A Fault Detection Mechanism in a Data-flow Scheduled Multithreaded Processor

Jian Fu*†, Qiang Yang*†, Raphael Poss*, Chris R. Jesshope*, Chunyuan Zhang†

*Informatics Institute, University of Amsterdam, Amsterdam, The Netherlands

Email: {j.fu, q.yang, r.poss, c.r.jesshope}@uva.nl

†School of Computer, National University of Defense Technology, Changsha, China

Email: {fujian, cyzhang}@nudt.edu.cn

***Abstract*—This paper designs and implements the Redundant Multi-Threading (RMT) in a Data-flow scheduled Multi-Threaded (DMT) multicore processor, called *Data-flow scheduled Redundant Multi-Threading* (DRMT). Meanwhile, It presents *Asynchronous Output Comparison* (AOC) for RMT techniques to avoid fault detection related inter-core communication and alleviate the performance and hardware overheads induced by output comparison. Results show that the performance overhead of DRMT is less than 60% even when the number of threads is four times the number of processing elements. Also the performance and hardware overheads of AOC are insignificant.**

I. Introduction

As CMOS technology scales ever further, multicore processors are becoming mainstream, both in commercial and research fields. However, their vulnerability is increasing due to greater hardware complexity, smaller feature size, higher frequency and lower voltage. Therefore, fault tolerance techniques will need to be considered in all modern and future processors. RMT is a family of fault tolerance techniques in which two threads redundantly execute the same program and compare the results for fault detection. Once a mismatch is detected, a fault is flagged and a recovery process is started. RMT was developed after the invention of simultaneous multithreading (SMT) and makes the overhead of redundant execution relatively small as the additional threads increase the efficiency of SMT. RMT has now shifted to multicore due to the broader fault coverage and the fact that multicore is the mainstream due to the advances in silicon technology.

The key concepts of RMT are the sphere of replication, input replication and output comparison. Sphere of replication defines the protected scope of a physical system. Components within the sphere of replication are protected by redundant execution, while components outside the sphere may not be. It also identifies which input and output values are required for special handling. In other words, values that enter and exit the sphere are the inputs and outputs that require replication and comparison, respectively. In order to feed the corresponding operations in both master and redundant thread with the same data, input replication is needed. Otherwise, the threads may produce different outputs that will be detected and recovered as if a fault had occurred. Output comparison is the most important part in RMT and is responsible for fault detection. Usually, memory store and register update are two types of output required to be compared. However, the comparison is done instruction-by-instruction if register update is counted, which puts pressure on the latency of comparison, especially with inter-core communication in multicore processors. A fault in register update can be detected during the memory store eventually. So most multicore based RMT techniques only compare the memory store. Furthermore, memory stores are compressed as fingerprints in order to alleviate the inter-core communication, which trades fault coverage for performance.

978-3-9815370-2-4/DATE14/©2014 EDAA

This paper makes contributions in two areas of RMT. First, we implement RMT technique in a DMT multicore processor, i.e., DRMT. This poses significant challenges in the design and the resulting design is described and evaluated in detail. Results show that the performance overhead of redundant execution is less than 11%, 30% and 60% when the number of threads is 1, 2 and 4 times the number of processing elements respectively. Second, we design an AOC method for RMT, which eliminates inter-core communication in output comparison, and halves the buffer size compared to prior studies. Evaluation shows that the performance and hardware overheads of AOC are low.

II. DMT Overview

Commercially, there are two forms of multithreading supported in mainstream processor designs. The first is hyper-threading or SMT, which is used in Intel cores, where several threads drive superscalar instruction issue. The other is the Sun/Oracle Niagara series of multicore chips, where each pipeline supports several threads. In Niagara, instructions are interleaved in the pipeline from all threads until some event causes a thread to suspend.

Besides these, many other forms of multithreading are studied in academic field. DMT [1] is one of them, which is inspired by the idea of data-driven thread execution [2], [3]. The most important feature of DMT is that the thread suspends if an instruction fails to read data in one of its register operands and a continuation to the thread is stored in that register until the data is provided. The key mechanism supporting this is a synchronizing register file, i.e., all registers are implemented with read-after-write or data-flow synchronization. When the current thread suspends, one of the other active threads may be selected for processing in the pipeline. Hence, DMT provides support the efficient execution of long-latency operations.

The MGSim [4] is Alpha-based software simulator of DMT. Normally an instruction only knows whether all its operands are ready or not in the third stage of Alpha pipeline (i.e., read stage). If all operands of current instruction are not

available, the pipeline would need to flush all instructions that come from the same thread in the previous stages. Then, an active thread may be rescheduled in the pipeline if there are ready threads. This means that the context switch costs four cycles. In order to reduce the overhead of context switch, the pipeline needs to be aware of the dependent instructions to long latency operation as early as possible. So a codesign of compiler and microarchitecture is implemented in MGSim. The compiler flags instructions that consume data from long-latency operations. And the instruction fetch stage of pipeline will switch contexts if a flagged instruction is fetched. This method reduces the overhead of context switch to zero if there are enough active threads to be rescheduled.

III. DRMT Implementation

A. Basic Implementation

To provide broader fault coverage, DRMT runs two copies of a program on separate cores of a multicore processor. The sphere of replication in DRMT includes pipeline, register file, thread scheduler and L1 cache. DRMT adopts relaxed input replication in order to simplify the hardware, i.e., no special input replication operation is applied. Because fault recovery techniques can recover the divergence caused by input incoherence (if required) induced by relaxed input replication. Furthermore, the most important point is that we have the same observation with Reunion [5]: relaxed input replication provides the correct inputs in all cases in our evaluation even for shared memory multithreaded program.

It is possible in a multithreaded program that more threads are required than exist in the multithreaded processor. Hence, thread contexts will be reused when the multithreaded program is executed. So a master and redundant threads pairing mechanism is needed for efficient output comparison. We implement the thread pairing implicitly during thread creation in a quasi-lockstep way. The execution of master and redundant threads are independent. And the overhead of thread pairing can also be hidden by the multithreading technique, which is demonstrated in our evaluation.

B. Asynchronous Output Comparison

Unlike the previous multicore based RMT, we use a Comparison Buffer (CB), added between the L1 and secondary memory, and shared by a fixed-core pair, to buffer the unverified store values and compare them. Each output should be stored to both the L1 and the CB, then compared in the CB before being committed to secondary memory. The operation of data input is the same as previously: the data comes from the secondary memory to L1.

The CB comprises a number of sets, which are identified by the thread identifier. Matching master and redundant threads use the same set. Each set is a FIFO queue, as all the stores are committed to the CB in program order, no matter if the pipeline is out-of-order or in-order. Each entry in a set has four fields: address, value, mask and flag. The flag identifies the data as being stored by master or redundant thread.

When a set of the CB receives data, the data will be written to the set directly if the set is empty. Otherwise, it will check whether the data received and the head of the set come from the same thread. If so, the data will be appended to the end of the set. If they come from different threads, then the data will be compared. A fault is detected when the data does not match. If they match, the data will be popped from the set and written to secondary memory. For a given set of the CB, one of the master and redundant threads is the producer, and the other is the consumer. Hence if they run close enough, the lifetime of a store in the CB could be very short. Any read requests coming from the L1, will first search the set indexed by the current thread. Data will be returned if it is available, otherwise the read request will be sent to secondary memory as usual. Meanwhile, the CB forwards the other memory protocol related messages without any extra operations.

The challenge of this CB design is how to suspend a thread when it writes to a full set in the CB. As the CB is located between L1 and secondary memory, it is not feasible that the thread is suspended in the same cycle with the store retired from memory stage of the pipeline. In order to address this problem, we design an asynchronous collaboration between the pipeline and the CB, called AOC. In this solution, each thread has a counter for its committed store, which is initialized as the capacity of the set in the CB. The store counter will be decremented when a store is committed. If the store counter is decremented to 0 (i.e., the set of CB is full), the incoming store of current thread cannot be committed any more. The store will be treated as an instruction that lacks an operand, and all its predecessors will be flushed from the pipeline if they come from the same thread. Meanwhile, the current thread will be suspended. On the other side, the CB will send a message to the thread if its set pops an entry due to a successful comparison. When thread receives the message, it increases its store counter and reschedules itself if it is suspended on a full comparison set. Obviously, there may be unnecessary thread suspensions due to the latency of message transfer, however the architecture tolerates latency and rescheduling is performed in hardware so has little overhead.

IV. Evaluation

A. Methodology

Platform. We evaluate DRMT using MGSim [4], which is an open source discrete event many-core simulator for on-chip hardware components. The core in MGSim is equipped with data-flow thread scheduler. The concrete configuration of MGSim for the evaluation is shown in table I.

TABLE I: The configuration of MGSim

4 cores
1GHz 6-statge, single-issue, in-order Alpha pipeline
8 threads per core
128KB Private L1 Cache (split I/D), 4-way, LRU, 64B lines
1MB Shared L2 Cache, 4-way, LRU, 64B lines
1 DDR3-1600 channel

Benchmarks. We use 6 Livermore Loops [6] to evaluate the performance overhead of DRMT, see table II. These 6 loops are concurrent, i.e., have no loop-carried dependencies. In our evaluation, each loop has 10,000 iterations. And each loop has two versions: a multithreaded version and a single-threaded one. Each iteration of the loop is implemented as a thread in

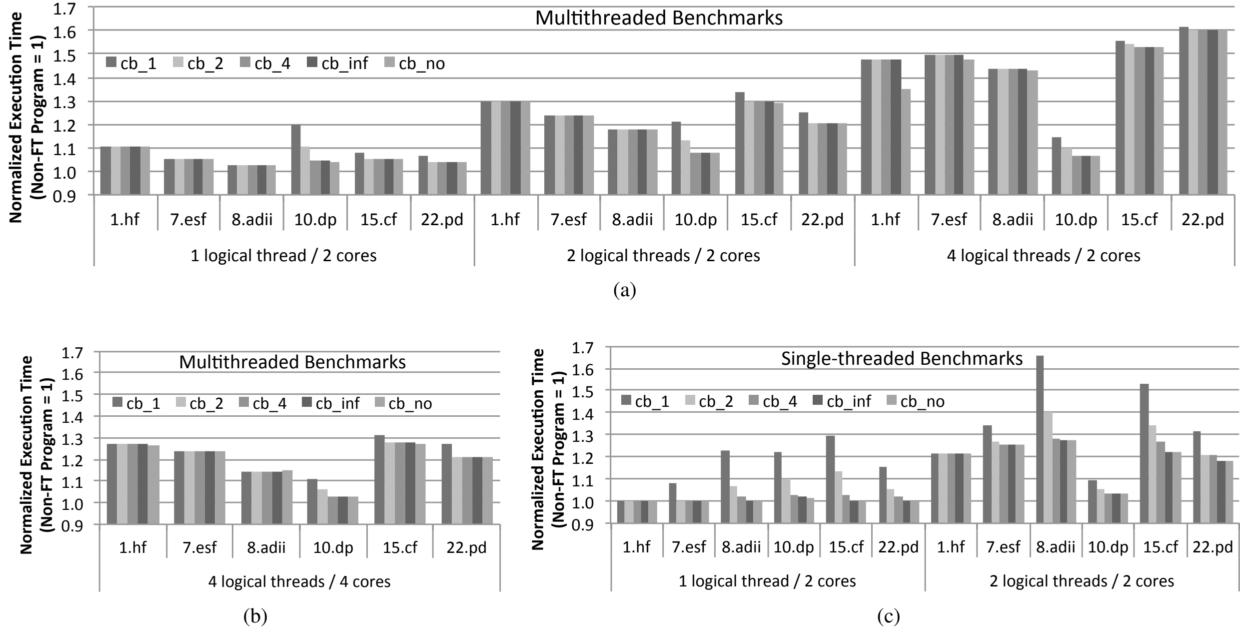

Fig. 1: The normalized execution time of DRMT with 1, 2, 4, infinite entries per set or no output comparison.

the multithreaded version. In other words, 10,000 threads run on the hardware by reusing thread contexts until the iterations complete. In contrast, the single-threaded loop version is a single thread that comprises a loop of 10,000 iterations.

TABLE II: The benchmarks

Loop 1.	Hydrodynamics fragment
Loop 7.	Equation of state fragment
Loop 8.	Alternating direction implicit integration
Loop 10.	Difference predictors
Loop 15.	Casual Fortran
Loop 22.	Planckian distribution

B. *Results*

This section evaluates the performance overhead of DRMT using one or more logical threads. A logical thread maps to a single hardware thread if fault detection is not enabled. However, in redundant mode, a logical thread is further decomposed into two hardware threads run on different cores. Meanwhile, we compare the results of multithreaded and single-threaded versions of the same program to evaluate the overhead of thread pairing and output comparison. In the evaluation of the multithreaded program, we find that 2-core and 4-core behavior are nearly identical, so we do not show 4-core results for the single-threaded program because of this similarity. Finally, we estimate the hardware overhead of the CB.

1) Multithreaded Benchmarks: Each multithreaded version of the benchmarks has 10,000 threads in total. The thread window[1] is the same as logical threads number in DRMT mode. Figure 1a shows that the average performance overhead is 5%, 22% and 44% for 1, 2 and 4 logical threads scenario, respectively. With the one logical thread scenario, the performance degradation is only between 3-11%. In this case the master and redundant threads are distributed to two separate cores, and the number of threads equals to the number of processing element, which means there is no resources contention. The performance overhead is mostly caused by thread pairing, see the results in section IV-B2. The performance overhead is increased to 8-30% and 7-60% as the number of threads is 2 or 4 times of processing elements in 2 and 4 logical threads scenario. Even if there is resources contention, the slowdown of programs in DRMT mode is still far less than 100% in a platform consisting of single-issue, in-order core. This is attributed to the latency tolerance of DMT.

Another result shown in figure 1a is that 4 entries per set of CB are enough for these benchmarks. It can reach the same performance with an infinite set. Also, the performance impact of increasing set size is not big, because each thread is small and only has a few stores in the multithreaded program. Besides it, we find that there is no performance loss due to output comparison if the set size is big enough. It is attributed to the overlapping of thread execution and output comparison.

We also run the evaluation on 4 cores, which is identical to the result of 2 cores. Figure 1b shows the normalized execution time of 4 logical threads run on 4 cores, which has the same distribution with 2 logical threads run on 2 cores. Comparing the results in figure 1a with 1b, they are nearly the same except that the performance overhead of the latter is slightly smaller than the former. It may be because the CB can ease the congestion caused by concurrent stores better in a more concurrent stores scenario, i.e., 4 logical threads with 4 cores.

2) Single-threaded Benchmarks: In order to evaluate the pressure of set capacity from the store number in a thread, we modify the multithreaded program to single-threaded program.

[1]The maximum number of threads that could be run in the same time in one core.

In figure 1c, we can find that the performance improvement via increasing set size is much bigger than multithreaded program as the single thread program executes less efficiently. However, we see that 4 entries per set can reach the same performance as the infinite set size. Loop 8 and 15 are store intensive benchmarks compared to the other loops, but all of them achieve the lowest performance overhead when each set has 4 entries. It shows that the set size is not only decided by the store number in a thread, but also related to the committing time difference of a store between master and redundant threads. When the set size is big enough to tolerate the communication latency between pipeline and CB, the set size is strongly related to the speed difference between master and redundant threads, which decides the lifetime of a store in the CB. In our evaluation, the workload in each core is quite similar. It means that master and redundant threads have nearly the same speed and progress, which does not pressure the CB. And the 4 entries are the critical size for tolerating the communication latency between pipeline and CB.

In addition, there is nearly no performance degradation in the 1 logical thread situation of the single-threaded program if the set is big enough. Comparing figure 1a with 1c, we find that thread pairing causes the performance overhead in 1 logical thread situation of multithreaded program, as there are 10,000 times thread pairings. Similar results can be concluded in the comparison of 2 logical threads scenarios in single-threaded and multithreaded program. But the difference of performance overheads between single-threaded and multithreaded benchmarks in the 2 logical threads scenario is much smaller than the 1 logical threads scenario. This is because of that the thread pairing overhead can also be hidden by multithreading.

3) Hardware Cost: The CB is the main hardware cost of DRMT. It comprises many sets, which correspond to the hardware threads in the core pair. Each set has many entries, which has four fields. As shown in table III, each entry has 137 bits. According to the experimental configuration of the MGSim (see table I) and the results above, the capacity of the CBs is nearly 1KB[2], which is small.

TABLE III: Length of an entry in comparison buffer

Filed	Address	Value	Mask	Flag	Total
Bit(s)	64	64	8	1	137

V. Related Work

The research on RMT became popular following the introduction of SMT, as it can benefit from the higher resource utilization when master and redundant threads co-exist within the processor. AR-SMT [7] proposed executing two copies of the same program in an SMT environment first. Then, SRT [8] improved the performance of AR-SMT using a slack fetch and branch outcome queue based on speculation and cache locality. SRTR [9] extends SRT to implement fault recovery. However, the drawback of SRT(R) is that it does not have scalability when the number of hardware threads increases.

Meanwhile, with the emergence of multicore processors, CRT [10], CRTR [11], Reunion [5], DCC [12], and HDTLR [13] apply redundant RMT to CMPs. They found that there were fewer overheads than performing RMT in a single SMT core, as CMPs mitigate the resource contention found in single cores. However, all these RMT works are based on SMT techniques. And the CMP-based RMTs use separate store buffers for master and redundant threads. The output needed to be compared is transferred via network, either a dedicated or existing one.

VI. Conclusion

This paper has presented the design and evaluation of the implementation of RMT in a DMT multicore processor in detail. The results with 6 Livermore Loops show that RMT can benefit from DMT, where the performance overhead is less than 60% even if the number of threads is 4 times the number of processing elements.

We also presented an asynchronous output comparison mechanism, which can be used by all RMT techniques. Compared to the previous mechanisms, it has three advantages. 1) It saves the capacity of CB since master and redundant threads share it. 2) It avoids the output transmission among cores because the output of both master and redundant threads are pushed to the CB. 3) It provides complete fault coverage of the program, as it does not compress the output for comparison. Evaluation shows that both performance and hardware overheads of asynchronous output comparison are insignificant.

[2] A 4-core processor has 2 CBs, each CB has 8 sets as each core supports 8 threads. Meanwhile, according to the above results, 4 entries per set can reach the lowest performance overhead. So the capacity of CBs is 2*8*4*137 = 8768 bits $\approx$ 1KB.

References

[1] A. Bolychevsky, C. R. Jesshope, and V. B. Muchnick, “Dynamic scheduling in RISC architectures,” in *IEE Proceedings Computers and Digital Techniques*, vol. 143, no. 5, 1996, pp. 309–317.

[2] R. A. Iannucci, “Toward a dataflow/von Neumann hybrid architecture,” in *Proc. of ISCA*, 1988, pp. 131–140.

[3] R. S. Nikhil, “Can dataflow subsume von Neumann computing?” in *Proc. of ISCA*, 1989, pp. 262–272.

[4] M. Lankamp, R. Poss, Q. Yang, J. Fu, I. Uddin, and C. R. Jesshope, “MGSim—simulation tools for multi-core processor architectures,” University of Amsterdam, Tech. Rep. arXiv:1302.1390v1, 2013.

[5] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe, “Reunion: complexity-effective multicore redundancy,” in *Proc. of MICRO*, 2006, pp. 223–234.

[6] F. H. McMahon, “Livermore Fortran kernels: a computer test of the numerical performance range.” Lawrence Livermore National Laboratory, Tech. Rep. UCRL-53745, Dec 1986.

[7] E. Rotenberg, “AR-SMT: a microarchitectural approach to fault tolerance in microprocessors,” in *Proc. of FTCS*, 1999, pp. 84–91.

[8] S. K. Reinhardt and S. S. Mukherjee, “Transient fault detection via simultaneous multithreading,” in *Proc. of ISCA*, 2000, pp. 25–36.

[9] T. N. Vijaykumar, I. Pomeranz, and K. Cheng, “Transient-fault recovery using simultaneous multithreading,” in *Proc. of ISCA*, 2002, pp. 87–98.

[10] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, “Detailed design and evaluation of redundant multithreading alternatives,” in *Proc. of ISCA*, 2002, pp. 99–110.

[11] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz, “Transient-fault recovery for chip multiprocessors,” in *Proc. of ISCA*, 2003, pp. 98–109.

[12] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar, “Utilizing dynamically coupled cores to form a resilient chip multiprocessor,” in *Proc. of DSN*, 2007, pp. 317–326.

[13] M. Rashid and M. Huang, “Supporting highly-decoupled thread-level redundancy for parallel programs,” in *Proc. of HPCA*, 2008, pp. 393–404.