A Case for Fault Tolerance and Performance Enhancement using Chip Multi-Processors

Huiyang Zhou
School of Computer Science, University of Central Florida
zhou@cs.ucf.edu

Abstract—**This paper makes a case for using multi-core processors to *simultaneously* achieve transient-fault tolerance and performance enhancement. Our approach is extended from a recent latency-tolerance proposal, dual-core execution (DCE). In DCE, a program is executed twice in two processors, named the front and back processors. The front processor pre-processes instructions in a very fast yet highly accurate way and the back processor re-executes the instruction stream retired from the front processor. The front processor runs faster as it has no correctness constraints whereas its results, including timely prefetching and prompt branch misprediction resolution, help the back processor make faster progress. In this paper, we propose to entrust the *speculative* results of the front processor and use them to check the *un-speculative* results of the back processor. A discrepancy, either due to a transient fault or a misprediction, is then handled with the existing misprediction recovery mechanism. In this way, both transient-fault tolerance and performance improvement can be delivered simultaneously with little hardware overhead.**

I. INTRODUCTION

Advances in semiconductor technology enable the integration of billion transistors on a single chip. Such exponentially increasing transistor counts makes reliability an important design challenge since a processor's soft error rate grows in direct proportion to the number of devices being integrated [7]. The huge amount of transistors, on the other hand, leads to the popularity of multi-core processor or chip multi-processor architectures for improved system throughput. In this paper, we make a case for using multi-core processors collaboratively to simultaneously achieve performance enhancement and transient-fault tolerance for single-threaded applications. To our knowledge, this is the first proposal that realizes *positive* performance improvements with *full* transient-fault coverage (i.e., redundancy checking for all committed instructions). Slipstream processors [13] also improve both performance and reliability but only with *partial* fault coverage.

Our approach is based on recently proposed dual-core execution (DCE) [15]. In DCE, a program is executed twice by two processors, named the front and back processors, similar to prior work on chip-level redundancy [6],[8],[13]. The front processor executes instructions in its normal way except for long-latency cache-missing loads, which are turned invalid similar to run-ahead execution [5],[9]. The retired instructions from the front processor are then re-executed in the back processor to provide precise program state. The front processor runs faster due to its virtually ideal L2 cache whereas the back

processors makes faster progress since the front processor fixes most branch mispredictions and initiates timely prefetches.

In this paper, we exploit the redundant execution in DCE to efficiently achieve transient-fault tolerance. In DCE, instructions are pre-processed by the front processor and re-executed by the back processor. Therefore, besides assisting the back processor execution, the pre-processing results can be used for redundancy checking. Moreover, since the pre-processing are speculative, a discrepancy due to a transient soft error at either the front or back processor can be simply treated as a misspeculation and corrected by the same misspeculation recovery mechanism. Since the front processor invalidates instructions such as cache-missing loads to run faster, such redundancy checking may not cover the complete instruction stream. To achieve the full redundancy coverage, we propose to modify the back processor slightly to enable redundant execution of those instructions that are invalidated by the front processor. Since such instructions are a small subset of a complete program and do not incur additional cache misses or mispredictions, the impact of such redundant execution is limited and full transient-fault coverage and significant performance improvement can be achieved.

The rest of the paper is organized as follows. Section II presents the background of DCE and discusses the related work. Extending DCE for fault tolerance is detailed in Section III. Experimental methodology and results are presented in Sections IV and V. Section VI concludes the paper.

II. BACKGROUND AND RELATED WORK

A. Background: Dual-Core Execution

Dual-core execution (DCE) is built upon two superscalar processors (called the front and back processors) coupled with a queue (called the result queue), as shown in Figure 1.
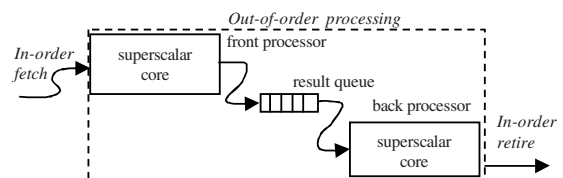

Figure 1. An overview of dual-core execution.

The front processor fetches instructions in-order and executes them in its normal way except for long-latency cache misses (e.g., L2 misses), for which an invalid (INV) value is used to substitute the data that are being fetched from memory, similar to run-ahead execution [5],[9]. The INV flag can be propagated through register data dependency and memory data dependency to invalidate the dependent instructions of the

long-latency cache-missing loads.

The front processor retires instructions as usual except store instructions and instructions raising exceptions. When a store instruction retires, it will not update the data cache and only updates a structure called run-ahead cache to communicate the store value to subsequent loads in the front processor. Exception handling is disabled in the front processor as the back processor maintains the precise program state.

The result queue is a first-in first-out structure, keeping the instruction stream retired from the front processor.

The back processor fetches instructions from the result queue and executes them in its normal way except for mispredicted branches. When a branch misprediction is detected, all the instructions in the back processor, the result queue, and the front processor are squashed. The back processor's current architectural state (including the program counter and architectural register values) are copied over to the front processor. Note that there is *no* need to synchronize memory states at the front processor and all the front processor needs to do is to invalidate its run-ahead cache.

In DCE, a program is pre-processed aggressively by the front processor and then re-executed by the back processor. The cache misses in the front processor become prefetches for the back processor. The branch mispredictions that are dependent on short latency operations are resolved promptly by the front processor and only those dependent on long-latency-cache misses are handled by the back processor. The high amount of instructions residing in the result queue are not associated with any centralized resource unlike in-flight instructions in a conventional superscalar design, which reserve their allocated resources such as physical registers, load/store queue entries, etc. Therefore, DCE provides a highly scalable, complexity -effective way to build a very large instruction window for latency tolerance. Overall, DCE requires only little hardware changes and achieves remarkable performance improvements [15]. In this paper, we focus on exploiting the redundant execution in DCE to improve transient-fault tolerance.

B. Related Work

In DCE, the front preprocessor pre-processes instructions similar to run-ahead execution [5],[9]. One key difference is that in run-ahead execution, whenever a mode-transition-triggering cache miss is repaired, the processor has to return to the normal mode even the pre-execution is along right paths and generates accurate prefetches. DCE eliminates this fundamental bottleneck and delivers much higher performance [15]. Compared to the approaches such as two-pass pipelining [2] or out-of-order processors with very large instruction windows [12],[4], the re-execution in the back processor eliminates the requirement of any centralized resources and enables the front processor to run further ahead [15]. Moreover, DCE enables efficiently ways to achieve fault tolerance as discussed in this paper.

By exploiting leading-thread results, leader/follower architectures achieve fault tolerance with low performance overhead. The leading thread/processor in most fault-tolerant leader/follower designs, including AR-SMT [11], DIVA [1], SRT [8], SRTR [14], and CRT [6], is *non-speculative*, i.e., correct if free from hardware faults. This constraint is a main reason for performance degradation since the leader has to wait the follower to check the execution results before retiring them (e.g., store values and store addresses). The delayed retirement increases the pressure on critical resources such as the store queue and register file. In DCE, the pre-processing is *speculative* and relieved of the correctness requirement, similar to the A-stream in slip-stream processors [13].

DCE and slipstream processors achieve their performance improvements in fundamentally different ways. In slipstream processors, the A-stream runs a shorter program based on the removal of ineffectual instructions while the R-stream uses the A-stream results as predictions to make faster progress. DCE, however, relies on the front processor to accurately prefetch data into caches. Conceptually, the R-stream in slipstream processors acts as a fast follower due to the near oracle predictions from the A-stream while the A-stream is a relatively slow leader since long-latency cache-misses still block its pipeline unless they are detected ineffectual and removed from the A-stream. Therefore, it is not necessary for the R-stream to take advantage of prefetching from the A-stream to make even faster progress. Compared to slipstream processors, DCE achieves much higher performance improvement with less hardware complexity (i.e., no need for IR-detectors, IR-predictors, and value prediction support) [15]. Both slipstream processors and DCE can exploit redundant execution to improve fault tolerance and the detected faults can be corrected with their existing misspeculation recovery mechanisms. In this paper, we also extend DCE to achieve redundancy checking for the complete instruction stream and the same extension is also applicable to slipstream processors.

III. FAULT TOLERANT DUAL-CORE EXECUTION

A. Extending DCE to improve fault tolerance

To improve fault tolerance using DCE, we can store the pre-processing results (if not invalid) in the result queue. The back processor then compares them with its results before committing them. If there is a transient fault at either the front or back processor resulting in an incorrect result, a discrepancy will be detected and the existing branch misprediction recovery mechanism in the back processor can transparently provide fault tolerance by rewinding both the front and back processors to the current architectural state, which is protected with information integrity coding schemes such as ECC. We call DCE with such a simple extension as DCE_R. To protect from the transient faults that could result in a deadlock, a watchdog timer similar to what used in DIVA [1] can be added in the back processor to restart the execution from the current architectural state whenever the timer expires.

Since the pre-processing in the front processor is speculative, some discrepancies detected in DCE_R may be due to a mispeculation by the front processor rather than an actual transient fault and such discrepancies will incur additional

mispeculation recoveries, thereby affecting the overall performance. Next, we analyze the sources of such mispeculations to show that they are extremely rare cases.

In DCE, only long-latency cache-missing loads are invalidated by the front processor and *independent* operations are not affected. Therefore, those pre-processing results should be correct if there are no transient faults. Among *dependent* instructions, if the dependency is carried through registers, the INV flag propagation will invalidate such dependent instructions. If the dependency is carried through memory, the INV propagation using store-load forwarding and the run-ahead cache will invalidate most of the dependent operations as well. Only in the very rare cases such as a store with an invalid address followed by a load accessing the same location or a store value being replaced from the run-ahead cache due to its limited capacity, a stale value could be fetched by a load instruction. When this stale value is detected in the back processor, the front processor is rewound to a correct architectural state and the same problematic load will fetch the right value since the previous stores have already been committed to the D-caches by the back processor.

Since the instructions that are invalidated by the front processor are executed only in the back processor, their results are susceptible to transient faults. In other words, only partial redundancy coverage is achieved by DCE_R, similar to slipstream processors. Nevertheless, as the front processor only invalidates cache-missing loads and their dependents, DCE_R effectively achieves high transient-fault coverage and remarkable performance improvements (see Section V).

B. Achieving full redundancy coverage

To achieve the redundancy coverage for all instructions, we propose to extend the back processor in DCE_R so that it dual-executes the instructions that are invalidated by the front processor and such a scheme is named DCE_FR.

In DCE_FR, the result queue appends a flag (F_INV) to each instruction to show whether it is validated by the front processor. When the back processor fetches an instruction with a true F_INV, the same instruction will be fetched twice, one for normal execution and the other for redundancy checking. To support such redundant execution, an additional renaming table (called A_Table) is introduced in the back processor and the renaming logic is modified as follows assuming the back processor is a MIPS R10000 style superscalar processor. The instructions, which are invalidated by the front processor, will access and update the original renaming table (called R_table) and their redundant copies will only access and update the A_table. For the instructions that do not need redundant execution, i.e., not invalidated by the front processor, their source operands will access the R_table but their destination operands will update both the R_table and A_table. The process can be illustrated using the example in Figure 2.

In Figure 2, instructions *A* and *B* are invalidated by the front processor, so they are replicated in the back processor when fetched from the result queue. Instruction *C* is not invalidated by the front processor and carries a valid result in the result

queue for redundancy checking. Instructions *A* and *B* access and update the R_table while their redundant copies *A'* and *B'* access and update the A_Table. Instruction *C*, however, accesses the R_table for its source operands and updates both the R_table and A_table for its destination operand. Doing so, however, it needs to store two previous mappings from the two tables and release both of them at the retire stage if they are different. The invalidated instructions and their redundant copies, however, only need to release a single previous mapping.
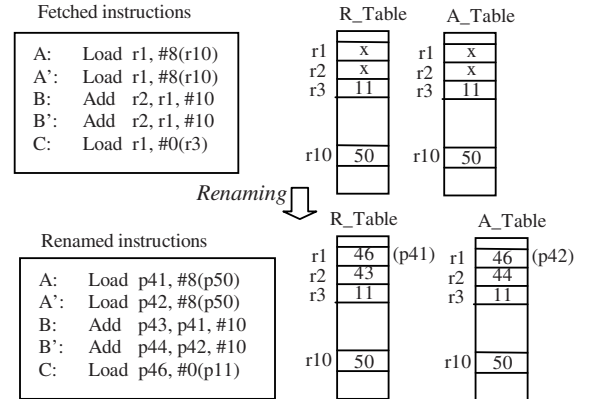
Figure 2. A code example to illustrate the renaming process in the back processor for redundant execution.

The redundant execution in the back processor is similar to the previous work on dual-use of data path for fault tolerance [10]. The difference is that in our approach, only a small subset of instructions needs to be executed twice. In addition, those instructions are replicated as early as the fetch stage, extending the pipeline replication sphere in [10].

The back processor commits instructions only after they are checked with the results either from the front processor or from their redundant copy. Any discrepancy will initiate a recovery process same as a branch misprediction resolution in DCE.

The power/energy overhead in DCE_R and DCE_FR mainly comes from those invalidated instructions in the front processor as they do not provide useful execution results. Fortunately, those instructions account for only a small subset of overall instructions and invalidating them consumes much less power/energy than actually executing the operations.

IV. SIMULATION METHODOLOGY

To evaluate the proposed schemes, we built our simulation environment using the SimpleScalar toolset [3]. The cache modules in our simulator model both data and tag stores and wrong path events are also faithfully simulated. The front and back processors have the same configurations (private L1 caches with a shared L2 cache), shown in Table 1. The default result queue has 1024 entries and the default run-ahead cache is 4kB, 4-way associative with a block size of 8 bytes. The delay of the result queue is assumed as 16 cycles to account for inter-processor communication. A latency of *64* cycles is assumed for copying the architectural register values from the back processor to the front processor. Therefore, a branch

misprediction resolved in the back processor has a minimum penalty of 10(front pipeline) + 16(queue) + 9(back pipeline) + 64(recovery penalty) cycles. Each processor also has a stride-based stream buffer hardware prefetcher and SPEC 2000 benchmarks are selected with the same criterion as in [15].

Table 1. Configuration of the front and back processors.

|	
Pipeline	3-cycle fetch stage, 3-cycle dispatch stage, 1-cycle issue stage, 1-cycle register access stage, 1-cycle retire stage. Minimum branch misprediction penalty = 9 cycles
Instruction Cache	Size=32 kB; Assoc.=2-way; Replacement = LRU; Line size=16 instructions; Miss penalty=10 cycles.
Data Cache	Size=32 kB; Assoc.=2-way; Replacement=LRU; Line size = 64 bytes; Miss penalty=10 cycles.
Unified L2 Cache (shared)	Size=1024kB; Assoc.=8-way; Replacement = LRU; Line size=128 bytes; Miss penalty=220 cycles.
Br Predictor	64k-entry G-share; 32k-entry BTB
Superscalar Core	Reorder buffer: 128 entries; Dispatch/issue/retire bandwidth: 4-way superscalar; 4 fully-symmetric function units; Data cache ports: 4. Issue queue: 64 entries. LSQ: 64 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
Mem. Disambig.	Perfect memory disambiguation

V. EXPERIMENTAL RESULTS

The performance impact of both DCE_R and DCE_FR is examined in Figure 3, which shows the normalized execution time of a single baseline processor, DCE, DCE_R, and DCE_FR. Each cycle is categorized as a pipeline stall with an empty reorder buffer (ROB), a stall with a full ROB due to cache-misses, a stall with a full ROB due to other factors such as long-latency floating-point operations, or a cycle in un-stalled execution. In DCE-based schemes, such cycle time distribution is collected from the back processor.
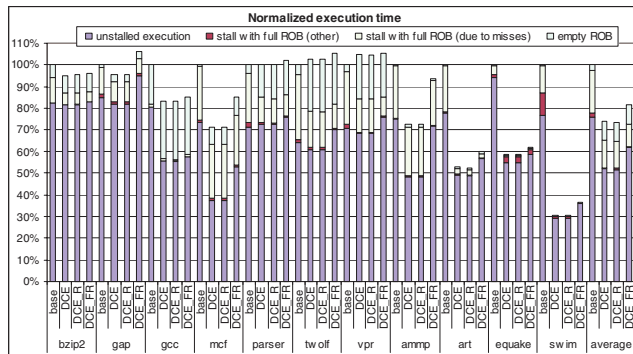
Figure 3. Normalized execution time of a single baseline processor, DCE, DCE_R, and DCE_FR.

From Figure 3, it can be seen that both DCE_R and DCE_FR achieve significant performance improvement, 35.7% and 22.5% on average respectively, over the single baseline processor. Slight performance degradation observed in *parser*, *twolf*, and *vpr* is due to their relatively high number of branch mispredictions dependent on cache-missing loads. Compared to DCE, DCE_R has negligible performance impact and the reason is that the number of additional recoveries due to redundancy checking is very small, 0.02 recoveries per 1000 retired instructions on average. On the other hand, an average of 81% of all the retired instructions is ensured with redundancy checking in DCE_R, as show in Figure 4. With redundant execution of the instructions that are invalidated by

the front processor, DCE_FR ensures *all* retired instructions with redundancy checking. Such redundant execution in the back processor has different performance impact for different benchmarks. As shown in Figure 3, for *gap*, *mcf*, *ammp*, *art*, and *swim*, DCE_FR results in many more non-stall execution cycles than DCE_R. The reason is that many instructions are invalidated by the front processor for those benchmarks, as reported in Figure 4 (lower coverage means more instructions invalidated by the front processor). For other benchmarks, the performance impact is relatively limited. Overall, DCE_FR achieves a 22.5% performance improvement with redundancy checking for all retired instructions, a remarkable improvement over the prior work on chip-level redundancy.
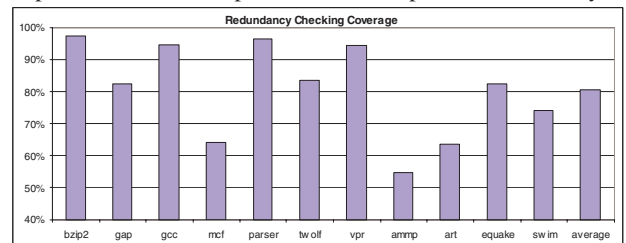
Figure 4. The percentage of retired instructions with redundancy check for each benchmark.

VI. CONCLUSIONS

This paper makes a case for simultaneously achieving both performance improvement and transient-fault tolerance using multi-core processors. The key innovation is fast, speculative, yet highly accurate pre-processing in one core followed by re-execution in another core. Fast pre-processing accelerates re-execution and enables efficient redundancy checking. Re-execution relieves the correctness requirement from pre-processing and eliminates the requirement of any centralized resources.

REFERENCES

[1] T. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design", MICRO-32, 1999.
[2] R. Barnes et. al., , "Beating in-order stalls with flea-flicker two pass pipelining", *MICRO-36*, 2003.
[3] D. Burger and T. Austin, "The SimpleScalar tool set, v2.0", *Computer Architecture News*, vol. 25, June 1997.
[4] A. Cristal et al., "Out-of-order commit processors", *HPCA-10*, 2004.
[5] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss", ICS-97, 1997.
[6] M. Gomma et. al., "Transient-fault recovery for chip multiprocessors", ISCA-30, 2003.
[7] S. Mukherjee, J. Emer, and S. Reinhardt, "The soft error problem, an architectural perspective", HPCA-11, 2005.
[8] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives", ISCA-29, 2002.
[9] O. Mutlu et. al., "Runahead execution: an alternative to very large instruction windows for out-of-order processors", HPCA-9, 2003.
[10] T. Ray, J. Hoe, and B. Falsafi, "Dual use of superscalar datapath for transient-fault detection and recovery", MICRO-34, 2001
[11] E. Rotenberg, "AR-SMT: a microarchitectural approach to fault tolerance in microprocessors", FTCS-29, 1999.
[12] S. T. Srinivasan et. al., "Continual flow pipelines", *ASPLOS-11*, 2004.
[13] K. Sundaramoorthy et. al., "Slipstream processors: improving both performance and fault tolerance", ASPLOS-9, 2000.
[14] T. Vijaykumar et. al. "Transient-fault recovery using simultaneous multithreading", ISCA-29, 2002.
[15] H. Zhou, "Dual-core execution: building a highly scalable single-thread instruction window", PACT'05, 2005.