Redundancy Mining for Soft Error Detection in Multicore Processors

Ransford Hyman, Jr., *Student Member*, *IEEE*,
Koustav Bhattacharya, *Member*, *IEEE*, and Nagarajan Ranganathan, *Fellow*, *IEEE*

Abstract—The trends in technology scaling and the reduction in supply voltages have significantly improved the performance and energy consumption in modern microprocessors. Microprocessors are being built with higher degrees of spatial parallelism and deeper pipelines to improve performance, which, however, makes them more susceptible to transient faults. Radiation causes "transient faults" or "single-event transients" in logic, which, once propagated and latched, become full cycle errors or soft errors. If radiation hits memory elements, this is usually called an "single-event upset" or "soft error" as it can further propagate as a full cycle error. The problem of soft errors is further exacerbated in large multiprocessors employed in servers in which reliability is a key concern. In the past, the technique of lockstep execution of the original and the duplicate instructions has been used for error detection in multiprocessors. However, the execution of redundant threads in the on-chip multiprocessor (CMP) provides error detection at lower overheads, since the branch outcomes of the leading thread can be exploited during the execution of the trailing thread, and also because the interprocessor communication latency is a key concern for lockstepping. In this paper, we show that by mining various redundancies inherent within a single core, the interprocessor communication can be brought down to a minimum. Toward this, we propose techniques based on 1) temporal redundancy, 2) data value redundancy, and 3) information redundancy for error detection in multicore designs. We exploit temporal redundancy by using the "latency slack cycles" (LSC) of an instruction, which we define as the number of cycles before the computed result from the instruction becomes the source operand of a subsequent instruction. The value-based detection technique is explored by exploiting the width of the operands with small data values and information redundancy is exploited by the generation of residue code check bits for the source operands. We show that with a clustered core multiprocessor, the interprocessor communication overhead can be significantly reduced. In our proposed multicore design, when a soft error is detected, error correction is achieved by rolling back the execution to a previous checkpoint state and re-executing the instructions. The proposed techniques have been implemented on the RSIM simulation framework and validated using the SPLASH benchmarks. Experimental results indicate that the soft error detection schemes proposed in this work, can be implemented, on the average, with less than 10 percent increase in CPI on modern multicore designs.

Index Terms—Soft errors, multicore processors, fault tolerance.

1 Introduction

The trends in technology scaling have led to exponential growth in the number of on-chip transistors and significant reductions in the voltage levels of a chip. These trends for improving performance and power have made modern processors increasingly susceptible to transient faults. A majority of soft errors in modern processors are due to radiation induced transient faults. Radiation causes "transient faults" or "single-event transients" to occur in logic which, once propagated and latched, become full cycle errors or soft errors. If radiation hits memory elements, this is usually called "single-event upset" or "soft error" as it can further propagate as a full cycle error. Soft errors could occur when the energetic neutrons coming from space or the alpha particles arising out of packaging materials hit the transistors. With the shrinking of device geometries, the critical charge (Q_{crit}) required for the occurrence of soft errors, decreases. However, as the active silicon area of the cells also decreases due to scaling, the probability of radiation strike also decreases. Thus, the vulnerability of individual transistors due to cosmic ray strikes remains almost constant [33]. However, the decreasing voltage levels and the exponentially increasing transistor counts have caused the overall chip susceptibility to increase significantly.

Several different strategies have been investigated in the past to avoid, detect, and recover from soft errors [6]. These solutions are applied at various levels of the system, from process technology, to circuit to microarchitecture levels. The soft errors that do not affect the program output are considered benign as no error is observed by the user. This situation can occur, for example, in branch prediction logic or in the instructions from the mis-speculated execution sequences which never commit, and thus, will never lead to visible error states. Soft errors, which affect the program output, are typically defined in terms of *failures in time* (FIT) [5], [6]. The chip manufacturers typically set budgets on soft error rates which should be met by the design. A soft error may manifest itself as a bit flip in a latch or memory element. The problem of soft errors has been well known in static memory systems (SRAM) like caches. Several techniques have been proposed for reduction of soft error rates in caches. These include techniques like the use of

• *The authors are with the Department of Computer Science and Engineering, University of South Florida, Tampa, FL 33620. E-mail: rhyman@mail.usf.edu, {kbhattach, ranganat}@cse.usf.edu.*

Manuscript received 1 Sept. 2009; revised 9 Mar. 2010; accepted 25 May 2010; published online 25 June 2010.
Recommended for acceptance by R. Marculescu.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2009-09-0449.
Digital Object Identifier no. 10.1109/TC.2010.168.

error correction codes (ECC), parity bits, bit interleaving, and small-value duplication [6].

Soft errors can also occur in any internal node of a combinational logic, and subsequently propagate to and be captured in a latch. It is projected that technology trends like smaller feature size, lower voltage level, higher operating frequency, and reduced logic depth will increase the soft error rate (SER) in combinational logic [1]. Thus, the vulnerability of the combinational logic structures and the latches has made modern processor pipelines significantly susceptible to soft errors. Soft errors may cause corruption in data which may lead to incorrect addressing, faulty instruction execution, or generation of false exceptions. Various memory and latch elements within the processor pipelines, such as the issue queues [7], are individually protected against soft errors. In one of the earliest works in this direction, complete processor pipelines were duplicated [9] for detecting transient faults. In [10], "valid but idle" instructions were exploited for soft error reduction. In [11], the authors propose the design of a self-stabilizing processor for improving soft error vulnerability. The core design in [17] uses *state history signatures* (SHS) on memory and functional units for error detection. In [14], the authors have introduced memory based core design that uses the FRAM technology for immunity to soft errors. However, this architecture has a high memory cost and has limited functionality. In [8], the authors have used exceptions and incorrect control flow as "symptoms" for detection. However, since these symptoms are not known until the execution phase of the pipeline, it allows false positives to occur which can lead to huge performance costs. In [13], the idea of compiler directed instruction duplication has been studied. The approach, however, leads to large increase in code size and inefficient resource utilization.

Recently, the chip multiprocessors (CMPs) are becoming an attractive option to overcome the performance and power wall faced by the superscalar machines. CMPs are building blocks for machines used as servers, where reliability is an important concern. The problem of soft errors is critical in a large scale multiprocessor server, which requires even lower failure rates for the individual microprocessors. In [31], the tradeoffs among the soft error rate, power and performance metrics are analyzed for a reliable multiprocessor. A two-way CMP enables on-chip fault detection, using the concept of lockstep in which the same computation is performed on a cycle-by-cycle basis. Error detection and correction can be achieved, using simultaneous redundant threads (SRT) of execution on a SMT processor, through using leading and trailing threads separated by a slack [12]. The trailing thread uses the load (from memory) values and branch outcomes of the leading thread to avoid memory latencies and mispredicted computations. Memory corruption is avoided by only committing stores after comparison with the trailing threads. The register values may be committed before or after comparison of the trailing threads [19]. In [32], the approach is to selectively add redundant instructions on the low ILP phases and to extend instruction reuse for less than perfect error coverage at low performance overheads. The motivation behind scheduling and checking independent threads against lockstepping is due to that lockstepping uses hardware structures less efficiently than SRT. In lockstepping, both copies of a computation are forced to waste resources on mis-speculation and cache misses. In the chip-level redundancy technique (CRT), the SRT scheme is employed for error detection. Unlike as in SMT processors, reliable multicore processors are required to execute the redundant threads concurrently on *different* chips. Error detection is performed by checking and by comparing the results from the two threads. The primary concern in CRT has been to reduce the latency associated with comparing the results from the two executions and often, high-speed hardware structures are used for interchip communication. Although, techniques have been suggested to speed up the process of checking threads, such as checking them at the tail of dependence chains and judicious chaining of the masking instructions, the performance overhead of such schemes is still significant. Recently, in [16], the authors have proposed preliminary results on detecting soft errors in a multicore system by utilizing the inherent redundancy in these systems.

In this work, we show that lockstepping can be used quite effectively, if sufficient redundancy can be mined within each core's boundary. We have developed schemes that utilize the properties of temporal, data value, and information redundancies in programs for detection of soft errors. We propose the use of latency slack cycle (LSC) for error detection, using temporal redundancy. In our approach, we propose the mining of value-based redundancy in a processor core by utilizing the small data value width of the operands. Information redundancy is exploited by encoding the operand values with residue codes when possible. Furthermore, we show that the latency overhead associated with interprocessor communication can be significantly reduced when a clustered core microarchitecture is used. Experimental results indicate that our proposed schemes, on the average, can detect and correct soft errors in multicore systems with negligible area and performance overheads.

The rest of the paper is organized as follows: The architecture of a clustered core multiprocessor for soft error detection is described in Section 2. In Section 3, we discuss how temporal redundancy can be exploited for error detection. We explore both static and dynamic techniques for exploiting temporal redundancy. Section 4 describes how data-value-based redundancy can be exploited, using small value replication. Section 5 discusses error detection, using residue code by exploiting information redundancy. The overall architecture for error detection, using the proposed approach in a typical multiprocessor, is shown in Section 6. In Section 7, we present experimental results and compare our work with prior works. Finally, some conclusions are provided in Section 7 of the paper.

2 Clustered Multicore Architecture for Error Detection

In multicore processors, multiple threads can be mapped to multiple cores and executed concurrently to increase the system throughput. These threads can be separate, independent applications or independent pieces of the same thread from a given application. Redundant execution of threads

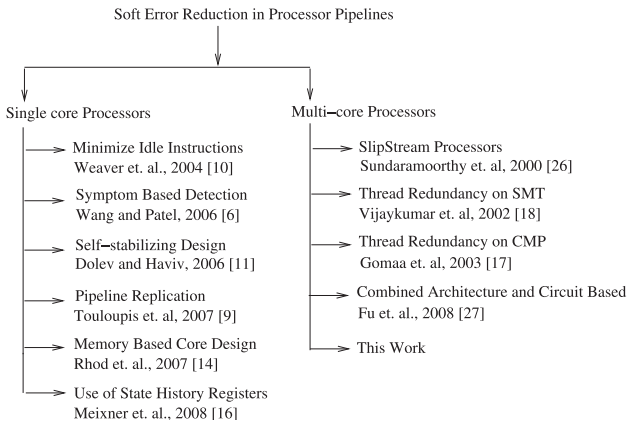

Fig. 1. Taxonomy of related work.

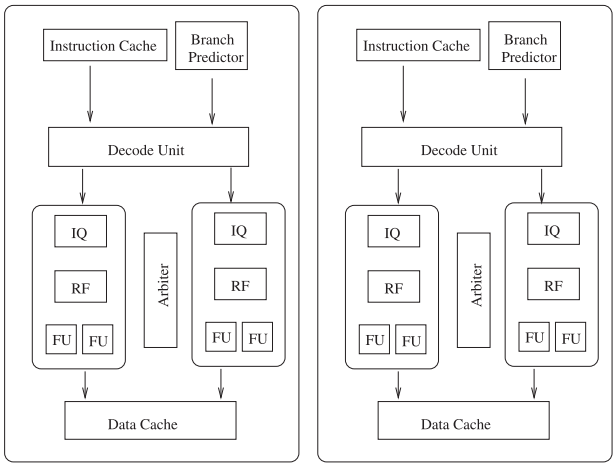

Fig. 2. Hardware architecture for error detection in a clustered core processor.

and comparing the results of their multiple executions can be effective toward reduction of soft errors. However, two problems seriously limit this approach. First, the interprocessor communication latency can be significant. The wire delays due to interprocessor communication can impose significant latency overhead, which cannot be hidden in the typical complete-to-commit times, if all register and memory accesses have to be checked. Second, due to the nondeterministic ordering of communication events, threaded applications can produce outputs in different orders and of different values when given the same inputs over consecutive runs [30]. This phenomenon can make error detection by dedicated execution of the original and the duplicate threads on different processors in a CMP ineffective or not beneficial. On the other hand, forcing deterministic execution for replication by lock-stepping will have a high performance cost.

We use a clustered multiprocessor architecture to reduce the communication latency due to interprocessor communication. Further, we show that by mining of available temporal and data-value redundancy and by exploiting the information redundancy, the interprocessor communication required in a fault-tolerant CMP can be brought down significantly. In this section, we discuss the clustered core processor architecture for low overhead error detection in a CMP. The technique to mine available redundancy is described in the following sections.

The clustered microarchitecture for a large multicore processor is divided into multiple clusters. Each cluster is small so that if the latency overhead for interprocessor communication can either be hidden completely by complete-to-commit times or is negligible in terms of impacting overall performance. Unlike a regular homogeneous multicore processor, this architecture consists of multiple clustered cores rather than monolithic ones, and each core is based on the clustered microarchitecture. Fig. 2 illustrates the typical architecture of a multiple clustered core multiprocessor with two homogeneous clustered cores, each having two identical clusters. Each cluster consists of register files (RF), instruction scheduling queue (IQ), and functional units (FU), while instruction and data caches, branch predictor and decoder are shared by all clusters in a core. Each cluster shares an instruction arbiter for interprocessor communication. The overall architecture of our clustered core processor is shown in Fig. 2. The detailed architecture of the instruction arbiter is described in the following section. A multiple clustered core multiprocessor with two cores in each cluster can be effectively used for error detection. Since cores in a cluster are close in proximity and the latency overhead for interprocessor communication can be minimal. It is possible to exploit the clustered microarchitecture combined with multicore architecture to tradeoff between power and performance, which is not considered in this work.

3 Mining Temporal Redundancy

It is generally not possible to detect a soft error in the processor pipeline before the instruction is executed at least once. Also, to achieve a low latency overhead, error detection must be done concurrently during execution of each instruction. Concurrent error detection can be achieved by complete duplication of the processor pipeline and issuing duplicate copies of each instruction. A soft error is detected when the result of the instruction and its duplicate instruction do not match. However, such an approach would be prohibitive in terms of area and power overheads. In this section, we propose a low overhead soft error detection technique by exploiting temporal redundancy. Temporal redundancy can be exploited by using compiler hints or by using a specialized predictor hardware. As discussed later, temporal redundancy in existing programs can be mined with a low overhead.

3.1 Compiler-Directed Slack Computation

Temporal redundancy can be exploited by duplicating and re-executing instructions in a different time slot in the same processor. We define the *latency slack cycle* (LSC) value of an instruction as the amount of cycles between the current instruction and the next true data-dependent instruction. The LSC values can be determined statically during code generation in the compiling phase of a program. Given a fixed threshold (MIN_LSC), the compiler probes the nearest data dependency on any of the source operands and checks to see if the number of instructions between them is lower than the MIN_LSC. If this condition is met, then a special bit called the *slack bit* is set. This bit can be checked during the

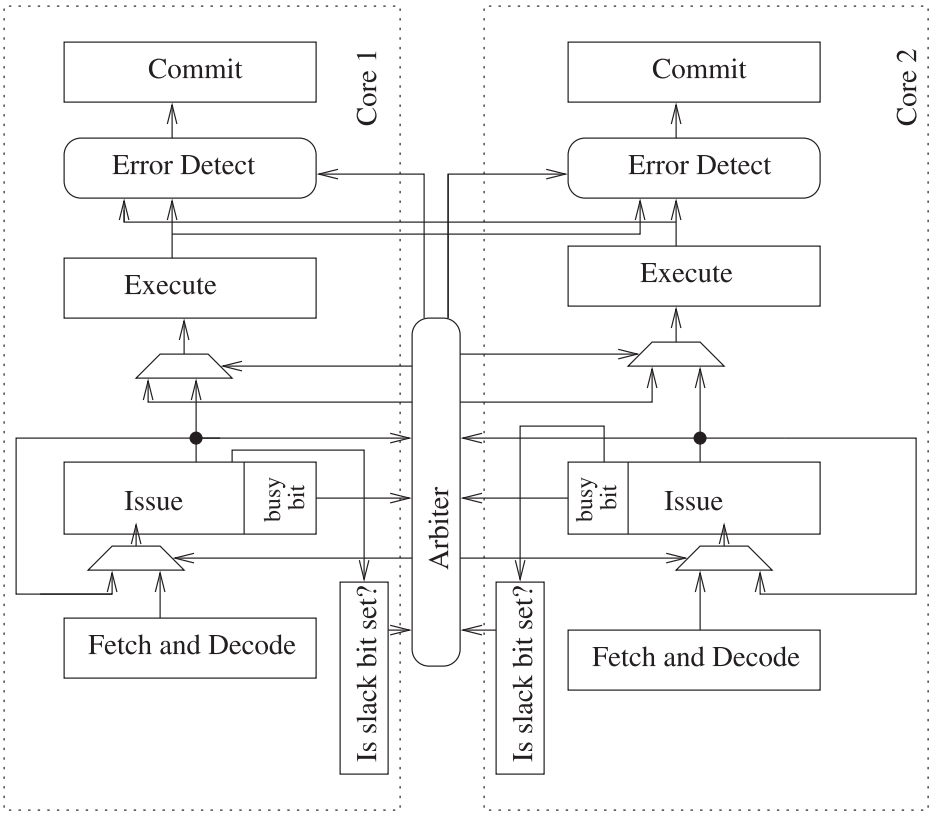

Fig. 3. Instruction arbiter for temporal redundancy-based detection.

issue stage of the processor pipeline. The value MIN_LSC should be chosen in a manner such that the majority of the instructions should be able to make use of.

Given the slack bit, temporal redundancy is exploited for error detection by re-executing the instruction within MIN_LSC cycles. Once both instructions have been executed, comparison can be made to detect any transient faults that may have occurred. We use the slack information encoded in the instruction itself to determine if the results of the instruction are not immediately needed. The duplicate instruction can be executed following in time using a temporal redundancy approach.

The use of temporal redundancy for soft error detection is illustrated using Fig. 3. A special *busy bit* is used, which simply indicates if any of the issue slots are free in the current cycle. We also use a special hardware structure called an instruction arbiter, which is a simple state-machine, and a special error-detection stage in the processor pipeline between the execute stage and the commit stage, shown as the boxes with chamfered edges, to indicate the presence of internal buffers for storage. The sequence of steps for error detection is as follows: The slack bit is checked for each instruction that is issued. If it is set, the arbiter initializes itself for error detection using temporal redundancy and guides the multiplexor (MUX) at the input of the issue stage to issue the duplicate instruction within MIN_LSC cycles, depending on the status of the busy bit in the issue stage. Since the arbiter knows *exactly* when the original and the duplicate instructions are issued and as the duplicate instruction does not wait in the issue queue due to data dependency, the arbiter can set the control signals at the appropriate clock cycle to latch the result from the execution of the original and duplicate instructions and compare them for error detection.

3.2 Dynamic Slack Implementation

In the previous section, we described a technique to statically estimate LSC, using compile time analysis. The technique, thus, assumes that compiler performs additional tasks and encodes slack information in the form of the slack bit in the instruction. This technique, thus, can lead to increase in binary code size. Moreover, as the compiler-directed technique only decides slack by analyzing instructions statically within a window, the slack estimate is not accurate. In this section, we propose a technique that exploits the temporal redundancy in a dynamic manner during execution.

We have developed a scheme for predicting instruction slack dynamically during execution. The technique is based on a *slack prediction* scheme and is illustrated in Fig. 4. As shown in the figure, the slack predictor is a simple, state-based cache structure. Each entry in the array type structure stores an operation field, a destination register field, a slot to indicate an execution start time, and a user-defined number of voter slots. The slots indicate a vote on various binned predicted slack values. The number of slack slots for each cache entry depends on the granularity of binning of the predicted slack and its range. For example, if the instruction slack varies between two and six, with a step size granularity of two, there will be three slots for the corresponding cache entry. The three slots correspond to the slack values of two, four, and six. The vote on each

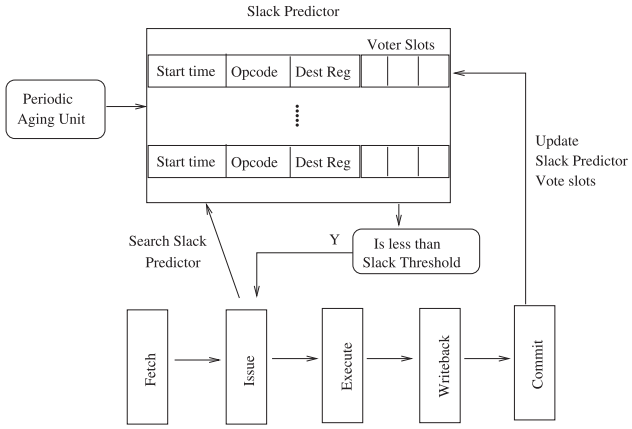

Fig. 4. Slack predictor hardware for dynamic temporal redundancy-based detection.

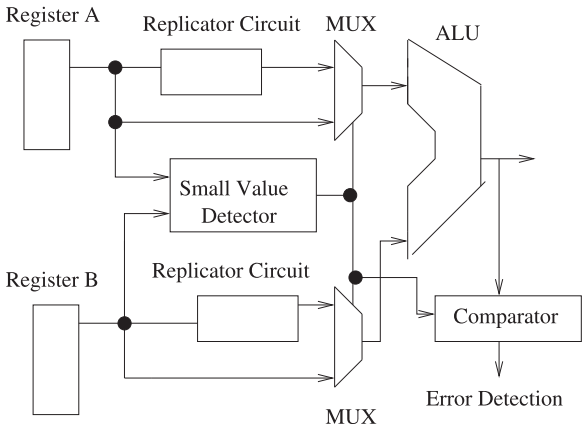

Fig. 5. Error detection using small values.

slack value slot indicates a confidence level on the amount of times the predicted slack was indeed correct.

We illustrate the hardware architecture for dynamic slack prediction in Fig. 4. The destination register of an executed instruction is noted in the destination register field of a free cache entry. It should be noted that this register number is the register address after register remapping, which is available at the decode stage. When the corresponding instruction starts execution, we also time stamp the execution start time slot. During this initialization, the votes associated with the cache entry are also set to zero. For each instruction, the slack predictor cache is also checked to see if the source registers for that instruction matches any destination register in the slack predictor for any previous operation. If such a match occurs, the current time step is again noted, and the difference between the start time stamp is binned and the vote for the corresponding predicted slack slot is increased by one. Thus, for each instruction, the destination register and the opcode, and the execution start time stamp are noted in an available cache entry, as well as the source registers are checked to find if they match any previous destination register entries for any operation. In case of a match, the slack value with maximum votes is selected as the predicted slack for this instruction. The slack votes are gradually aged to a saturating zero at regular intervals of time. The LRU replacement policy is used for the individual entries of the slack predictor. The dynamic slack prediction scheme thus mimics a simple voting scheme. A correct prediction means that the actual slack will be truly higher than the predicted slack, and the duplicated instruction can be executed without any performance penalty. In this case, the next data-dependent instruction still executes at the same time as that without instruction duplication.

4 Mining Data Value Redundancy

It is commonly known that a large percentage of memory values are small [23], [22], [24]. Typically, small memory values use at the most half of the bit width of the registers. These small data values can be exploited to increase redundancy and improve the reliability of the processor pipeline. Small value replication is carried out for instructions with operand values that can be represented with half of the bit width size [22]. Small value-based error detection is carried out differently on different operations. For example, if both source registers meet this requirement then their values are replicated and executed on the same functional unit. Note that before the execute stage of the pipeline, the data values of the source operands are already available. Thus, source registers with small data value width can be locally duplicated and both copies of the small data value can be executed on the same functional unit and compared for error detection. If the results are correct, the upper half words are filled with 0s or 1s to obtain the actual results.

Soft errors during execution of arithmetic operations like addition and subtraction can also be detected even if one operand is of small data value, by using value-based redundancy. For example, consider the case when one operand is small and the arithmetic operation on the other operand does not generate any carries. In this case, the upper half word of the other operand can be stored in a register and the small value operand in the lower half word of the other operand can be locally duplicated as before. Again, both copies of the small data value operands can be executed on the same functional unit and compared for error detection. If the results from the upper and lower half words agree, the upper half word of the other operand is concatenated with the lower half word of the result to form the actual result. Thus, in the absence of a carry, local duplication can be applied to detect errors in the execution of arithmetic instructions even involving only one small operand. The same idea can also be extended for logical operations. For example, for a AND or OR operation if only one operand is small, local duplication can be applied to perform duplicate execution of the operation and detection of errors. We can recover the upper half word of the result of the logical operation by filling them with all zeros, all ones, or with the upper half word of the other operand.

Fig. 5 illustrates the hardware architecture for small value detection method. The source operand values stored in the registers are sent to the *small value detector* (SVD) to determine whether their data values are small. The SVD circuit is simply a zero detector for the MSB portion of the data bits. If a small value is detected, the SVD sends the signals to the multiplexors so that the replicated values are chosen. The two duplicate values are then sent to the ALU for execution. The results of the ALU operations are compared with a comparator that is controlled by the

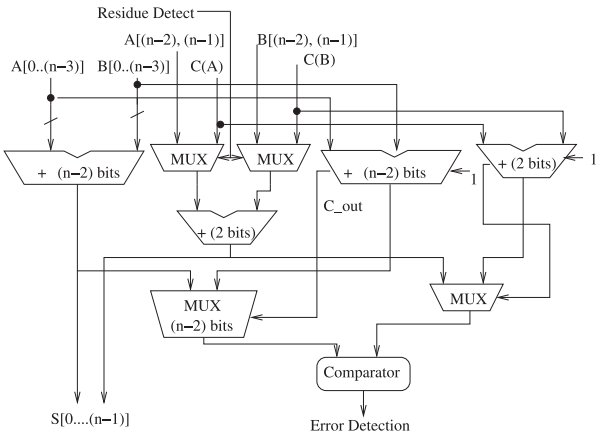

Fig. 6. Error detection using residue code.

SVD circuit. The SVD is checked to see if it indicates a small value when the MSB and the LSB portion of the result should match. However, if a small value is indicated by the SVD and LSB, and MSB portion of result does not match a soft error is detected and recovery by rollback from the previous checkpointed state can be initiated. Thus, as shown in Fig. 5, error detection by exploiting small data value size can be implemented without much latency and area costs. In contrast to [22] our technique detects the small register data values and copies their values before execution on the functional unit, and our soft error detection is carried out immediately after execution before the register values are latched into the next pipeline stage. Thus our duplication and soft error detection method are carried out within the same pipeline cycle, which is not the case in [22]. In addition, our method does not require any extension to the bit widths of the data bus or register files.

5 Information-Redundancy-Based Detection

In general, parity codes can be used to detect single bit soft errors in memory; however, parity check bits are not preserved across arithmetic operations. To maintain check bits across arithmetic operations, residue codes have been proposed in the literature. Residue codes have the desirable property that for arithmetic operations the check bits of the result can be determined directly from the check bits of the operands. Residue code is a systematic code, i.e., the check bits can be represented separately from the data bits. The residue check bits C are represented by $log_2 N$ bits, where N is the total number of bits used for representation and can be computed as $C = (N) mod (m)$, where m represents the residue of the code. It can be shown that for a pair of operands, the residue code for the sum (or product) of the operand is the sum (product) of the residue codes of the operands modulo m, and that if the residue m is odd, all single-bit errors occurring during any arithmetic operations can be detected.

As shown in Fig. 6, error detection using residue code requires only an extra adder unit. However, as the extra addition operation is off the critical path, and since we can neglect the delay due to the multiplexors, no performance penalty will be incurred. If the residue check bits of the sum (product) do not match the sum (product) of the check bits of the operands, a single bit error is detected. Otherwise, no single-bit error has occurred and the result is committed to update the processor state. The determination of the residue code is dominated by the computation of modulo addition. However, the operation can be simplified if the modulo is of the form $2^n - 1$, as follows:

$$\begin{aligned} &(x+y) mod (2^n-1) = (x+y+1) mod (2^n), \\ &if\ (x+y+1) \geq 2^n, \\ &(x+y) mod (2^n-1) = (x+y) mod (2^n), \qquad otherwise, \end{aligned} \tag{1}$$

where x and y represent the source operands and n represents the bit width of the registers used to store the source operands. In our work, we exploit this property and use the two most significant bits for representation of the residue parameter m.

6 Proposed Multicore Architecture

In the previous sections, we described several low overhead soft error detection techniques, which exploit the various types of redundancy that can be mined within the boundary of the single core. Error detection is performed during the issue stage of each core's processor pipeline for each instruction. The error-detection mechanism, in order to maintain low overheads, makes use of the available functional resources as well as the remaining latency slack cycles (LSC), which is the number of cycles before the computed result becomes the source operand of a subsequent instruction. For instructions with high LSC, the soft errors are detected using temporal redundancy wherein the duplicate instruction is executed at a later time step and compared with the result of the original instruction, ensuring that there is no loss in performance. For instructions with small operand value size, soft errors are detected by duplicating the operands and executing on the same functional unit, thus using the available resources to maintain low overhead. For instructions with low LSC and not having small valued operands, detection is implemented by using the residue code. If none of these methods are applicable, the duplicate instruction is executed on a nearby idle processor core. The overhead associated with interprocessor communication is reduced, using a multiple clustered core architecture. Finally, correction is done by recovery and rollback from the checkpoint states. It should be noted, however, that error correction is required only when a soft error has actually happened, which is rare, while error detection needs to be performed during execution of each instruction. Therefore, throughout in this paper, we have focused on providing techniques for low overhead error detection in multicore processors.

We described previously, how the slack bit is used in an instruction arbiter for error detection using temporal redundancy. In the case when the slack bit is not set, the arbiter initializes itself for error detection by using spatial redundancy. The arbiter latches the instruction and its source operand values from the original core, keeps on polling the busy bit of the issue stage of the nearby core, and once the busy bit is disabled the duplicate instruction is issued to the adjacent core. The arbiter then sets the control signals at the appropriate clock cycles in the error-detect stage to latch the results of the execution from the original core and the results of execution from the nearby core, which are then compared

TABLE 1
Processor Configurations

Parameter	Configuration
Active list	64 instructions
fetch and commit rate	4 per cycle
Functional units	3 ALUs, 2 FPUs, 2 Addr. gen. units
Branch predictor	2-bit history predictor
Cache Line Size	64 Bytes
L1 Cache	32KB 8-way associative, Write Back
L1 Cache Latency	6 cycles
L2 Cache	64KB 8-way associative
L2 Cache Latency	16 cycles

for error detection. This is possible, as the arbiter knows exactly when issue starts and execution finishes for the original and duplicate instructions. We note that many variants of this idea have been proposed [20], [21] in the literature. Our experiments with redundancy-based detection technique suggested that the scheme of spatial redundancy incurs a considerable latency overhead due to the interconnect delay in sending/receiving the instruction to the arbiter. In general, using the spatial redundancy-based detection can lead to high performance overhead, and about 200 percent increase in power. However, by mining of various redundancy schemes and using the clustered core microarchitecture, this scheme is seldom used for error detection. The overall hardware algorithm for error detection in a CMP processor using our proposed schemes is shown in Algorithm 1.

Algorithm 1. The algorithm for soft error detection
Compute LSC value for all ALU instructions statically or dynamically
for Each ALU instruction **do**
if $slack > S_{Th}$ **then**
Detect soft error using temporal redundancy
end if
if $slack < S_{Th}$ AND Both source operands are small **then**
Duplicate data values and execute on the same functional unit
Detect soft error if the LSB and the MSB of the result do not match
end if
if $slack < S_{Th}$ AND Both source operands are not small AND Two upper bits of the source operands are zero **then**
Compute residue code for the operands
Replace the top 2 bits of the operand with check bits
Execute the operation and compute modulus m
Compute the check bits of the data without the top two bits
Compare this check code with the top two bit and detect soft error if they do not match
else
Duplicate the instruction into a nearby idle core
Commit from ROB only when both original and duplicate instruction's result match
end if
end for

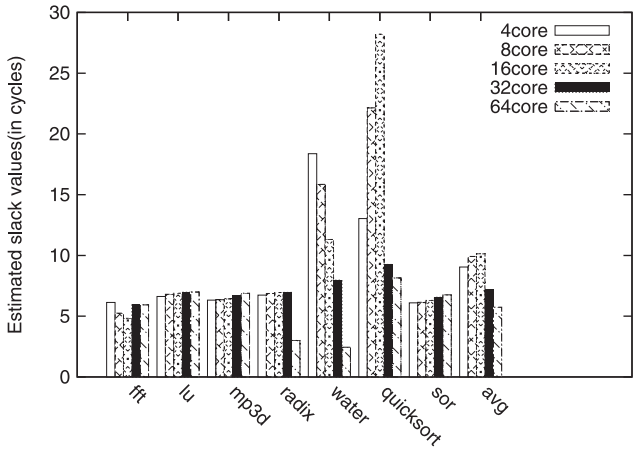

Fig. 7. Static slack breakdown.

Our error-recovery mechanism achieves error correction, using traditional rollback and recovery scheme. Thus, a check-point state of the processor after execution of each instruction is maintained. Such fine-level checkpointing can easily be achieved by using a structure similar to a reorder buffer. Instructions are committed from the reorder buffer only if no error has been detected. Otherwise, we flush all instructions in the reorder buffer and re-execute instructions from our last checkpoint. Thus, we leverage the hardware structures already present in current processors for speculative execution for error correction.

7 Experimental Setup

In this section, we describe the experimental setup used for simulations and the results of the various schemes proposed in the paper for improving the reliability against soft errors for multicore systems. We modified the RSIM multi-multiprocessor simulator [25] and used for this study. RSIM is an execution-driven simulator primarily designed to study shared memory multiprocessors and can simulate applications compiled and linked for SPARC V9/Solaris. Our base specifications are given in Table 1. Each core uses static scheduling, has an issue width of 4, with 3 integer and 2 FP ALUs, with an active list of 64 entries and used a 2-bit branch predictor. The local L1 cache is 32KB 8-way associative while the shared L2 cache is 64KB 8-way associative with cache access latencies selected as given in [26]. Our soft error detection architecture follows the procedure presented in Algorithm 1. Both redundancy-based and information-dependent methods can be determined during runtime by checking the source operands during the decode stage of RSIM. The simulation results are stored in a well defined data structure so that they can be examined using Perl scripts after runtime. Our simulations were performed on a subset of the SPLASH benchmark suite [27] that were run on a Sun Blade 1500 uniprocessor with 4 GB of RAM. Within a superscalar processor, the performance overhead of temporal redundancy due to redundant instructions delaying independent instructions is negligible. We, therefore, calculate the performance overhead of our proposed schemes as follows:

$$overhead_{perf} = I_{spatial} \times cost_{spatial,} \quad (2)$$

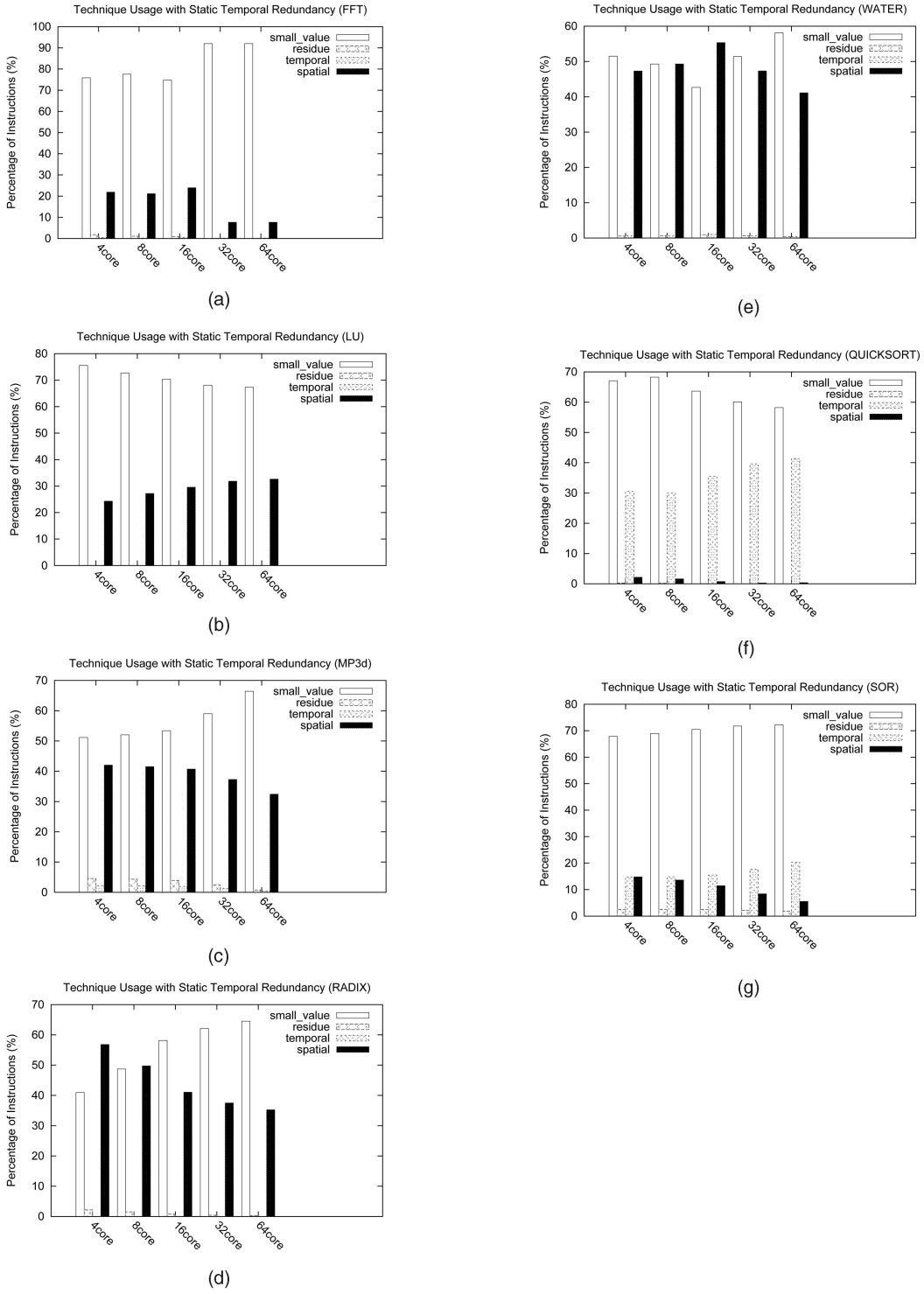

Fig. 8. Results from static temporal redundancy technique. (a) Static temporal redundancy SPLASH-FFT. (b) Static temporal redundancy SPLASH-LU. (c) Static temporal redundancy SPLASH-Mp3d. (d) Static temporal redundancy SPLASH-Radix. (e) Static temporal redundancy SPLASH-Water. (f) Static temporal redundancy SPLASH-Quicksort. (g) Static temporal redundancy SPLASH-SOR.

where $I_{spatial}$ is the number of instructions that are forced to use the spatial redundancy technique, and $cost_{spatial}$ is the amount of additional cycles it takes to execute the replicated instruction in the neighboring core. Power calculations are given for the 16-core processor, using the Nangate FreePDK 45 nm Open Cell Library. Power consumption values for the various techniques were calculated using the product between the percentage of usage and the power consumption of the functional units used.

8 Simulation Results

Fig. 7 shows the average slack values in cycles for each benchmark for the implementation of static temporal redundancy. Since our static temporal redundancy technique

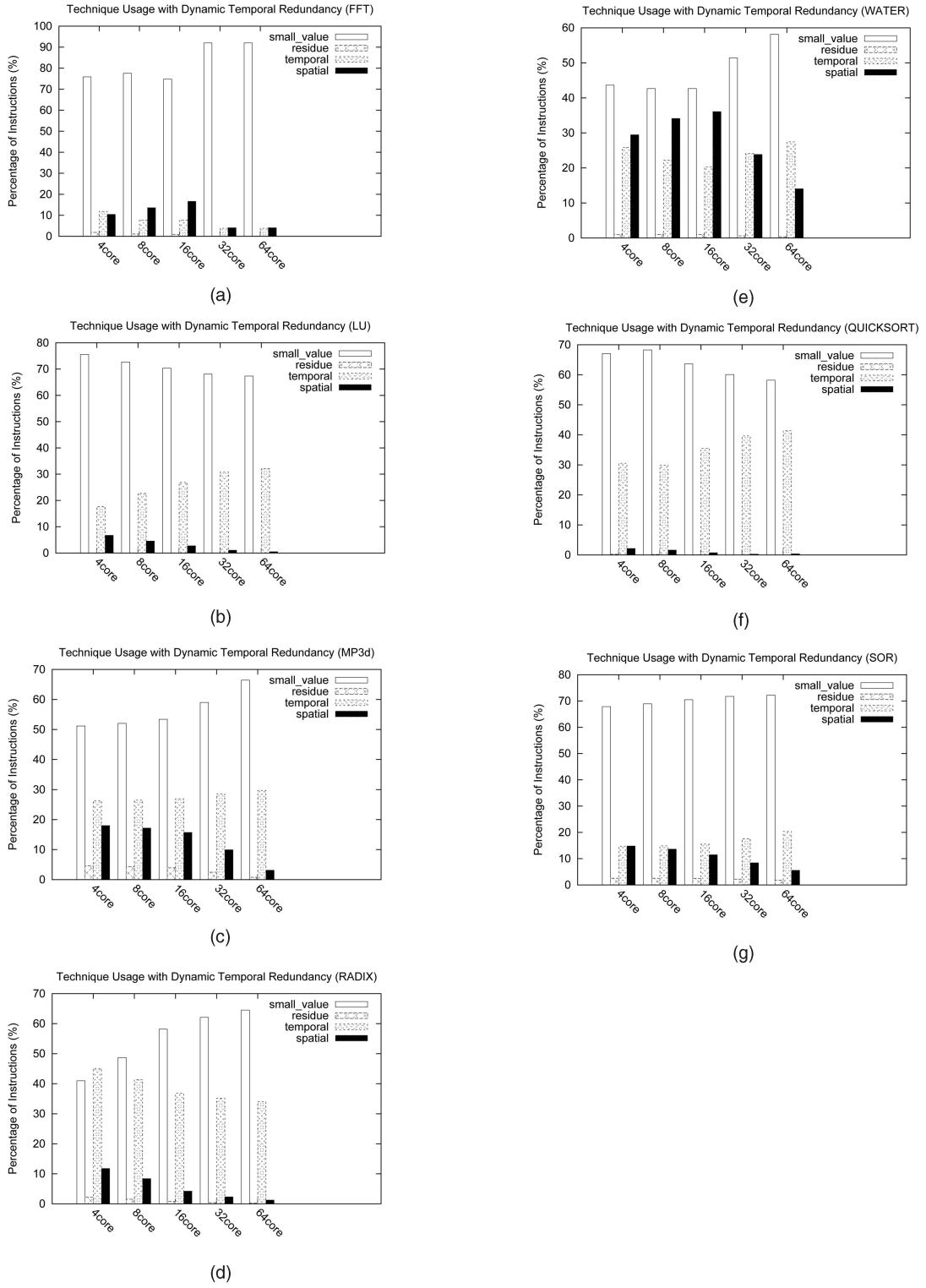

Fig. 9. Results from dynamic temporal redundancy technique. (a) Dynamic temporal redundancy SPLASH-FFT. (b) Dynamic temporal redundancy SPLASH-LU. (c) Dynamic temporal redundancy SPLASH-Mp3d. (d) Dynamic temporal redundancy SPLASH-Radix. (e) Dynamic temporal redundancy SPLASH-Water. (f) Dynamic temporal redundancy SPLASH-Quicksort. (g) Dynamic temporal redundancy SPLASH-SOR.

is implemented on the compiler level, we assume single-issue width so that the technique is architecture independent. From the results given in column *avg*, it is clear that the appropriate LSC value to use is 5. Note that the MIN_LSC value may vary from different applications but the value of five suits best for our set of benchmarks. The results from our static slack implementation of *temporal redundancy* are given in Fig. 8. The majority of the benchmarks had less than 30 percent of their instructions using the spatial redundancy technique. The benchmarks, *quicksort* and *sor* were the most efficient with respect to the usage of static temporal redundancy technique. The other benchmarks either had

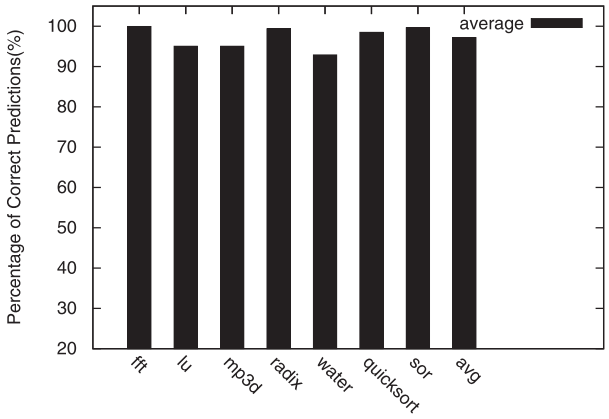

Fig. 10. Dynamic prediction technique.

high usage of data-dependent techniques or the MIN_LSC value may have been too large, thus giving us a low percentage usage for our static temporal redundancy method. To further lessen the amount of spatial redundant instructions, in our dynamic slack technique, we use one, two, and four cycles as our voting choices for the dynamic slack implementation. The results with the dynamic slack implementation are given in Fig. 9. As illustrated in all benchmarks, more instructions are able to implement the temporal redundancy method, thus lowering spatial redundancy usage as well as the performance overhead.

For the correct dynamic prediction rate and power overhead, we only analyze the 16-core configuration. We can see from Fig. 10 that most predictions were correctly made for our benchmarks. Fig. 11 shows that our technique generates very little power overhead. We have designed the instruction arbiter at the RTL level using verilog and synthesized it with a standard cell library of 45nm technology using Design Compiler. Our small value based detection scheme only required small number of MUXes while residue code based detection required an extra n-bit adder. Our calculations indicate that the overall area overhead for our combined error detection framework was less than 5 percent. In Fig. 12, we plot the percentage overhead in CPI compared to a multiprocessor with no error-detection capability for 8, 16, 64, and 128 cores. As shown, the performance overhead was only eight percent for the 8 and 16 core multiprocessors, while it was 10.5 percent for the 64-core system and

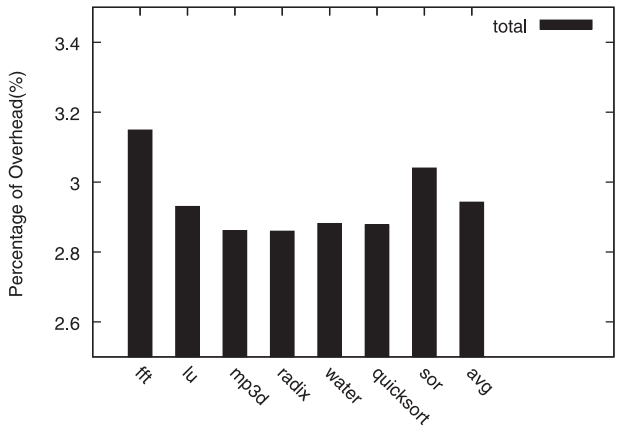

Fig. 11. Power results.

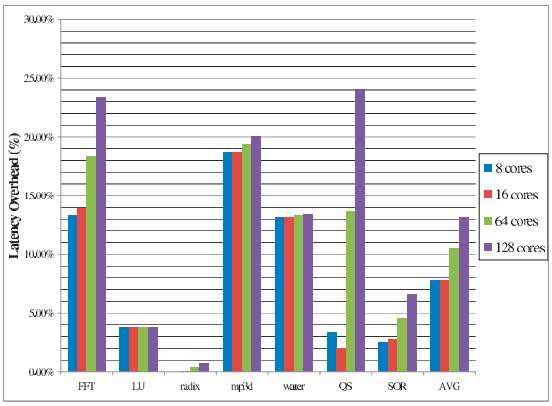

Fig. 12. Performance overhead for the proposed schemes.

13 percent for 128-core multiprocessor system. We noted that the large increase in average CPI is partly due to the large amount of spatial redundancy instructions in the mp3d benchmark, which, if excluded, will lead to even a lower performance overhead.

In Table 2, we briefly compare our work with some recent works provided in literature, using the data provided in the papers and extrapolating according to our experimental setup. SRT technique [28] executes redundant threads on a SMT processor for error detection. The CRT technique extends the same idea on CMP processors. The CRT approaches [18] achieve considerably low-latency overhead compared to instruction lockstepping in CMP processors. This is due to the fact that the trailing thread can effectively utilize speculative outcomes for a speedier execution. However, the trailing thread may read values which can be modified by other instructions in the leading thread, and hence, false positives may occur. The problem is circumvented by using additional hardware structures like the load value queue and by ensuring stores are committed only after the stores from the leading thread finishes execution, thus not changing memory state before checking. Further, as relative thread execution rates on different processors are nondeterministic, events among concurrent threads in a program cannot be replicated precisely and efficiently, leading to spurious divergences [30]. We have shown in this work that by mining and utilizing various redundancy mechanisms available within a core's boundary and using a clustered core architecture, interprocessor communication overhead can be made negligible. Using the various techniques proposed in this paper, duplication on a per-instruction basis is more attractive than duplicating at the thread level. Slipstream processors proposed in [29] use a reduced instruction stream in the leading thread to guide the trailing thread. As

TABLE 2
Comparison with Related Works

Approach	Error Coverage	Latency Overhead
RMT [28]	100%	21%
CRT [18]	100%	15%
Slipstream [29]	74%	-7%
This work	100%	less than 10%

the leading thread is reduced in size and as it can provide key speculation outcomes to the trailing thread, this method can actually improve performance than executing the single thread with speculation. However, compared to our approach and the CRT approaches, the error detection coverage is not high, and hence, is not viable in server application where reliability can be a key concern. Thus, compared to other works proposed in literature, our combined framework can achieve complete error coverage at significantly lower latency overheads with negligible area cost.

9 Conclusion

In this paper, we have investigated several ways to detect and correct soft errors in multicores with negligible overheads in latency, area, and power using techniques like mining temporal and data value dependency and inserting information redundancy. Temporal redundancy has been exploited by static-compiler-directed slack computation and dynamically using an efficient slack predictor hardware. Small data value widths are mined to exploit data value redundancy. Information redundancy is supported by using an efficient implementation of residue codes. We also propose a cluster core architecture to reduce the latency overhead for interprocessor communication. Our results indicate that our combined framework can achieve complete error coverage with significantly less overheads than other works existing in literature.

References

[1] 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," *Proc. Int'l Conf. Dependable Systems and Networks (DSN),* pp. 389-398, 2002.

[2] K. Meng, F. Huebbers, R. Joseph, and Y. Ismail, "Modeling and Characterizing Power Variability in Multicore Architectures," *Proc. Int'l Symp. Performance Analysis of Systems and Software (ISPASS),* pp. 146-157, 2007.

[3] C. Slayman, "Cache and Memory Error Detection, Correction, and Reduction Techniques for Terrestrial Servers and Workstations," *IEEE Trans. Device and Materials Reliability,* vol. 5, no. 3, pp. 397-404, Sept. 2005.

[4] K. Bhattacharya, S. Kim, and N. Ranganathan, "Improving the Reliability of On-Chip L2 Cache Using Redundancy," *Proc. Int'l Conf. Computer Design (ICCD),* pp. 224-229, 2007.

[5] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. Mukherjee, and R. Rangan, "Computing Architectural Vulnerability Factors for Address-Based Structures," *Proc. Int'l Symp. Computer Architecture (ISCA),* pp. 532-543, 2005.

[6] H. Asadi, V. Sridharan, M. Tahoori, and D. Kaeli, "Vulnerability Analysis of L2 Cache Elements to Single Event Upsets," *Proc. Design Automation and Test in Europe (DATE) Conf.,* vol. 1, pp. 1-6, 2006.

[7] X. Fu, W. Zhang, T. Li, and J. Fortes, "Optimizing Issue Queue Reliability to Soft Errors on Simultaneous Multithreaded Architectures," *Proc. Int'l. Conf. Parallel Processing,* pp. 190-197, 2008.

[8] N.J. Wang and S.J. Patel, "Restore: Symptom-Based Soft Error Detection in Microprocessors," *IEEE Trans. Dependable and Secure Computing,* vol. 3, no. 3, pp. 188-201, July/Sept. 2006.

[9] E. Touloupis, J. Flint, V. Chouliaras, and D. Ward, "Study of the Effects of SEU-Induced Faults on a Pipeline Protected Microprocessor," *IEEE Trans. Computers,* vol. 56, no. 12, pp. 1585-1596, Dec. 2007.

[10] C. Weaver, J. Emer, S. Mukherjee, and S. Reinhardt, "Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor," *Proc. Int'l Symp. Computer Architecture (ISCA),* pp. 264-275, 2004.

[11] S. Dolev and Y.A. Haviv, "Self-Stabilizing Microprocessor: Analyzing and Overcoming Soft Errors," *IEEE Trans. Computers,* vol. 55, no. 4, pp. 385-399, Apr. 2006.

[12] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," *Proc. Int'l Symp. Fault-Tolerant Computing,* pp. 84-91, 1999.

[13] J.S. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M.J. Irwin, "Compiler-Directed Instruction Duplication for Soft Error Detection," *Proc. Design, Automation and Test in Europe (DATE '05) Conf.,* pp. 1056-1057, 2005.

[14] E. Rhod, C. Lisboa, and L. Carro, "A Low-SER Efficient Core Processor Architecture for Future Technologies," *Proc. Design, Automation and Test in Europe (DATE '07) Conf.,* pp. 1-6, 2007.

[15] T. Sato and T. Funaki, "Power-Performance Trade-Off of a Dependable Multicore Processor," *Proc. Int'l. Symp. Dependable Computing,* pp. 268-273, 2007.

[16] R. Hyman, Jr., K. Bhattacharya, and N. Ranganathan, "A Strategy for Soft Error Reduction in Multi-Core Designs," *Proc. Int'l. Symp. Circuits and Systems (ISCAS),* pp. 2217-2220, 2009.

[17] A. Meixner, M.E. Bauer, and D.J. Sorin, "ARGUS: Low-Cost, Comprehensive Error Detection in Simple Cores," *IEEE Micro,* vol. 28, no. 1, pp. 52-59, Jan./Feb. 2008.

[18] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz, "Transient-Fault Recovery for Chip Multiprocessors," *Proc. 30th IEEE Ann. Symp. Computer Architecture,* pp. 98-109, 2003.

[19] T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-Fault Recovery Using Simultaneous Multithreading," *Proc. Int'l Symp. Computer Architecture (ISCA),* pp. 87-98, 2002.

[20] G.B. Bell and M.H. Lipasti, "Skewed Redundancy," *Proc. Int'l. Conf. Parallel Architectures and Compilation Techniques,* pp. 62-71, 2008.

[21] J.C. Smolens, B.T. Gold, B. Falsafi, and J.C. Hoe, "Reunion: Complexity-Effective Multicore Redundancy," *Proc. Int'l. Symp. Microarchitecture,* pp. 223-234, 2006.

[22] J. Hu, S. Wang, and S. Ziavras, "In-Register Duplication: Exploiting Narrow-Width Value for Improving Register File Reliability," *Proc. Int'l. Symp. Microarchitecture,* pp. 281-290, 2006.

[23] D. Brooks and M. Martonosi, "Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance," *Proc. Int'l Symp. High Performance Computer Architecture (HPCA),* pp. 13-22, 1999.

[24] H. Lee, G. Tyson, and M. Farrens, "Eager Writeback—a Technique for Improving Bandwidth Utilization," *Proc. Symp. Microarchitecture,* pp. 11-21, 2000.

[25] V.S. Pai, P. Ranganathan, and S.V. Adve, "RSIM: Rice Simulator for ILP Multiprocessors," *ACM SIGARCH Computer Architecture News,* vol. 25, no. 5, p. 1, 1997.

[26] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S.W. Keckler, "A NUCA Substrate for Flexible CMP Cache Sharing," *IEEE Trans. Parallel and Distributed Systems,* vol. 18, no. 8, pp. 1028-1040, Aug. 2007.

[27] J.P. Singh, W. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory," Stanford technical report, 1991.

[28] S.S. Mukherjee, M. Kontz, and S.K. Reinhardt, "Detailed Design and Evaluation of Redundant Multithreading Alternatives," *Proc. Int'l Symp. Computer Architecture (ISCA),* p. 0099, 2002.

[29] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream Processors: Improving Both Performance and Fault Tolerance," *ACM SIGOPS Operating Systems Rev.,* vol. 34, no. 5, pp. 257-268, Dec. 2000.

[30] J. Pool, I.S.K. Wong, and D. Lie, "Relaxed Determinism: Making Redundant Execution on Multiprocessors Practical," *Proc. USENIX Workshop Hot Topics in Operating Systems,* 2007.

[31] X. Fu, T. Li, and J. Fortes, "Combined Circuit and Microarchitecture Techniques for Effective Soft Error Robustness in SMT Processors," *Proc. Dependable Systems and Networks (DSN) Conf.,* pp. 137-146, 2008.

[32] M.A. Gomaa and T.N. Vijaykumar, "Opportunistic Transient-Fault Detection," *Proc. Int'l Symp. Computer Archtitecture,* pp. 172-183, 2005.

[33] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and D. Changhong, "Impact of CMOS Process Scaling and SOI on the Soft Error Rates of Logic Processes," *Proc. Symp. VLSI Technology 2001, Digest of Technical Papers,* pp. 73-74, 2001.

Ransford Hyman, Jr., received the Bachelor of Science degree in mathematics from Bethune-Cookman University in 2006. He is currently working toward the PhD degree in the Department of Computer Science and Engineering, University of South Florida. His current research interests include computer architecture, reliability in VLSI systems, and multicore designs. He is supported by the Florida Education Fund's Mcknight Doctoral Fellowship Program and the US National Science Foundation (NSF, S-STEM DUE # 0807023, Florida-Georgia LSAMP Bridge to Doctorate Supplement Award HRD # 0217675). He is a student member of the IEEE and the IEEE Computer Society.

Koustav Bhattacharya received the BTech degree in computer engineering from Kalyani University, West Bengal, India, in 2002, and the master's degree in computer technology from the Indian Institute of Technology, Delhi, India, in 2004. He received the PhD degree in computer science and engineering from the University of South Florida, Tampa, in 2009. In 2004, he worked as a design engineer at ST Microelectronics, Noida, India. He was awarded the Richard E. Merwin Scholarship in 2007. His research interests include design and optimization for reliable and secure VLSI systems, VLSI design automation, and computer architecture. He is a member of the IEEE.

Nagarajan Ranganathan (S'81-M'88-SM'92-F'02) received the BE (Hons.) degree in electrical and electronics engineering from the Regional Engineering College (National Institute of Technology), Tiruchirapalli, India, in 1983, and the PhD degree in computer science from the University of Central Florida, Orlando, in 1988. From 1998 to 1999, he was a professor of electrical and computer engineering at the University of Texas at El Paso. He is currently a distinguished university professor of computer science and engineering at the University of South Florida (USF), Tampa. His current research interests include VLSI circuit and system design, VLSI design automation, multimetric optimization in hardware and software systems, bioinformatics, computer architecture, and parallel computing. He has developed many special-purpose very large-scale integrated (VLSI) circuits and systems for computer vision, image and video processing, pattern recognition, data compression, and signal processing applications. He has authored or coauthored more than 250 papers in refereed journals and conferences and four book chapters and holds six US patents and two pending. He has edited three books titled: *VLSI Algorithms and Architectures: Fundamentals (IEEE CS Press, 1993), VLSI Algorithms and Architectures: Advanced Concepts* (IEEE CS Press, 1993), and *VLSI for Pattern Recognition and Artificial Intelligence* (World Scientific, 1995). He has coauthored a book titled: *Low-Power High-Level Synthesis for Nanoscale CMOS Circuits* (Springer-Verlag, June 2008). He was elected as a Fellow of the IEEE, in 2002, for his contributions to algorithms and architectures for VLSI systems. He is a member of the IEEE Computer Society, the IEEE Circuits and Systems Society, and the VLSI Society of India. He has served as an associate editor of several journals: the *IEEE Transactions on Circuits and Systems* (1997-1999), the *IEEE Transactions on Circuits and Systems for Video Technology* (1997-2000), the *IEEE Transactions on Computers* (2008-present), the *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2008-present), and the *Association for Computing Machinery (ACM) Transactions on Design Automation of Electronic Systems* (2007-present). He has also been on the editorial boards of *Pattern Recognition* journal (1993-1997) and the *International Journal of VLSI Design* (1994-2000). From 1997 to 2001, he served as the chair of the IEEE Computer Society Technical Committee on VLSI. He has been a member of the Steering Committee of the *IEEE Transactions on Very Large-Scale Integrated (VLSI) Systems* during 1999-2003 and 2008-present, the Steering Committee chair during 2002-2003, and the editor-in-chief for two consecutive terms from 2003 to 2007. He was the program cochair for the International Conference on VLSI Design (ICVLSID '94), the IEEE Computer Society Annual Symposium on VLSI (ISVLSI '96, ISVLSI '05, and ICVLSID '08), and the general cochair for the International Conference on VLSI Design (ICVLSID '95, IWVLSI '98, ICVLSID '98, ISVLSI '05, and ISVLSI '09). He has served as a member of the technical program committees of several international conferences including the International Conference on Computer Design (ICCD), the International Conference on Parallel Processing (ICPP), the International Parallel Processing Symposium (IPPS), the Symposium on Parallel and Distributed Processing (SPDP), the International Conference on High-Performance Computing (ICHPC), the International Symposium on High-Performance Computer Architecture (HPCA), the Great Lakes Symposium on VLSI (GLSVLSI), the International Symposium on Asynchronous Circuits and Systems (ASYNC), the International Symposium on Quality Electronic Design (ISQED), the International Symposium on Low Power Electronics and Design (ISLPED), the International Workshop on Computer Architectures for Machine Perception (CAMP), the International Symposium on Circuits and Systems (ISCAS), the International Conference on Microelectronic Systems Education (MSE), and the International Conference on Computer-Aided Design (ICCAD). He received the USF Outstanding Research Achievement Award in 2002, the USF President's Faculty Excellence Award in 2003, the USF Theodore Venette-Askounes Ashford Distinguished Scholar Award in 2003, the SIGMA XI Scientific Honor Society Tampa Bay Chapter Outstanding Faculty Researcher Award in 2004, the Distinguished University Professor Honorific Title and the University Gold Medallion Honor in 2007, and the USF Outstanding Undergraduate Teaching Award in 2009. He was a corecipient of the three Best Paper Awards at the International Conference on VLSI Design in 1995, 2004, and 2006, and the IEEE Circuits and Systems Society *VLSI Transactions* Best Paper Award in 2009.