Multiplexed Redundant Execution: A Technique for Efficient Fault Tolerance in Chip Multiprocessors

Pramod Subramanyan, Virendra Singh
Supercomputer Education and Research Center
Indian Institute of Science
Bangalore, India
{*pramod@rishi., viren@*}*serc.iisc.ernet.in*

Kewal K. Saluja
Electrical and Computer Engg. Dept.
University of Wisconsin-Madison
Madison, WI
saluja@engr.wisc.edu

Erik Larsson
Dept. of Computer and Info. Science
Linköping University
Linköping, Sweden
erila@ida.liu.se

***Abstract*—Continued CMOS scaling is expected to make future microprocessors susceptible to transient faults, hard faults, manufacturing defects and process variations causing fault tolerance to become important even for general purpose processors targeted at the commodity market.**

To mitigate the effect of decreased reliability, a number of fault-tolerant architectures have been proposed that exploit the natural coarse-grained redundancy available in chip multiprocessors (CMPs). These architectures execute a single application using two threads, typically as one leading thread and one trailing thread. Errors are detected by comparing the outputs produced by these two threads. These architectures schedule a single application on two cores or two thread contexts of a CMP. As a result, besides the additional energy consumption and performance overhead that is required to provide fault tolerance, such schemes also impose a *throughput loss*. Consequently a CMP which is capable of executing $2n$ threads in non-redundant mode can only execute half as many (n) threads in fault-tolerant mode.

In this paper we propose multiplexed redundant execution (MRE), a low-overhead architectural technique that executes multiple trailing threads on a single processor core. MRE exploits the observation that it is possible to accelerate the execution of the trailing thread by providing execution assistance from the leading thread. Execution assistance combined with coarse-grained multithreading allows MRE to schedule multiple trailing threads concurrently on a single core with only a small performance penalty. Our results show that MRE increases the throughput of fault-tolerant CMP by 16% over an ideal dual modular redundant (DMR) architecture.

I. Introduction

Over the last three decades continued scaling of silicon fabrication technology has permitted exponential increases in the transistor budgets of microprocessors. In the past higher transistor counts were used to increase the performance of single processor cores, but the increasing complexity and power dissipation of these cores forced architects to turn to chip multiprocessors (CMPs) in order to deliver increased performance at a manageable levels of power and complexity with each succeeding generation of silicon fabrication technology. While deep sub-micron technology is enabling the placement of billions of transistors on a single chip, it also poses unique challenges. ICs are now increasingly susceptible to soft errors [24], wear-out related permanent faults and process variations [2, 4].

Traditionally, high availability systems have been restricted to the domain of mainframe computers or specially designed fault-tolerant systems [3, 11]. However, the trend towards unreliable components means that fault tolerance is now important for the commodity market as well [1]. Fault tolerant solutions for the commodity market have different requirements and present a different set of design challenges for architects. The commodity market requires *configurable* [1] and *low cost* fault tolerance. CMPs are appealing in this context as they inherently provide replicated hardware resources which can be exploited for error detection and recovery. A number of proposals [1, 5, 8, 9, 12, 13, 18, 26] have attempted to take advantage of these aspects of CMPs to provide fault tolerance. Typically, these schemes use some form of space redundancy [21] where two cores or thread contexts of a CMP are used to execute a single logical thread. Inputs to the two cores are replicated and the outputs generated by the two cores are compared to detect the occurrence of errors.

However, the use of two cores or thread contexts to execute a single program means that the throughput of the CMP is reduced by half. Due to this *throughput loss*, a fault-tolerant system must have twice as many cores to achieve the same throughput as non-redundant execution. Not only does this increase the procurement cost of systems, it also increases the running cost of the system due to increased cooling costs, energy costs and maintenance costs; these costs may be greater than the cost of purchasing the system. Such high costs are undesirable for fault-tolerant general purpose microprocessors targeted at the commodity market. Therefore, there is a need for fault-tolerant architectures that can minimize this throughput loss.

In this paper, we present multiplexed redundant execution (MRE), a technique that reduces the throughput loss due to fault tolerance. MRE employs the well known technique of space redundancy [21], where two copies of a program are executed on different cores of a CMP with replicated inputs. The outputs of these two streams of execution are compared to detect faults. MRE is based on the observation that forwarding branch outcomes and load values from the leading thread to the trailing thread *accelerates* the execution of the trailing thread. Supplying the trailing thread with the branch outcomes from the leading threads eliminates mispredictions in the trailing thread. Similarly, forwarding load values eliminates performance degradation due to data cache misses in the trailing thread. This accelerated execution *frees execution bandwidth* that can be used for executing other threads or applications.

Based on this observation, MRE uses coarse-grained multithreading to schedule several trailing threads on a single processor core. Therefore unlike previous work, where each leading/trailing thread combination requires two cores for execution, we are able to *multiplex multiple trailing threads on a single trailing core* with only small performance penalty as compared to non-redundant execution. Our evaluation shows that MRE increases the throughput of a fault-tolerant CMP by 23% as compared to Reinhardt and Mukherjee's chip level redundant threading (CRT) [18] architecture. When compared to an ideal dual modular redundant architecture with no performance overhead for communication and comparison, MRE still increases throughput by 16%.

II. Conceptual Overview

MRE partitions the processors of a CMP into different pools of cores. One pool is the set of leading cores. These cores execute the

978-3-9810801-6-2/DATE10 © 2010 EDAA

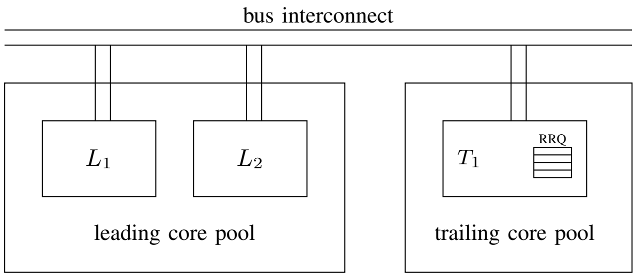


Fig. 1. Conceptual block diagram of MRE.

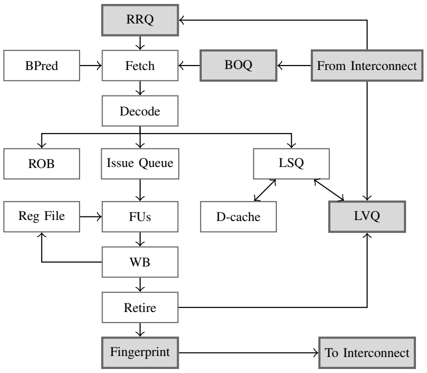


Fig. 2. Diagram showing processor core augmented with structures required for redundant execution. Newly added structures are shaded and not used when redundant execution is not being performed.

leading threads of applications that require fault tolerance. A second pool consists of the set of trailing cores. These cores execute trailing threads of applications which require fault tolerance. A third pool of processors executes non-redundant applications. The division of processor cores into leading, trailing and non-redundant cores is only a logical distinction. Physically, all cores are identical and unused structures are appropriately disabled based on the mode of operation.

This partitioning is similar to traditional fault tolerance schemes for CMPs except that the pool of trailing cores is allowed to be smaller than the pool of leading cores. In this case, *a single trailing core executes multiple trailing threads*. These trailing threads are executed concurrently using coarse-grained multithreading.

Execution of the application is carried out in *chunks*. The leading core executes a *chunk* of instructions and sends a message to the trailing core requesting the execution of the corresponding chunk. Upon the receipt of a request to execute a chunk, the trailing core pushes this request into a *run request queue (RRQ)*. When the current chunk executing in the trailing core completes execution, the trailing core (if necessary) switches contexts and executes the next request from the head of the RRQ.

To accelerate the execution of the trailing core, the leading core provides the trailing core with execution assistance in the form of load values and branch outcomes. The trailing core uses the branch outcomes from the leading core as branch predictions, and reads the load values instead of accessing the data cache. These two factors improve the performance of the trailing core, enabling *multiplexing* of multiple trailing threads on a single core.

III. Detailed Design

Figure 2 shows the block diagram of an MRE-enabled processor core. In the rest of this section we describe in detail the components of an MRE processor.

A. Overview

In this subsection, we give an overview of an MRE-enabled processor core.

Input Replication: A fault-tolerant architecture that relies on redundant execution has to address the issue of input replication. Precise input replication requires that both the leading and trailing threads of a redundant application see exactly the same sequence of inputs (e.g., load values, interrupts etc.). To ensure precise input replication, MRE uses the load value queue (LVQ) structure introduced by Reinhardt and Mukherjee [21]. When the leading thread retires a load instruction, the result of that instruction is transferred over the interconnect to the trailing core's load value queue. The trailing thread's load instructions access the LVQ instead of the data cache.

Branch outcome forwarding: Upon the retirement of a branch instruction in the leading thread, the branch's outcome is forwarded to the trailing thread. This outcome is stored in the branch outcome queue (BOQ) structure [21] in the trailing core. The trailing thread accesses the BOQ instead of the branch predictor to obtain branch predictions.

Interconnect: MRE's interconnection strategy divides the processors of a CMP into *clusters*. The processors in a cluster are connected by a bus interconnect. We model an 8-core CMP with 2 clusters of 4 processors each.

B. Coarse-grained multithreading

The trailing core uses coarse-grained multithreading to execute multiple trailing threads. Execution of an application is carried out in *chunks*. Each time the leading core executes a certain number of instructions, it sends a request to the trailing core to execute the corresponding chunk. This request is enqueued in the *run request queue* (RRQ) in the trailing core. When the trailing core finishes the execution of a chunk, it executes the next chunk from the head of the RRQ. If necessary, a context switch is performed. Unless a fingperprint comparison is to be made (described in §III-C), the leading core continues execution after signalling the end of a chunk.

Execution in chunks is required for fairness. Since requests are enqueued in the RRQ in the order in which they are received, this mechanism ensures forward progress for all threads.

Implementation of coarse-grained multithreading requires that all architecturally visible state of each of the trailing threads needs to be stored in the trailing core. This requires storage space for holding $N_{threads} \times N_{ArchRegs}$ registers. When a context switch is performed all architectural registers of the previously executing thread need to be copied to the register storage space. Architectural registers of the new thread need to be copied from the register storage space to the physical register file. Storing the state of multiple threads on a single core has been proposed in previous work in the context of core salvaging [20].

Sharing of the LVQ and BOQ: The LVQ and BOQ structures have to be shared among multiple threads in the trailing core. To ensure greater utilization MRE dynamically allocates *sections* of each structure to each thread in an *on-demand* fashion.

The sharing of the LVQ and BOQ is performed by maintaining a set of queues for each structure. One queue is shared across all the threads and is called the *free queue*. This contains a list of unused sections. A second set of queues called the *allocated queues* are maintained on a per-thread basis. Each time a thread is allocated a section, that section is removed from the free queue and added to the tail-end of the thread's allocated queue. Besides the section number, the allocated queue also maintains a count of the free entries for that section.

The head of the allocated queue points to the section where the next value can be deleted from. An LVQ entry is freed at the time of instruction retirement by incrementing the free entry count stored in the allocated queue. For the BOQ, the same operation is performed at the time of instruction fetch itself. When the number of free entries reaches the section size, the section is freed by deleting the allocated queue entry and pushing it to the free queue. The tail of the allocated queue points to the section where the next value will be inserted. This entry is consulted when a branch outcome or load value arrives over the interconnect so that it can be inserted in the appropriate location. Insertion of a value decreases the free entry count for that section. When the free entry count reaches zero, a new section is allocated. For the LVQ a third pointer into the queue is maintained, which is updated at the time of instruction dispatch. This points to the section from which the next load instruction to be dispatched will read its value.

In our implementation, each queue consists of a maximum of 32 entries storing five bits for each entry in the free queue and 11 bits for each entry in the allocated queue. Hence, the hardware overhead due to the sharing mechanism is very small.

C. Fault Tolerance Mechanisms

Any fault-tolerant system needs to address four important issues: fault detection, fault isolation, fault recovery and fault coverage. The following subsections discuss these topics in the context of MRE.

Fault Detection: The occurrence of faults is detected by MRE in two ways. The cores executing a logical thread periodically exchange fingerprints which summarize the execution history of the two cores. A mismatch in the fingerprints generated by the cores indicates the occurrence of an error. Another way in which MRE detects errors is when a branch misprediction occurs in the trailing thread. Since resolved branch outcomes are forwarded from the leading to the trailing thread as predictions, when the trailing thread mispredicts, it must be due to an error.

Fault Isolation: When a fault occurs in an MRE processor, it may be detected only when the next fingerprint comparison occurs. Between the time that the fault occurs and the time it is detected, fault isolation requires that the fault must not propagate outside the processor or to other processes. There are two ways in which this can happen.

Firstly, a corrupt cache block may be replaced and written back to a lower level of the memory hierarchy, from where it can propagate to main memory or other processes. MRE prevents this by using a speculative versioning L1 data cache [10]. Such a cache stores a speculative bit along with every cache line. Any write to cache line sets the speculative bit. If the speculative bit is set, a cache line is deemed to be *locked* and is not allowed to be written back to a lower level of the memory hierarchy. When fingerprints are compared and found to match, the speculative bits of all lines in the cache are cleared. When a write to a non-speculative line is performed, the line must be written back to the L2 cache to ensure that a verified copy of the line is always available for recovery. If fingerprints do not match, then memory state is recovered by invalidating all speculative lines in the L1 data cache. Since lower levels of the memory hierarchy always contain verified data, all memory updates since the last checkpoint are "undone" by the invalidation.

For correct execution of multithreaded workloads, the speculative bit must be transmitted along with the data when one cache supplies data to another cache. If a line needs to be replaced and all the lines in its set are locked, then a fingerprint comparison is initiated. When the fingerprint comparison is complete, the lines will be unlocked and the memory access can be completed.

The second method by which a fault may propagate outside the processor is through I/O operations. MRE forces a checkpoint to be taken and fingerprints compared before each I/O operation. This ensures that I/O is done only with verified data.

Checkpointing and recovery: Periodically, the two cores exchange fingerprints to detect errors. If the fingerprints are found to match in both cores, the leading core stores all architecture registers in the *checkpoint store*. It then clears the speculative bits of all cache lines.

If the fingerprints do not match in at least one core, then recovery is performed in three steps. Firstly, the register states of the two processors are restored from the *checkpoint store*. Secondly, all speculative lines in the L1 data cache are invalidated. Finally, the leading processor restarts execution from the next instruction after the last checkpoint.

Fault Coverage: MRE detects faults that occur in processor logic with the exception of those that occur in certain parts of the memory access circuitry. Since only one core accesses the memory hierarchy, faults that affect the memory access circuitry (e.g., the store-to-load forwarding logic) may not be detected. Circuit level techniques [7] may be used to detect these errors. A similar problem exists with cache controller logic and memory controller logic. Although ECC can protect the data and possibly the tag bits in a cache, it cannot protect against errors in the controller logic. Depending on the reliability target, this logic may have to be protected by circuit level techniques such as radiation hardening. This loss in coverage is not unique to MRE, but is common to SRT [21] and all its derivatives such as SRTR [28], CRT [18] and CRTR [13] and. Even architectures like Reunion [9], which independently access the L1 cache instead of replicating load values, share the lower levels of memory hierarchy. As a result, they provide only slightly higher fault coverage than MRE for the memory subsystem. For example, an error occurring in the L2 cache *controller* circuitry cannot be detected by Reunion even if the L2 cache *data* is protected using ECC.

Since MRE only compares fingerprints, there is some loss in fault coverage due to *fingerprint aliasing*. However as shown in [8], for reasonable error rates and fingerprint widths, the probability of an undetected error due to fingerprint aliasing is minuscule.

D. Comparison with SMT

Like coarse-grained multithreading, Simultaneous Multithreading (SMT) also allows the execution of multiple threads on a single core. However SMT requires major modifications of the processor core as compared to coarse-grained multithreading. In an SMT processor, fetch circuitry is modified to implement a fetch policy like I-count. The processor has to decode and issue from multiple threads in a single cycle. Per-thread structures like register map tables have to be replicated. SMT also suffers from the problem of interference between threads. This interference necessitates increasing the size of structures like the physical register file, the data cache and reorder buffer as well as increasing the width of the superscalar processor to provide performance and power characteristics that are commensurate with the hardware overheads of SMT [16, 17].

In comparison, the implementation of coarse-grained multithreading is relatively straightforward. Our results show that coarse-grained multithreading is a viable mechanism to increase fault-tolerant CMP throughput, without the overheads of SMT.

IV. Evaluation

In this section we evaluate MRE's performance and energy characteristics for single-threaded and multithreaded workloads. To put our results in context we also compare MRE with the Chip-level Redundantly Threaded (CRT) processors proposed by Mukherjee et al. [18]. We wish to study both MRE's single-thread performance, as well as quantify the increase in throughput due to multiplexing.

A. Simulation Methodology

We used an appropriately modified version of the execution-driven simulator SESC [14]. Unlike previous work [13, 18] we modeled an interconnect with finite bandwidth, assuming that the width of the interconnect is scaled linearly with the number of processors connected to it. Our simulator carefully modeled the interconnect, the associated arbitration and contention delays and the send and receive queues connecting the processor to the interconnect.

Our power model is based on Wattch [6]. We used CACTI [27] to generate power models for the LVQ and BOQ structures and incorporated these in our simulator.

Our workload consisted of nine benchmarks from the SPEC CPU 2000 suite. To reduce simulation times, we used the MinneSPEC [15] reduced input sets. For the single threaded results, we simulated the program until completion, while the multithreaded results were simulated for a total of 10^9 instructions. For multithreaded benchmarks which completed before the execution of 10^9 instructions, we used the *ref* inputs instead of the MinneSPEC inputs. When using the *ref* inputs we skipped the initialization part of the program and simulated 10^9 instructions.

B. Normalized Throughput Per Core

To quantify the reduction in throughput loss we introduce the metric *normalized throughput per core (NTPC)*. The *normalized throughput of a single thread* is defined as the IPC of that thread when when running in redundant mode divided by the IPC of that thread when running in non-redundant mode. The *normalized throughput per core of a workload* is defined as the sum of the normalized throughputs of the threads comprising the workload divided by the total number of cores the workload is running on. For an ideal dual modular redundant system which suffers no performance overhead, the normalized throughput would be 0.5. A real DMR system will always have some overheads due to communication and comparison, reducing the NTPC to less than 0.5.

$$NTPC = \frac{1}{N_{cores}} \sum_{i=1}^{N} \frac{IPC_{redundant}(i)}{IPC_{non-redundant}(i)}$$

NTPC is the same as Snavely and Tullsen's [25] weighted speedup metric scaled by the number of cores being used.

C. Results: One Logical Thread Without Multiplexing

In this section we show results for a single threaded workload on MRE and compare these with Mukherjee and Reinhardt's chip level redundant threading (CRT) [18] scheme.

Figure 3(a) shows the CPI of each application, normalized with respect to the CPI of non-redundant execution. MRE's performance overhead is minimal, restricted to a few percent in every case while CRT's performance overhead is significant for a number of benchmarks. There are two reasons for this. Firstly, the comparison of stores between the leading and trailing threads increases interconnect traffic. This is part of the reason for the slowdown of `bzip2`, `crafty` and `sixtrack`. Secondly, since leading thread stores cannot be retired from the store buffer until they are compared with trailing thread stores, the occupancy of the store buffer is increased. This additional pressure on the store buffer plays a major role in the slowdown of `mgrid`, `bzip2` and `crafty`.

MRE's mean performance degradation is just 2%, whereas CRT's mean performance degradation is about 18%.

Figure 3(b) shows the energy consumption of MRE and CRT normalized with respect to that of non fault-tolerant execution. MRE's energy consumption is slightly less than that of CRT. On an average CRT consumes 1.86 times the energy of non fault-tolerant execution, while MRE consumes 1.81 times the energy of non fault-tolerant execution.

The results in this section indicate that without multiplexing, our proposal suffers from *almost no performance loss*. Multiplexing is dynamically configurable at runtime by the operating system. Therefore if any workload suffers excessive performance degradation due to multiplexing, then multiplexing can be turned off to execute the program with almost no loss in performance.

D. Results: Multiplexing With Two Logical Threads

We constructed several 2-program workloads using benchmarks from SPEC CPU 2000 suite. For MRE, we multiplexed the trailing threads of both benchmarks on a single core, using *only 3 cores to redundantly execute 2 logical threads* without any loss in fault coverage. For CRT, we executed the 2 logical threads using 4 cores. Figure 4(a) shows the normalized throughput per core for the workloads. Since MRE multiplexes two logical applications on three threads, the maximum NTPC that can be obtained is $2/3 = 0.67$. A couple of benchmarks are able to approach this limit. `Ammp_bzip2` has NTPC 0.65, while `ammp_swim` and `mcf_parser` have NTPC 0.64. In all the benchmarks the MRE's NTPC is greater than that of CRT. The mean NTPC achieved by MRE is 0.58, while the mean NTPC of CRT of 0.47, indicating that MRE increases the throughput of a fault-tolerant CMP by 23% over CRT and 16% over an ideal DMR system (*i.e.* a system with NTPC 0.5).

Figure 4(b) shows the *weighted speedup* metric introduced by Snavelly and Tullsen [25] for the 2-program workloads. The weighted speedup of a workload is defined as the average of normalized IPCs of the threads in the workload. Here, the normalization is performed with respect to the IPC of non-redundant execution. This metric measures the performance of the fault-tolerant system relative the non-redundant system.

In all cases, MRE's performance with 3 cores is similar to CRT's performance with 4 cores. In fact, for the benchmarks `ammp_bzip2` and `crafty_swim`, *MRE executing on only 3 cores is faster than CRT executing on 4 cores.*

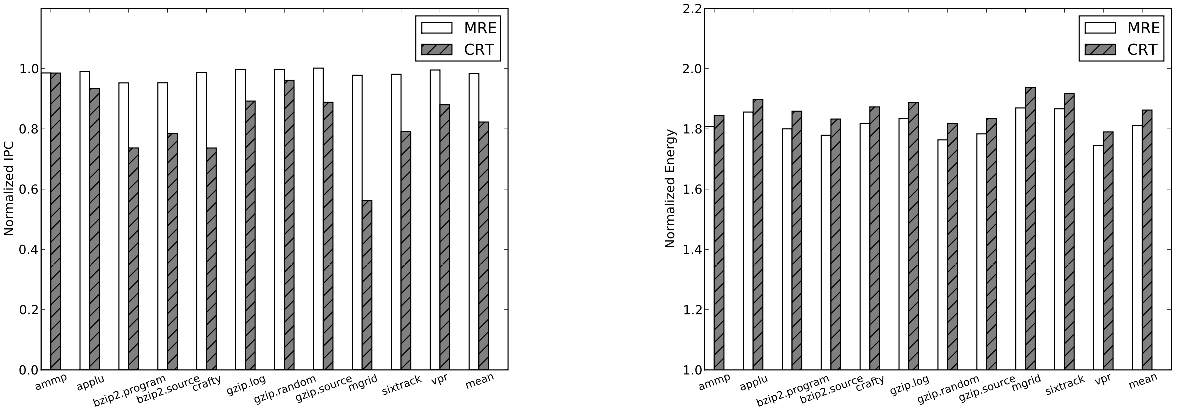
V. Related Work

Current high availability systems like the HP Nonstop Advanced Architecture [3] and the IBM zSeries [11] are high cost systems that spare no expense to meet reliability targets. Although they provide excellent fault coverage, they impose a high cost of 100% hardware duplication and 100% additional energy consumption, and 50% throughput loss. For high availability systems targeted at the commodity market, these high costs are unacceptable.

Transient fault detection using simultaneous multithreading was introduced by Rotenberg in AR-SMT [22] and Reinhardt and Mukherjee [21] in Simultaneously and Redundantly Threaded (SRT) processors. An SRT processor augments SMT processors with additional architectural structures like the branch outcome queue and load value queue for transient fault detection. The branch outcome

TABLE I
CMP Configuration

Number of cores	8	Fetch/issue/retire width	6/3/3	ROB size	128
Int/FP window	64	Load/store queue	32	Mem/Int/FP units	2/6/4
Branch predictor	hybrid/16k/16k/16k	BTB	2k/2-way	RAS	32 entries
I-cache	32k/64B/4-way/2 cycles	D-cache	64k/64B/4-way/2 cycles	L2 (private)	1MB/64B/8-way/14 cycles
Memory	450 cycles	LVQ	2k/2 cycles/2R1W port	BOQ	2k/2 cycles/1R1W port
RRQ	64 entries	LVQ sections	64 entries	BOQ sections	64 entries
Interconnect latency	8 cycles (obtained using [19])	Interconnect width	$n \times WordSize$ (n = # of processors)	Context switch latency	30 cycles



(a) Normalized CPI

(b) Normalized energy

Fig. 3. Comparison of MRE with CRT for a single program workload. Results are normalized with respect to non-redundant execution.

queue enhances the performance of the redundant thread, while the load value queue provides input replication. Since an SRT processor provides an unpredictable combination of space and time redundancy, it cannot guarantee the detection of all permanent faults. Muhkerjee et al. also introduced chip level redundant threading (CRT) [18], which extends SRT to simultaneously multithreaded chip multiprocessors. Gomaa et al. studied Chip Level Redundant Threading with Recovery (CRTR) [13], which uses the state of the trailing thread to recover from an error. In [26] we proposed a modification to CRT that when combined with per-core DVFS reduced the energy consumption of the trailing core. Although this improves energy efficiency of redundant execution, it does not affect the throughput loss.

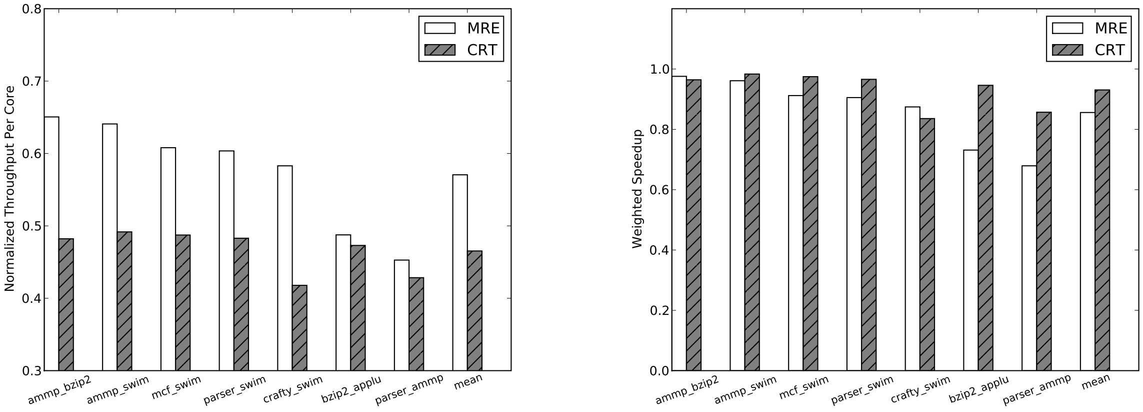
SRT and SRTR execute the leading and trailing threads on the same core. Hence, they can only provide transient fault coverage. CRT can provide both transient and permanent fault coverage. However our evaluation shows that MRE is superior to CRT in terms of both performance and energy characteristics even without multiplexing. With multiplexing, MRE increases throughput of a CMP significantly over CRT, and provides similar performance for individual programs. Furthermore, MRE can detect and recover from faults, while CRT can only detect faults. CRTR adds fault recovery to CRT by comparing outputs of all instructions at the end of dependence chains. CRTR has a higher performance overhead than CRT. As noted previously, MRE is faster than CRT, and hence CRTR.

SRT, CRT, SRTR, CRTR, Reunion etc. all use some form of redundant execution to detect errors. An alternate circuit level approach was introduced by Ernst et al. in [7]. Their approach, called Razor, is based on circuit level augmentation of a processor design to detect transient and wear-out related faults. Razor replicates critical pipeline registers and detect errors by comparing the values stored in them. The base assumption of a Razor-based design is that augmenting only a small number of time-critical paths of a circuit is sufficient to detect wear-out related errors. Recent work by Sartori et al. [23] has called into question the assumption that high-performance processor circuits contain only a small number of time-critical paths. They find that even for circuits which have only a small number of timing-critical paths, Razor may be ineffective if there are some short paths, leading to false positive error detections.

VI. Conclusion and Future Work

Decreasing feature sizes, lower design tolerances and higher operating temperatures have resulted in the emergence of wear-out related permanent faults, transient faults and process variations as significant concerns in modern microprocessors. Consequently, fault tolerance is expected to become important even for processors targeted at the commodity market.

A large number of existing proposals have attempted to take advantage of the natural coarse-grained multithreading offered by CMPs to provide fault tolerance. However these proposals typically use two cores or thread contexts to execute a single logical application, reducing the throughput of the system by half as compared to non-redundant execution. In this paper we introduced multiplexed redundant execution (MRE), an architecture that mitigates the throughput loss due to redundant execution. Our evaluation showed that in single-threaded workloads, MRE is able to provide transient and permanent fault detection and recovery with a performance overhead of only 2% and energy consumption that is similar to existing proposals for fault tolerance in CMPs. We also showed that for multiprogrammed workloads MRE increases the throughput of a CMP by a maximum of 30%, and a mean value of 16% as compared to an ideal DMR system.



(a) Normalized Throughput Per Core

(b) Weighted Speedup

Fig. 4. Comparison of MRE with CRT for a 2-program workload. MRE uses only 3 cores for the execution of two logical threads while CRT uses 4 cores.

In comparison to Mukherjee and Reinhardt's Chip level Redundant Threading (CRT) processors, MRE improves CMP throughput by 23%.

The results presented in this work provide an initial evaluation of the benefits of multiplexed execution using a static multiplexing scheme. A number of optimizations to MRE are possible. Two examples are: (1) adaptively migrating trailing threads across cores to reduce the performance losses due to multiplexing, and (2) dynamically turning off multiplexing for program phases where the trailing thread does not benefit from execution assistance. In future work, we intend to explore these ideas.

References

[1] Nidhi Aggarwal, Parthasarathy Ranganathan, Norman P. Jouppi, and James E. Smith. Configurable isolation: building high availability systems with commodity multi-core processors. *ISCA '07: Proc. of the 34th ISCA*, 2007.

[2] Todd Austin, V. Bertacco, S. Mahlke, and Yu Cao. Reliable Systems on Unreliable Fabrics. *IEEE Des. Test*, 25(4), 2008.

[3] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. Nonstop®advanced architecture. In *DSN '05: Proc. of DSN*, 2005.

[4] S. Y. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6), 2005.

[5] C. LaFrieda et al. Utilizing Dynamically Coupled Cores to Form a Resilient Chip Multiprocessor. In *Proceedings of the 37th DSN, Jun. 2007.*

[6] D. Brooks et al. Wattch: a framework for architectural-level power analysis and optimizations. *Proc. of the 27th ISCA, 2000.*

[7] D. Ernst et al. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO 36: Proc. of the 36th MICRO*, 2003.

[8] J. C. Smolens et al. Fingerprinting: Bounding soft error detection latency and bandwidth. *Proceedings of the 12th ASPLOS, Oct. 2004*, 2004.

[9] J. C. Smolens et al. Reunion: Complexity-Effective Multicore Redundancy. *Proceedings of the 39th MICRO, Dec. 2006*, 2006.

[10] Meyrem Kyrman et al. Cherry-MP: Correctly Integrating Checkpointed Early Resource Recycling in Chip Multiprocessors. In *MICRO 38: Proc. of the 38th MICRO*, pages 245–256, 2005.

[11] M.L. Fair, C.R. Conklin, S. B. Swaney, P. J. Meaney, W. J. Clarke, L. C. Alves, I. N. Modi, F. Freier, W. Fischer, and N. E. Weber. Reliability, Availability, and Serviceability (RAS) of the IBM eServer z990. *IBM Journal of Research and Development*, 2004.

[12] A. Golander, S. Weiss, and R. Ronen. DDMR: Dynamic and Scalable Dual Modular Redundancy with Short Validation Intervals. *IEEE Computer Architecture Letters, Jul-Dec. 2008*, 7(2).

[13] M. Gomma, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-Fault Recovery for Chip Multiprocessors. *Proceedings of the 30th ISCA, June 2003.*

[14] J. Renau et al. SESC Simulator. http://sesc.sourceforge.net/, 2005.

[15] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *IEEE Computer Architecture Letters, Jan. 2002.*

[16] B. Lee and B. Brooks. Effects of Pipeline Complexity on SMT/CMP Power-Performance Efficiency. *Workshop on Complexity Effective Design in conjunction with 32nd ISCA, Jun. 2005.*

[17] H. M. Mathis, A. E. Mericas, J. D. McCalpin, R. J. Eickemeyer, and S. R. Kunkel. Characterization of simultaneous multithreading (SMT) efficiency in POWER5. *IBM Journal of R&D, July/Sept. 2005.*

[18] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. *Proceedings of the 29th ISCA, May 2002.*

[19] Rahul Nagpal, Arvind Madan, Amrutur Bhardwaj, and Y. N. Srikant. INTACTE: An Interconnect Area, Delay, and Energy Estimation Tool For Microarchitectural Explorations. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2007.

[20] Michael D. Powell, Arijit Biswas, Shantanu Gupta, and Shubhendu S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *ISCA '09: Proc. of the 36th ISCA*, 2009.

[21] S. K. Reinhardt and S. S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. *Proceedings of the 27th ISCA, Jun. 2000.*

[22] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in a Microprocessor. *Proceedings of FTCS, 1999.*

[23] J. Sartori and Rakesh Kumar. Characterizing the Voltage Scaling Limitations of Razor-based Designs. *Workshop on Energy Effective Design*, 2009.

[24] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. *Proceedings of the 32nd DSN, Jun. 2002.*

[25] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proc. of 8th ASPLOS*, 2000.

[26] P. Subramanyan, V. Singh, K. K. Saluja, and E. Larsson. Power-efficient redundant execution for chip multiprocessors. *Proc. of 3rd WDSN*, 2009.

[27] S. Thoziyoor, N. Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. CACTI 5.1 . *Technical Report HPL-2008-20, HP Labs* .

[28] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-fault recovery using simultaneous multithreading. *SIGARCH Comput. Archit. News*, 30(2), 2002.