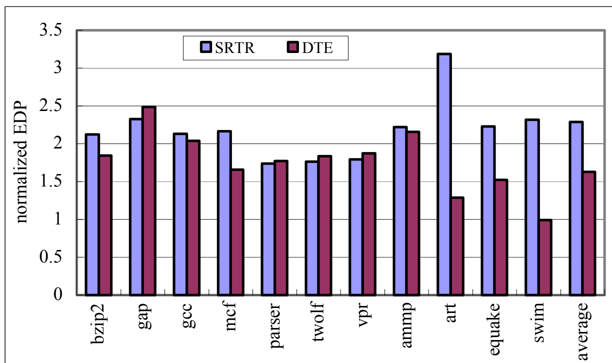

Fig. 7, Normalized Energy-Delay Product relative to single-thread execution.

VI. Conclusion

In this paper, we propose dual-thread execution (DTE) to achieve efficient transient-fault tolerance for SMT processors. DTE extends the recently proposed FTDCE, a transient-fault tolerance scheme for CMPs, to SMT architectures. In DTE, the front thread and the back thread execute the instruction stream collaboratively, not only providing redundancy check to protect against transient faults but also forming a large instruction window to hide memory access latencies. As DTE builds upon SMT processors, we propose and evaluate various fetch policies to address the critical resource contentions between the two threads. With the Queue-Occupancy fetch policy, our experimental results show that DTE achieves full-coverage transient-fault tolerance along with an average of 15.5% performance improvement over single-thread execution. Compared to a previously proposed transient-fault tolerance scheme for SMT processors, SRTR, DTE achieves both significantly higher performance and better energy efficiency.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable suggestions to improve the paper.

References

[1] T. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design", *Proc. of the 32nd Int. Symp. on Microarch. (MICRO-32)*, 1999.

[2] R. Barnes, E. Nystrom, J. Sias, S. Patel, N. Navarro, and W. Hwu, "Beating in-order stalls with flea-flicker two pass pipelining", *Proc. of the 36th Int. Symp. on Microarch. (MICRO-36)*, 2003.

[3] D. Brooks, V. Tiwari and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimization", *Proc. of the 27thInt. Symp. on Comp. Arch. (ISCA-27)*, 2000.

[4] D. Burger and T. Austin, "The SimpleScalar tool set, v2.0", *Computer Architecture News*, vol. 25, June 1997.

[5] F. Cazorla, E. Fernandez, A. Ramirez and M. Valero, "Improving memory latency aware fetch policies for SMT processors", *Proc. of the 5th Int. Symp. on High Perf. Computing (ISHPC-5)*, 2003.

[6] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss", *Proc. of the 1997 Int. Conf. on Supercomputing (ICS-97)*, 1997.

[7] M. Gomma, C. Scarbrough, T. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors", *Proc. of the 30th Int. Symp. on Comp. Arch. (ISCA-30)*, 2003.

[8] M. Horowitz, T. Indermaur, and R. Gonzalez, "Low power digital design", *Int. Symp. on Low Power Electronics*, 1994.

[9] S. Kumar and A. Aggarwal, "Reducing resource redundancy for concurrent error detection techniques in high performance microprocessors", *Proc. of the 12th Int. Symp. on High Perf. Comp. Arch. (HPCA-12)*, 2006.

[10] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors", *Proc. of the 9th Int. Symp. on High Perf. Comp. Arch. (HPCA-9)*, 2003

[11] S. Mukherjee, M. Kontz and S. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives", *Proc. of the 29th Int. Symp. on Comp. Arch.(ISCA-29)*, 2002.

[12] S. Reinhardt and S. Mukherjee, "Transient fault detection via simultaneous multithreading", *Proc. of the 27th Int. Symp. on Comp. Arch. (ISCA-27)*, 2000.

[13] E. Rotenberg, "AR-SMT: a microarchitectural approach to fault tolerance in microprocessors", *Proc. of the 29th Int. Symp. on Fault-Toler. Computing (FTCS-29)*, 1999.

[14] J. Smolens, J. Kim, J. Hoe and B. Falsafi, "Efficient Resource Sharing in Concurrent Error Detecting Supersalar Microarchitectures", *Proc. of the 37th Int. Symp. on Microarch. (MICRO-37)*, 2004.

[15] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: improving both performance and fault tolerance", *Proc. of the 9th Int. Conf. on Arch. Support for Prog. Lang. and Operating Sys. (ASPLOS-9)*, 2000.

[16] D. Tullsen and J. Brown, "Handling long-latency loads in a simultaneous multithreaded processor", *Proc. of the 34th Int. Symp. on Microarch. (MICRO-34)*, 2001.

[17] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo and R. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor", *Proc. of the 23rd Int. Symp. on Comp. Arch. (ISCA-23)*, 1996.

[18] T. Vijaykumar I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading", *Proc. of the 29th Int. Symp. on Comp. Arch. (ISCA-29)*, 2002.

[19] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, "Hotleakage: a temperature-aware model of sub-threshold and gate leakage for architects", Tech. Reports CS-2003-05, U. Va. Dept. of CS, 2003.

[20] H. Zhou, "Dual-core execution: building a highly scalable single-thread instruction window", *Proc. of the 2005 Int. Conf. on Para. Arch. And Compiler Tech, (PACT'05)*, 2005.

[21] H. Zhou, "A case for fault-tolerance and performance enhancement using Chip Multiprocessors", *IEEE Comp. Arch. Letters*. September 2005.