A Light-weight Cache-based Fault Detection and Checkpointing Scheme for MPSoCs Enabling Relaxed Execution Synchronization

Chengmo Yang and Alex Orailoglu
Computer Science and Engineering Department
University of California, San Diego
9500 Gilman Drive, La Jolla, CA 92093
{c5yang, alex}@cs.ucsd.edu

ABSTRACT

While technology advances have made MPSoCs a standard architecture for embedded systems, their applicability is increasingly being challenged by dramatic increases in the amount of device failures that may occur during execution. Conventional fault tolerance techniques employ a ***duplication-and-comparison*** *strategy to detect arbitrary execution faults, as well as a* ***checkpointing-and-rollback*** *strategy to recover from the faulty state. Comparison and checkpointing are performed either at task level, thus imposing a large amount of overhead in verifying and backing up memory pages, or at instruction level, thus necessitating a lock-step execution model which significantly limits the attainable performance. To overcome the shortcomings of both strategies, in this paper we propose a cache-based fault tolerance scheme wherein the comparison and checkpointing process is performed at the cache-memory interface. By allowing two processors that execute duplicated tasks to share a single data cache, the proposed scheme is able to verify execution results before writing them back into memory, thus protecting the memory from being polluted by execution faults. This in turn significantly reduces the checkpointing overhead. Meanwhile, since only the data written into memory are compared, the strict instruction-by-instruction synchronization model used in multithreading processors can be relaxed. The simulation results confirm that the proposed scheme only imposes a performance overhead ranging from 1.4% to 10.4%, while both fault detection and execution checkpointing can be effectively attained.*

Categories and Subject Descriptors: C.4 [Performance of Systems]: –*Fault tolerance*

General Terms: Reliability, Performance, Design

Keywords: Fault detection, Fault recovery, Checkpointing

1. INTRODUCTION

The Multiprocessor System-on-Chip (MPSoC) [1] is rapidly becoming a standard organization for embedded systems to help utilize the ever increasing amount of extant computational power. As technology scaling advances towards nanoscale, however, MPSoCs will not only hit the wall of parallelism, but also suffer from various types of device failures that may occur during execution [2]. The shrinking feature size, the higher frequency and the lower threshold voltages have accentuated a number of electronic effects (e.g., electromigration) that may lead to permanent device failures. Meanwhile, transient failures, which can be caused by alpha-particle strikes, cosmic rays, or radiation from radioactive atoms [3], have also increased by orders of magnitude due to these aggressive technology trends and tighter noise margins.

The increasingly pessimistic news on the reliability front requires the consideration of *fault tolerance* as a primary design constraint for embedded MPSoCs so as to guarantee functional and timing correctness even in the presence of hardware and software failures. Generally speaking, the achievement of fault tolerance necessitates *redundancy* at an amount inversely proportional to the regularity of the hardware components. Storage structures, such as caches and memory, have regular patterns, thus enabling the use of *protection codes* such as parity bits and Error Correcting Codes (ECC). Faults in instructions or in control flow can also be effectively detected by *signature monitoring* techniques [4] through exploiting internal redundancies. As a comparison, combinational logic structures typically exhibit irregular patterns, thus requiring the entire execution to be *replicated* in order to detect arbitrary transient and permanent faults.

One fundamental obstacle to the adoption of fault resilient systems has been the high cost and inflexibility associated with standard techniques such as *triple modular redundancy* (TMR), wherein each computation needs to be triplicated so as to correct a single error. A more efficient set of solutions typically employs a *duplication-and-comparison* strategy to detect faults, as well as a *checkpointing-and-rollback* strategy to restore the computation to a previously saved clean state (a checkpoint). This class of techniques can be classified according to the granularity of the comparison and checkpointing scheme. *Task-level* fault tolerance techniques duplicate each task on distinct processors and provide distinct memory regions to each task, necessitating the comparison and checkpointing of all the modified memory pages of the two tasks. This comparison and checkpointing process usually requires the operating system to be involved, thus imposing significant overhead. In contrast, *instruction-level* fault tolerance techniques duplicate each instruction and compare the results before writing them into the cache, thus preventing memory from being polluted by execution faults. The two identical threads can be executed either on two tightly-coupled cores within an MPSoC [6], or on a Simultaneous and Redundantly Threaded (SRT) processor [5]. However, both schemes necessitate a *lock-step* execution model such that no instruction in the leading thread can be committed until the trailing thread verifies its correctness, thus significantly increasing the execution latency of a single instruction. These techniques also

CASES'08, October 19–24, 2008, Atlanta, Georgia, USA.

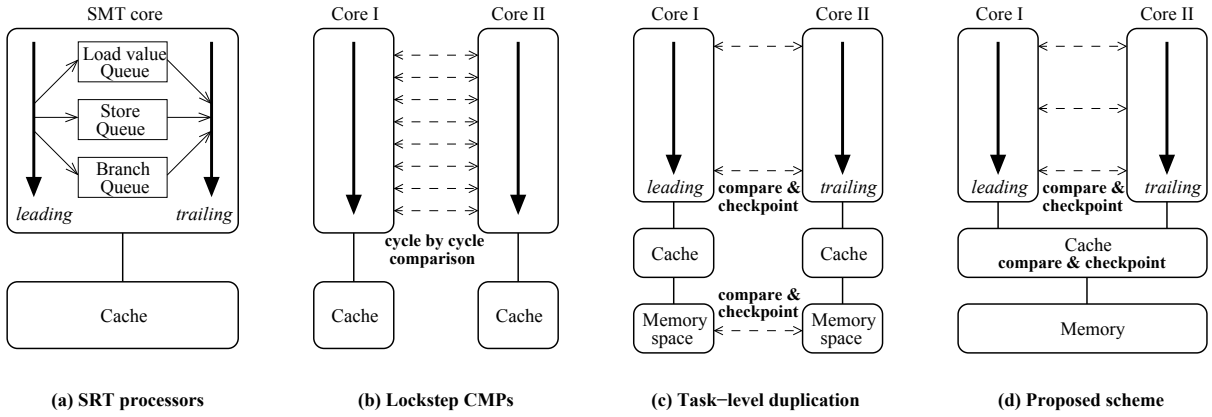

Figure 1: Differences between SRT, lockstep CMP, task-level duplication and proposed scheme

require the use of dedicated hardware queues [5, 6] to forward the load values, the branch results and the execution results of the leading thread to the trailing thread. The significant hardware overhead and the performance degradation preclude these techniques from being widely used by embedded MPSoCs, which typically exhibit stricter hardware and timing constraints than general purpose multiprocessors.

To design a light-weight fault detection and checkpointing scheme for embedded MPSoCs, in this paper we propose a *cache-based* fault tolerance framework wherein the comparison and checkpointing process is performed at the cache-memory interface. Rather than employing a dedicated hardware queue to buffer execution results of the leading thread, we propose to share a single cache between the two identical threads, thus allowing the trailing thread to verify the store values produced by the leading thread directly in the cache. Meanwhile, as data stored in the cache are compared before being written into memory, the memory is strictly protected from being polluted by execution faults. This implies that a checkpoint only needs to be established whenever a dirty cache line is written back into memory, and only the **processor state** needs to be checkpointed. As a result, the leading and the trailing threads do not need to compare each pair of execution results. Instead, they only need to synchronize at each checkpoint, thus significantly reducing the performance overhead imposed by the lock-step execution model.

The remainder of this paper is organized as follows. Section 2 discusses the limitations of current fault tolerance techniques. Section 3 motivates the proposed scheme. Sections 4 and 5 present the detailed design and implementation of the proposed light-weight fault detection and checkpointing mechanism. Section 6 experimentally verifies the efficacy of the technique, while section 7 summarizes the paper.

2. TECHNICAL BACKGROUND

The elevation in fault rates has caused increasing research attention to be paid to the incorporation of fault tolerance into computation systems. However, as the occurrence of faults is still relatively sparse compared to the fault-free cases, fault tolerant techniques should be evaluated based not only on their *effectiveness* in detecting and correcting errors, but more importantly on their *efficiency* so that the associated performance and hardware overhead can be minimized. Given the severe power constraints of modern SoC devices, the need for maximally efficient fault tolerance methods becomes even more critical and urgent.

Essentially, the attainment of fault tolerance necessitates solutions to two fundamental issues: *fault detection* and *execution recovery*. *Redundant execution* has been established as an effective approach for detecting arbitrary transient and permanent faults in combinational logic structures. Traditionally these techniques have been classified into two categories: *time redundancy* and *space redundancy*. Time redundancy is achieved by executing a task on the *same* hardware multiple times, such as in the Simultaneously and Redundantly Threaded (SRT) processors [5] shown in Figure 1a. To detect faults, SRTs require not only an *output comparator* to verify execution results, but also an *input replicator* so as to enable the two threads to independently read input data. Typically the input replicator and the output comparator are implemented through two centralized shared structures [5], namely, a Load Value Queue and a Store Queue. The leading thread needs to be stalled if either queue is full, while the trailing thread needs to be stalled if either queue is empty. Obviously, time redundancy techniques, such as SRT, can only be used to tolerate transient faults. Space redundancy, on the other hand, is achieved by executing a task on multiple disjoint processors so as to tolerate both transient and permanent faults. The duplicated tasks may be executed at different timing steps (Figure 1c), implying that distinct memory regions need to be provided for each task, and each modified memory page needs to be compared for fault detection. As an alternative, the duplicated tasks can also be executed on two tightly-coupled cores on a cycle-by-cycle basis, such as the *Compaq NonStop Himalaya* system [9] (Figure 1b). This highly synchronized lock-step execution model enables the results of each pair of instructions to be easily compared, yet imposes significant performance overhead. Meanwhile, as the two copies of the task do not communicate during execution, both copies will incur mis-speculated branches and cache misses independently, leading to less efficient resource utilization and unnecessary performance degradation.

In addition to fault detection, the achievement of fault resilience also necessitates *recovery* techniques, which should either preclude faults from modifying computation states, or roll the execution back to a previously saved clean checkpoint upon fault detection. The first strategy is usually employed in SRTs and lock-step multiprocessors [7, 8], wherein an instruction in the leading thread cannot write its result to either a register or the cache until the trailing thread verifies its correctness. Obviously, this strategy significantly increases the execution latency of a single instruction, thus not only requiring extra hardware to save instruction results, but furthermore delaying the release of hardware resources, such as physical registers and ROB entries. In contrast, the checkpointing and rollback strategy allows results to be written into registers and memory without being compared, yet needs to constantly save the computation state, including both register space and memory space. One set of techniques estab-

lishes checkpoints at a higher level using the virtual memory translation hardware [10], wherein a backup copy needs to be created before modifying any memory page. Another standard technique is to use a recovery cache to record all the data written in memory that are part of a checkpoint state [11]. Every store to a memory location must be preceded by a load to maintain the data in the recovery cache. These backup techniques impose significant hardware and performance overhead constantly on the system, since traditionally checkpoints are established per a fixed number of instructions or execution cycles. Some researchers have tried to adaptively adjust checkpointing frequency according to fault rates [12]. However, since the checkpointing overhead is not reduced, these adaptive techniques are only effective when the fault rate is low. If the fault rate is high, these checkpointing approaches will be either impractical as an application would spend most of its time taking checkpoints, or infeasible as there would be insufficient time for an application to save its execution state before the next failure occurs.

3. TECHNICAL MOTIVATION

The detailed examination presented in the last section indicates that the fundamental shortcoming of the task-level fault tolerance techniques (Figure 1c) is that data is allowed to be written into memory before being verified, thus creating extra complexity in detecting faults and establishing checkpoints. On the other hand, the crucial limitation of the cycle-by-cycle comparison techniques (Figure 1a&b) is the significant performance overhead imposed by the highly synchronized execution model, as well as the hardware overhead in holding the results of the leading thread. To overcome these limitations, a light-weight fault detection and checkpointing scheme should allow execution results to be written into registers and caches before being compared, yet protect memory from being polluted by execution faults. This has motivated us to propose a *cache-based* fault tolerance mechanism wherein the comparison and checkpointing process is performed at the cache-memory interface. As shown in Figure 1d, both threads are allowed to write results into registers and a shared data cache, yet only the store values in the cache are compared for fault detection. This fault detection scheme completely eliminates the necessity of dedicated hardware queues to capture load and store values. Meanwhile, as the data in cache are compared before being written into the memory, a checkpoint only needs to be established when a dirty cache line needs to be replaced, necessitating only the *processor state*, that is, the program counter (PC) and the register values[1], to be checkpointed. This highly reduced checkpointing overhead in turn enables checkpoints to be established more frequently for systems with high fault rates.

Utilizing the cache-memory interface to reduce checkpointing overhead has been proposed in [13] for single processors, and in [14] for shared-memory multiprocessors with snooping protocols. However, previous research has not specifically addressed the issue of sharing a single data cache among two duplicated threads so as to enable fault detection to be integrated together with rollback recovery. On the other hand, the sharing of a single cache does create a critical issue of *cache block dependences*, which may force the leading thread to constantly wait for the trailing thread, thus creating unnecessary performance degradation. The impact of this issue can be illustrated more clearly through the loop example presented in Figure 2.

[1]Some branch handling and exception taking techniques may necessitate a few special purpose registers to be additionally saved.

```
for (i=1; i<MAX; i++) {
   A[i] = A[i+1] + 1;
   A[i-1] = 2*i;
}
```

Figure 2: Loop with cache block dependences

In this loop, each array element $A[i]$ is first read at the $(i-1)^{th}$ iteration, and then written at the i^{th} and $(i+1)^{th}$ iterations. As the two threads are not executed in a lock-step manner, the **L** (leading) thread may run one iteration ahead of the **T** (trailing) thread, leading to two possible types of cache block dependences. The L thread may try to write $A[i]$ before the T thread reads the old value (a *WAR dependence*), thus causing the T thread to obtain an incorrect value, or may try to overwrite $A[i-1]$ before the T thread checks the old value (a *WAW dependence*), thus causing the T thread to incorrectly report the detection of an error.

Although both types of cache block dependences can be preserved by forcing the L thread to wait for the T thread, this extra synchronization requirement may reduce the performance benefit that could be obtained by the relaxed non-lockstep execution model. Another possible yet highly inefficient solution would be to duplicate each load value and each store value in dedicated hardware queues for the T thread to access, as has been performed by the SRT processors. Instead of duplicating all the load and store values, we propose to only duplicate a cache block upon the detection of a WAR or WAW cache block dependence. The cache design is extended so as to incorporate a *split* capability: the L thread is allowed to write to the block in the regular cache, while the old value is placed in a *victim cache* for the T thread to read or to verify. The cache access controller is furthermore extended to detect both types of cache block dependences, split a cache block into two versions, and merge the two versions together if later the T thread catches up to the L thread in the execution progress.

4. CACHE-BASED FAULT TOLERANCE

4.1 Architectural overview

Essentially, the proposed fault detection and recovery framework duplicates an application into two threads on an MPSoC platform to achieve fault resilience. Unlike traditional SRT processors, these two threads will be simultaneously executed on **different** cores to achieve a more balanced workload and to detect permanent faults in addition to transient faults. This architecture model is more suitable for embedded systems since the processing elements (PEs) on the MPSoC platform are not necessarily multithreaded cores. In contrast, the PEs can either be in-order single-issue cores, or VLIW cores, or superscalar cores, or even heterogeneous cores with identical instruction sets. The only architectural constraint imposed by the proposed fault tolerance scheme is that each core should perform memory accesses *non-speculatively* and *in-order*, so that the duplicated tasks would generate identical memory access patterns during execution. For most embedded MPSoCs, this requirement can be naturally fulfilled.

Since the two threads generate identical memory access patterns in the fault free case, a single L1 data cache can be shared between the two threads to achieve more efficient resource utilization. Meanwhile, this sharing enables the design of an efficient *fault detection* scheme. By ensuring that one thread, denoted as the **leading (L)** thread, always runs ahead of the other, denoted as the **trailing (T)** thread, the L thread can write results into the cache, while the T thread

can directly verify the correctness of these values. Only when the two threads agree can a data be written to the lower level storage in the memory hierarchy (either the main memory or the L2 cache if the system has multiple levels of caches). Obviously, this cache-based fault detection mechanism eliminates the use of dedicated hardware queues for holding the results of the L thread.

While the proposed execution model requires the L thread to always run ahead of the T thread, this run-ahead requirement is not imposed on a cycle-by-cycle basis, but only for memory access instructions. In other words, the two threads can execute non-memory access instructions *independently*; yet before executing a load/store, the T thread needs to ensure that the L thread has already executed that instruction. To accomplish this run-ahead property, the proposed scheme uses an *access counter* to globally track the difference in the memory access counts of the two threads: the counter value is incremented whenever the L thread executes a load/store, and decremented whenever the T thread executes a load/store. The execution of the T thread is stalled if the value of the counter is 0, thus fulfilling the run-ahead requirement.

The run-ahead requirement of the L thread provides an extra benefit in balancing the workload. On one hand, the L thread runs ahead in execution, brings data into cache upon misses, and initiates a checkpointing request if a dirty cache needs to replaced. On the other hand, the T thread manages fault detection: each store initiated by the T thread will be changed into a read of the corresponding cache block and a comparison of the two values. Once the T thread also reaches the computation point at which the L thread initiates the checkpointing request, the two processor states will be compared to verify the correctness of the checkpoint.

In line with other redundant execution-based fault tolerance schemes, the proposed technique only targets the faults occurring in the execution pipeline. Faults in instructions or in control flow are assumed to be detected using *signature monitoring* techniques [4]. Storage structures such as caches, register files, and the main memory are assumed to be protected using ECC, while buses are assumed to be protected using parity. This safe storage is therefore utilized to store the checkpoints, so that the execution can be recovered to a clean state upon the detection of any computation fault.

4.2 Fault detection

The design of a light-weight checkpointing scheme necessitates the preclusion of execution faults from polluting memory states. In general, an execution fault would affect memory states if it propagates through the dependence chain to a store instruction, thus causing either the store *value* or the store *address* to be incorrect. Accordingly, the proposed cache-based fault detection scheme not only compares each pair of store values, but also verifies memory access patterns. Meanwhile, the two threads will also record their register values *individually* at each checkpoint, thus enabling a comparison of the two processor states to detect execution faults. The combination of the *store* verification and the *register* verification ensures that any execution fault, if it has not been masked during computation, will be detected eventually.

The proposed fault detection process is managed by the cache controller. During execution, the L thread is allowed to directly write its results into cache, while each write initiated by the T thread will be changed into a read of the corresponding cache block and a comparison of the two values. Traditionally a valid cache block can either be '*clean*' or '*dirty*', depending on whether the value of the block has been updated or not. To support fault detection, however, an extra '*verified*' state needs to be maintained so as to differentiate whether or not the data in cache has been verified by the T thread. A store initiated by the L thread would therefore make a cache block *dirty*, while the same store later initiated by the T thread would make the cache block *verified*, if the two store values match.

Using this extra state, the proposed cache access strategy can directly detect any mismatch in a pair of store values, as well as faults in store addresses. In general, if an execution fault causes a store address to change from A_x to A_y, both the A_x and the A_y block would exhibit a mismatch in the number of store instructions. This mismatch typically would result in the value of the extra store being compared to the value of the preceding or the subsequent store, thus causing a value mismatch to be reported. If the extra store fails to be compared, the cache controller can still detect an inconsistent access pattern by monitoring the state of the cache block. In fact, the cache controller will generate a "fault-detected" signal for any of the following **three** types of faults:

Disagreeing register values: The T thread reaches the checkpoint, while the current processor state of the T thread does not match the processor state recorded by the L thread at the same checkpoint.

Disagreeing store values: The T thread is about to write a *dirty* cache block, while the value to be written does not match the value in the block produced by the L thread.

Inconsistent store sequences: As the L thread always runs ahead of the T thread, two cases will indicate the existence of a mismatch in the store sequences: 1) the T thread is about to write a cache block, while the data is not in the cache or the block is not in the '*dirty*' or the '*split*' state, indicating that the L thread has not written to that block yet. 2) The T thread reaches the checkpoint, while at that time there exists a '*dirty*' block in the cache, indicating that the T thread has not verified the data written by the L thread.

These three cases clearly confirm that the proposed technique can detect un-masked execution faults that propagate to either store values, or store addresses, or register values at a checkpoint. Masked faults, on the other hand, would not affect the correctness of the computation. Therefore, the proposed light-weight fault detection technique can effectively preclude execution faults from polluting either the main memory or the established checkpoints, which in turn enables the design of a light-weight checkpointing and rollback scheme.

The design of a semantically correct fault detection scheme also necessitates the preclusion of a third processor from polluting memory states. More specifically, in a multi-core environment, a third processor may try to modify a block in the proposed cache design using the cache snooping protocol. However, if the block is pending to be verified by the T thread, this modification would cause the verification to fail. Similarly, if the block is pending to be read by the T thread, this modification would cause the T thread to load a data inconsistent with the data already loaded by the L thread. To preclude these potential violations of fault detection semantics, in the proposed fault tolerance scheme, a third processor is only allowed to modify a block in the shared cache if the block is not at the *dirty* stage, and the block exhibits a balanced number of read accesses from the two threads (which can be detected using the technique presented in Section 4.5).

4.3 Checkpoint establishment

A *checkpoint* records complete information about the computation state so that it may be utilized later to recover from an error by restarting the computation from that point. Typically a checkpoint consists of the processor state as well as the corresponding memory footprint. To reduce checkpointing overhead, however, the proposed fault tolerance scheme does not maintain extra copies for values written into memory. In

order to make the processor state and the memory footprint *consistent*, whenever a data needs to be written into memory, the corresponding processor state needs to be checkpointed. In write-through caches, each store value needs to be written into memory, thus requiring the register values to be recorded on every store instruction so as to generate consistent checkpoints. This checkpointing frequency can become intolerably high. In contrast, the proposed fault tolerance scheme employs a write-back cache, implying that a checkpoint only needs to be established upon the write-back of a block, that is, whenever a dirty cache line needs to be replaced. For set associative caches, the checkpointing frequency can be furthermore reduced by modifying the replacement policy so that clean lines are selected over dirty cache lines for replacement. Meanwhile, the size of the shared cache can be enlarged to be twice the size of the original private caches of each core, thus reducing the write back frequency and in turn the checkpointing frequency furthermore.

As the L thread always runs ahead of the T thread and as a cache block will not be replaced if it is pending to be read by the T thread, only the L thread will encounter cache misses during execution. As a result, checkpointing requests will always be initiated by the L thread, if a cache miss requires a dirty cache line to be replaced. However, the processor state to be checkpointed is not the state when the last write to the dirty cache line took place, since that state has already been overwritten by subsequent execution. Instead, only the *current* processor state of the L thread, that is, the computation point at which the checkpointing request is initiated, can be recorded. Therefore, in order to establish a consistent checkpoint, not only the dirty cache line selected for replacement, but also all the other dirty cache lines that have been updated since the last checkpoint need to be written into memory.

The proposed checkpointing strategy can be illustrated more clearly by considering the loop example presented in Figure 2. To simplify the analysis, we assume that the cache is a direct mapped cache with 8 lines, while each line is used to hold a single array element. As all the cache blocks are invalid at the beginning of the loop execution, the execution of the first six iterations ($i = 1$ to 6) causes the 8 cache lines to be filled with $A[0]$, ..., $A[7]$, respectively. At the 7^{th} iteration, $A[8]$ needs to be brought into the cache, while the corresponding cache line is occupied by $A[0]$. Since the value of $A[0]$ has been modified, this block needs to be written back into memory, thus requiring a checkpoint to be established before the L thread loads $A[8]$ into the cache. The processor state to be recorded is the computation point when the L thread is about to replace $A[0]$ with $A[8]$ (at the 7^{th} iteration), rather than the PC and register values corresponding to the last update of $A[0]$ (at the 1^{st} iteration). Therefore, in order to maintain a consistent memory footprint corresponding to the current processor state, not only $A[0]$, but also the other dirty cache blocks, $A[1]$, ..., $A[6]$, should be written into memory.

When the L thread reaches a checkpoint, the T thread is still in the process of verifying store values. The execution of the T thread needs to be monitored so as to determine whether the T thread has also reached that checkpoint. Only monitoring the PC value would not suffice, since the checkpoint may happen to be an instruction in a loop. Instead, the proposed scheme only starts to compare the PC of the T thread to the PC of that checkpoint if the two threads have performed the same number of memory accesses, that is, if the value of the global *access counter* equals 0. In the fault-free case, a match of the two PC values, which indicates the arrival of the T thread at that checkpoint, can always be detected. If, on the other hand, no match between the two PC values has been reported before the T thread issues a subsequent load/store, a mismatch in the memory access pattern has been detected, thus resulting in the reporting of a fault.

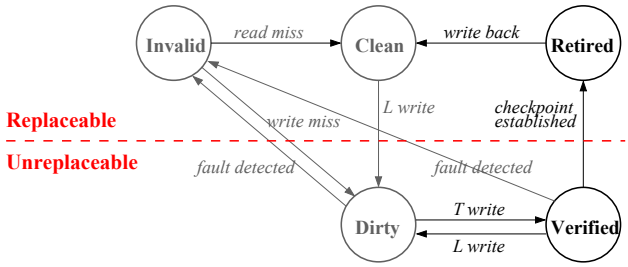

Figure 3: Cache state diagram to implement fault detection, checkpointing, and rollback

Once the T thread also arrives at a checkpoint with no intervening error detection, a new checkpoint is established by saving the processor state into reliable storage, either a dedicated hardware buffer or a fixed location in main memory, and copying all *verified* cache lines into memory. However, copying all verified cache lines into memory would generate a burst of memory requests that may significantly degrade system performance. To overcome this issue, we employ the idea of making these cache lines *unchangeable*, which is originally proposed in [13] for single processors. More specifically, we maintain an extra "*retired*" state for each cache line, and all the "*verified*" cache lines will be marked as "*retired*" once the T thread reaches the checkpoint with **no** error being detected. The use of this extra state enables a distribution of the write back requests for the *retired* blocks: during subsequent execution, a *retired* block can be written back into memory upon a replacement of that block, or upon the first subsequent *write-hit* on that block.

The complete cache state diagram in supporting fault detection, checkpointing and rollback is presented in Figure 3. As can be seen, each cache block can be in any of five possible states: the three traditional states of *invalid*, *clean* and *dirty*, as well as the two extra, *verified* and *retired*, states. The transitions among these five states accomplish **four** fundamental functions of the proposed fault tolerance scheme:

Fault detection: A store performed by the L thread will make the corresponding cache block *dirty*, while the same store performed by the T thread will make the cache block *verified* if the two store values match. A fault will be reported if the two store values differ.

Checkpoint initiation: The *invalid*, the *clean*, and the *retired* states are marked as **replaceable**, implying that, a cache block in these states can be replaced with no need of establishing a new checkpoint. In contrast, if a *dirty* or a *verified* block is selected for replacement upon a cache miss, a new checkpointing request will be initiated by the L thread.

Checkpoint establishment: Once the T thread reaches a checkpoint and no fault has been reported, all the *verified* cache blocks will be marked as "*retired*". During subsequent execution, a replacement or the first subsequent write-hit of a *retired* cache block requires the data to be first written into memory, which will therefore make that block *clean*.

Execution rollback: Upon the detection of any fault, the register values saved at the old checkpoint will be reloaded to the register file of each thread, and all *dirty* and *verified* cache blocks will be marked as *invalid*. The PC value saved at the old checkpoint will then be reloaded to each thread so as to resume execution from the old checkpoint.

4.4 Thread synchronization at checkpoints

The examination in the last subsection clearly shows that in the proposed fault tolerance scheme, a checkpointing re-

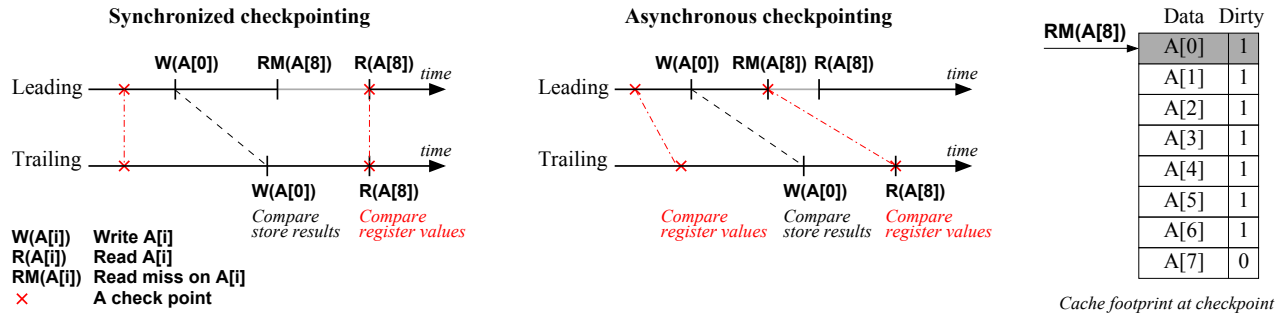

Figure 4: Synchronized v.s. asynchronous checkpointing for the loop example

quest is initiated by the L thread whenever a *dirty* or *verified* cache line needs to be replaced, while the new checkpoint is established by the T thread once it also reaches the same computation point. The two threads can synchronize at each checkpoint using two distinct approaches. On one hand, each new checkpoint can be established *synchronously*: after initiating a checkpointing request, the L thread needs to await the completion of the fault detection process of the T thread and the isochronism of the two threads. Once the new checkpoint has been established, both threads will simultaneously be allowed to proceed, the block to be replaced (which is at the *retired* state at that time) will be written into memory, and the cache miss will be served. This *synchronized* checkpointing strategy only needs to maintain a single checkpoint at any time, yet unnecessarily forces the L thread to await the T thread to fully catch up. On the other hand, if the T thread has verified the correctness of the data to be replaced and no more read accesses to that data are needed by the T thread[2], the cache block selected by the L thread is able to be replaced. The data to be replaced, however, needs to be stored in a dedicated register rather than being written back into memory immediately, since the new checkpoint has not been established yet. Meanwhile, since the T thread has not verified the correctness of the processor state of the L thread, the old checkpoint cannot be overwritten, and this state needs to be recorded in extra storage. The value of the global *access counter* also needs to be duplicated, thus preventing subsequent memory accesses performed by the L thread from interfering with the detection of the T thread reaching the checkpoint.

The differences between *synchronized* vs *asynchronous* checkpointing schemes can be observed more clearly in Figure 4, which applies both strategies on the loop example shown in Figure 2. A checkpointing request is initiated when the L thread encounters the read miss on $A[8]$ at the 7^{th} iteration. In the synchronized scheme, the L thread waits until the T thread also reaches this computation point. At that time, if no fault has been detected, a new checkpoint will be established, $A[0]$ will be written into memory, and then $A[8]$ will be brought into the cache. In contrast, in the asynchronous checkpointing scheme, the L thread only needs to wait until the T thread verifies the correctness of $A[0]$, that is, upon the completion of the store instruction at the 1^{st} iteration. Then, the value of $A[0]$ and the processor state of the L thread will be stored in safe storage, and the L thread will proceed to bring $A[8]$ into the cache.

Compared to the synchronized checkpointing scheme, the asynchronous scheme necessitates extra storage to record the data to be replaced, the global access counter, and the processor state of the L thread, yet reduces the waiting time of the L thread by allowing it to proceed if the data selected for replacement ($A[0]$) is at the *verified* state and no more read accesses to the data are needed by the T thread. However, in this process the L thread should not write to the cache if the new checkpoint is still pending to be established. Allowing the L thread to write to the cache would either overwrite a *dirty* cache block, causing the verification of the old value to fail, or change a *clean/verified* block to *dirty*. When the T thread arrives at the new checkpoint, this extra *dirty* block will cause a fault to be reported. These two potential violations of the fault detection semantics constrain the L thread to proceed only in a quite restricted manner without performing any cache write operation, if the new checkpoint is still pending to be established.

4.5 Splitting and merging a cache block

As has been illustrated using the loop example, the non-lockstep execution model we employ creates a critical issue of *WAR* and *WAW cache block dependences*: the L thread initiates a write request to a cache block, while the old value is still pending to be read or be verified by the T thread. Rather than forcing the L thread to constantly wait for the T thread or duplicating each load value, the design goal of the proposed fault tolerance scheme is to execute the two threads *independently* without reliance on unnecessary synchronization requirements, yet to selectively duplicate a load value upon the detection of a cache block dependence. To achieve this goal, we propose to incorporate a *split* capability into the cache design: whenever a *cache block dependence* is detected, the L thread is allowed to update the regular cache block, while the old value is placed in a victim cache for the T thread to read or to verify.

The fundamental issue encountered in supporting the incorporation of the *split* capability is the detection of a pending read or write of the T thread. The latter case can be easily detected through checking the cache block state, since a *dirty* state indicates the existence of a pending write of the T thread. The detection of a pending read would also be trivial, had the compiler marked the last load associated with each store using profiling techniques. Unfortunately, such compiler techniques would fail if the last read is conditionally performed according to branch outcomes. Nonetheless, as the two threads exhibit an identical memory access pattern, a purely dynamic technique can be developed to detect a pending read through maintaining a *read counter* for each cache block. The counter value, initialized to **zero** upon a store by the L thread, is incremented upon a load by the L thread and decremented upon a load by the T thread. In this way, a store request initiated by the L thread is only allowed to proceed directly if the counter of the corresponding cache block is zero. If the counter is non-zero, however, a pending read of the T thread is detected.

Upon the detection of a pending read/write by the T thread, the cache block can be split into two if there exists a free

[2]Further information on the detection of pending read accesses is to be found in the next subsection.

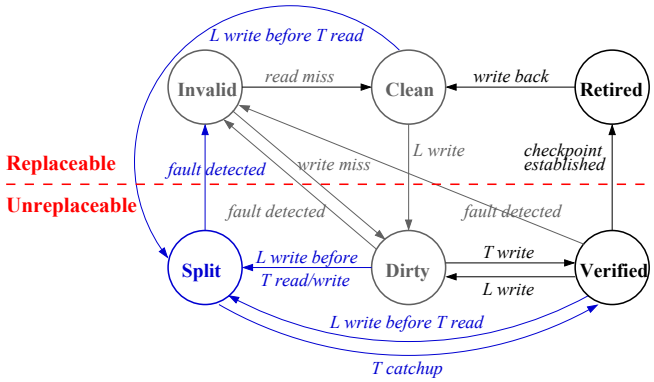

Figure 5: Adding a *split* state into cache design

entry in the victim cache: the old value will be saved in the victim cache for the T thread to access, while the L thread can proceed to overwrite the regular cache block. To accomplish this function, an extra *split* state is added to the cache state diagram, as shown in Figure 5. A write initiated by the L thread would cause a cache block to enter the *split* state if 1) the block state is *dirty*, or 2) the block state is *clean* or *verified* and the read count is nonzero. This state diagram exhibits no transition from the *retired* state to the *split* state, since a write-hit to a *retired* cache block requires the data to be first written into memory, thus making the block *clean*.

Once a cache block has been split into two, the execution of each thread becomes *independent* in that each thread has its own copy of data to write, and each thread would read the data written by itself. This independence also implies that a store value cannot be verified if, before the T thread initiates that store, the corresponding value in the regular cache block has already been overwritten by the L thread. This overwrite, however, would not cause any loss in fault coverage, since the old store value would never be written back into memory.

Figure 5 also shows that a victim cache block can be merged with a regular cache block if the T thread catches up to the L thread in execution, that is, if the two copies of data are produced by the same store of the two threads **and** the two values match. This capability can be attained in the proposed cache design through the addition of a *version counter* to each block in the victim cache so as to determine whether the data in the regular cache and in the victim cache are produced by the same store. The counter value, initialized to 1 upon the splitting of a cache block, is incremented upon a store by the L thread and decremented upon a store by the T thread. If a store performed by the T thread changes the version counter value to 0, the T thread will additionally trigger a comparison of the two split data copies for fault detection. If the two values match, the entry in the victim cache will be deallocated so that it can be reused to serve a subsequent split request. Meanwhile, the corresponding regular cache block will be set to the "*verified*" state, as shown in Figure 5.

It needs to be noted that in the aforementioned mechanism two split blocks are only checked for a merging possibility upon a *store* initiated by the T thread, since reading a victim cache block would not alter the value of the version counter. More crucially, only performing merge checking upon a store eliminates the need of any *read counter* for a victim cache block. Once two split blocks are merged together, the read counter value in the regular cache should be set to the difference between the read counts of the two threads. For a *split* cache block, however, the read counter in the regular cache only records the read count of the L thread. Nonetheless, upon the completion of a store of the T thread, the read count of the corresponding victim cache block is always 0.

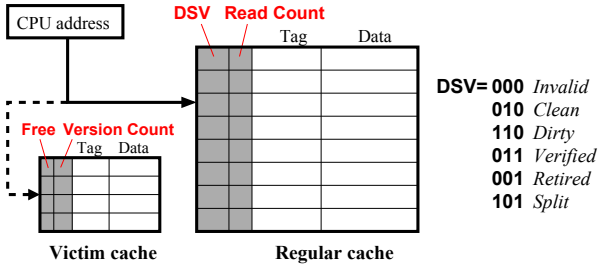

Figure 6: Hardware extension to traditional cache

Therefore, by performing merge checking only upon a store of the T thread, the read count of the merged block would remain constant, if the checking indicates that two data copies can be merged.

So far, we have discussed in detail the approach to split a cache block upon the detection of a *WAR* or *WAW cache block dependence*. Another situation that also necessitates the split of a cache block is when the L thread wants to replace a *clean* or *retired* block upon a cache miss, yet the old value is still pending to be read by the T thread. Replacing a *clean* or *retired* block does not create a checkpoint request. However, if the L thread proceeds to replace the data, a pending read by the T thread would encounter a cache miss, which is considered to be a faulty case in the proposed scheme. To preclude this potential violation of fault detection semantics, whenever the L thread selects a *clean* or *retired* cache block for replacement, the old value needs to be written into the victim cache if the read counter of that block is nonzero. The corresponding read count also needs to be written into the *version counter* field of the victim cache so that the T thread can decrement the read count upon subsequent read accesses, and the block can be freed once the counter becomes 0.

5. IMPLEMENTATION

5.1 Cache access control

Figure 6 presents the hardware extension to traditional cache design in supporting the proposed fault detection and checkpointing scheme. As can be seen, a fully associative small victim cache is added into the design to achieve the split capability upon the detection of a cache block dependence. A small counter is added to each regular cache block to record the read count, and to each victim cache block to record the version count. Meanwhile, the *valid* and the *dirty* bits used in a traditional cache are replaced by a ***Dirty-Shared-Verified (DSV)*** vector, which is used to encode the 6 possible states of each cache block. The encoding presented in Figure 6 is assigned so that a 1 on the ***D*** bit denotes a *dirty* or *split* state for the cache block. As a result, when the T thread reaches a checkpoint, the cache controller can globally check if any block is in the *dirty* or the *split* state, thus simplifying the fault detection process. This encoding scheme furthermore enables the use of the following logic expressions to check whether a block is **replaceable** and whether a block has been **split**:

$$\text{Replaceable} = \overline{D + SV} \quad (1)$$

$$\text{Split} = D \cdot V \quad (2)$$

In the proposed framework, the L thread is allowed to perform a read/write if the global *access counter* has not reached its upper bound, while the T thread is allowed to perform a read/write if the counter value is nonzero. More crucially, based on the values of the DSV vector and the counter, all the unblocked read/write accesses to each cache block can be controlled as follows:

Read hit by the L thread: A regular cache read is performed. The block state remains constant, while the read counter is incremented by 1 unless the load is speculative.
Write hit by the L thread: If a checkpoint is pending to be established, this write is blocked; otherwise, a regular cache write is performed. However, the old value needs to be first written into the victim cache if the block has not been split yet the *read counter* is nonzero, or written into memory if the block is at the *retired* state. The execution of this write will cause the cache block to be at either the *dirty* state if the current value of *read counter* is 0, or the *split* state if the value is nonzero. In the latter case, the *version counter* of the corresponding victim cache block is also incremented by 1. Once the block state has been updated, the read counter is reset to 0 regardless of the final block state.
Miss by the L thread: If a non-replaceable block is selected for replacement, a checkpointing request is initiated; otherwise, a regular cache replacement is performed, while the old value needs to be first written into the victim cache if the read counter of that block is nonzero.
Read hit by the T thread: If the block has been split, the read operation is redirected to the victim cache; otherwise, a regular cache read is performed, and the read counter of that block is decremented by 1.
Read miss by the T thread: The victim cache is accessed. If the access misses, a fault is reported; otherwise, the *version counter*, used to record the read count in this case, is decremented by 1, and the victim cache entry is subsequently released if the counter value changes to 0.
Write by the T thread: If the data is not in the cache, a fault is reported; otherwise, if the block has not been split, this write is changed to a read of the corresponding cache block. A fault is reported if the block is at the *dirty* state, or if the two values do not match. In the fault-free case, the block state is changed to *verified*. On the other hand, if the block has been split, this write is redirected to the victim cache, and the corresponding version counter is decremented by 1. If the counter value changes to 0, the store value is compared to the regular cache block for fault detection. If no fault is detected, the victim cache entry is then released, and the state of the regular cache block is set to *verified*.

5.2 Impact analysis

The aforementioned access strategy can be implemented as a small state machine that only checks the DSV vector and the counter to make decisions. The performance overhead introduced by this state machine is practically nonexistent since the decoding of the DSV vector can be performed in parallel with the comparison of tag values for hit/miss checking. In fact, only the accesses to a *split* block performed by the T thread will be delayed by one clock cycle, since these reads need to be redirected to the victim cache. However, these extra cycles do not necessarily cause the workload of the two threads to be globally unbalanced, since only the L thread but not the T thread would encounter cache misses. Therefore, the T thread, which spends extra cycles in accessing the victim cache, would eventually catch up to the L thread in execution when the latter is blocked on a cache miss.

The cache access control strategy indicates that during execution, the T thread is forced to wait for the L thread if and only if the value of the *access counter* is 0. The L thread, on the other hand, is forced to wait for the T thread in five cases: 1) The L thread tries to execute a load/store, while the *access counter* reaches its upper bound. 2) The L thread tries to read a cache block, while the corresponding *read counter* reaches its upper bound. 3) The L thread tries to write a cache block, while a checkpoint is pending to be established. 4) The L thread tries to split a cache block, while the victim cache is full. 5) The L thread tries to write a *split* cache block, while the corresponding *version counter* reaches its upper bound. These five cases constrain the amount of execution by which the L thread can run ahead of the T thread. Nonetheless, the occurrence frequency of these cases, except for the third one, can be effectively reduced by increasing the size of the counter and the victim cache.

It can be observed that the five conditions for blocking the L thread contradict with the condition for blocking the T thread, implying that **no** deadlock would occur in the fault-free case. More specifically, in the proposed scheme a deadlock can only be caused by a mismatch in memory access patterns. For instance, execution faults may cause the L thread to perform a number of extra read accesses, resulting in the L thread to block if any *read counter* has reached its upper bound. This blocking condition cannot be cleared, since the T thread would not perform these read accesses, thus never being able to decrement the counter value. Meanwhile, the T thread would also be blocked once the value of the global *access counter* becomes 0, thus creating a cyclic waiting condition. This condition, however, would never occur in the fault-free case, since the T thread is always able to unblock the L thread if the latter is blocked on any read counter. Therefore, the proposed cache controller will report an error whenever it detects a cyclic waiting condition, in turn causing the execution to be rolled back to the most recent checkpoint.

The organization presented in Figure 6 indicates that the proposed cache design can be implemented within a limited amount of extra hardware. Typically the victim cache only needs to contain 32 blocks. A 2-bit counter would suffice to record either the read count or the version number, if the corresponding cache block is not continuously read/written within a short period. The 3-bit DSV vector replaces the *valid* and the *dirty* bits in the traditional cache. As a result, for an $8K$ byte L1 data cache with $2K$ blocks, the extra storage required by the proposed fault tolerance technique equals $(3-2+2)*2K+32*32=7K$ bits. As a comparison, traditional SRT processors need to not only enlarge the reorder buffer, but also employ three centralized queues (so as to record load values, store values, and branch outcomes) with a total size of $(128+20+96)*32=7.6K$ bits according to the queue sizes reported in [8].

The proposed fault tolerance technique is also more power- and heat- friendly than traditional SRT processors. The three centralized queues in SRT processors are constantly accessed by both threads, thus not only consuming a large amount of energy, but also ending up becoming thermal hotspots which may degrade the reliability of the entire chip. In contrast, the proposed fault tolerance technique only employs a single centralized structure, the victim cache, which is accessed only when the corresponding cache block has been split. The remaining extra storage, the counters and the DSV vectors, is distributed into every cache block, while the number of cache accesses remains unchanged. As a result, each cache access only needs to read 5 extra bits, thus refraining from imposing significant power or heat overhead on the MPSoC system.

6. SIMULATION RESULTS

To evaluate the proposed cache-based fault detection and checkpointing technique, we have performed a set of experimental studies on the *Mediabench* [15] benchmarks. Meanwhile, given the ever increasing complexity of embedded applications, we have also selected a number of graphics and compression programs from the *SPECint 2000* set for evaluation. The SimpleScalar toolset [16] has been modified to model a dual-core MPSoC and to simulate the behavior of

Table 1: Impact of cache configuration and replacement policy on miss rate and checkpointing frequency

	Miss rate %				Checkpointing frequency (K insns/ckpt)			
	16K-dm	16K-2w (L/C)	32K-2w (L/C)	32K-4w (L/C)	16K-dm	16K-2w (L/C)	32K-2w (L/C)	32K-4w (L/C)
adpcm	0.203	0.203 / 0.203	0.198 / 0.198	0.198 / 0.198	3345	3345 / 3345	- / -	- / -
epic	5.340	4.104 / 4.421	3.898 / 4.049	3.793 / 3.899	1.755	24.05 / 62.75	65.31 / 137.4	89.59 / 186.5
gsm	0.023	0.006 / 0.006	0.003 / 0.003	0.003 / 0.003	121.6	1229 / 1326	117327/117327	234653/234653
mpeg2	4.753	0.621 / 3.644	0.264 / 1.482	0.253 / 0.606	1.769	22.48 / 149.1	99.67 / 506.1	418.5 / 2140
bzip2	1.931	1.371 / 1.389	1.309 / 1.327	1.302 / 1.309	0.687	18.29 / 25.81	32.89 / 46.29	84.90 / 130.2
eon	2.962	1.004 / 1.129	0.334 / 0.405	0.113 / 0.162	0.903	5.393 / 74.51	6.773 / 136.7	246.7 / 1693
gzip	4.782	4.437 / 4.459	3.930 / 3.938	3.936 / 3.897	1.839	9.791 / 13.10	17.49 / 23.72	40.62 / 72.98

Table 2: Overall writeback rate and checkpointing-induced writeback increase

	Writeback rate % ($(WB_{ex}+WB_{ori})/REF_{total}$)				Writeback increase (WB_{ex}/WB_{ori})			
	16K-dm	16K-2w (L/C)	32K-2w (L/C)	32K-4w (L/C)	16K-dm	16K-2w (L/C)	32K-2w (L/C)	32K-4w (L/C)
adpcm	0.023	0.024 / 0.024	- / -	- / -	0.017	0.016 / 0.016	- / -	- / -
epic	2.149	1.773 / 1.728	1.695 / 1.681	1.679 / 1.664	0.043	0.062 / 0.042	0.075 / 0.071	0.064 / 0.060
gsm	0.280	0.081 / 0.078	0.000 / 0.000	0.000 / 0.000	26.17	30.62 / 30.40	0.514 / 0.514	0.630 / 0.630
mpeg2	0.579	0.295 / 0.160	0.174 / 0.137	0.030 / 0.125	0.678	1.053 / 0.233	0.401 / 0.132	0.858 / 0.047
bzip2	4.139	1.033 / 0.919	0.869 / 0.788	0.724 / 0.665	3.115	0.657 / 0.482	0.460 / 0.328	0.233 / 0.135
eon	8.309	4.857 / 0.553	4.351 / 0.386	0.288 / 0.175	7.400	16.53 / 2.080	38.47 / 7.063	3.944 / 2.144
gzip	4.048	2.996 / 2.872	2.746 / 2.631	2.432 / 2.235	0.837	0.477 / 0.420	0.464 / 0.406	0.309 / 0.212

the proposed cache controller. In line with the fault tolerance literature [6, 8], we only measure the performance of the MPSoC without evaluating fault coverage.

Checkpointing frequency: The checkpointing frequency of the proposed fault tolerance scheme is determined by the frequency at which a dirty cache line is replaced, which is in turn determined by the cache miss rate and replacement policy. As a result, we retain the remaining architectural parameters, while simulating 4 distinct configurations for the L1 data cache: 16K directly mapped, 16K 2-way associative, 32K 2-way associative, and 32K 4-way associative. For each set-associative cache, we furthermore simulate two distinct replacement algorithms: the standard *LRU*, and a *Clean-first* policy that selects clean cache lines over dirty lines. The results on miss rate and checkpointing frequency are listed in Table 1, with a pair of values listed for each set-associative cache so as to clearly show the impact of replacement policies.

It can be easily seen from Table 1 that for most cases, the proposed scheme only requires a checkpoint to be established every tens of thousands of instructions. Embedded applications usually exhibit a smaller checkpointing frequency (for *adpcm* no checkpoints have even been taken for the latter two cache configurations) than *SPECint 2000* benchmarks, since the L1-data cache is able to absorb most of the load/store requests during execution. The checkpointing frequency of directly-mapped caches is usually high due to the large amount of conflict misses. Increasing the associativity from direct-mapped to 2-way associative, even if it cannot significantly reduce the miss rate for some applications (such as *adpcm*, *epic* and *gzip*), can always sizably reduce the number of checkpoints. More importantly, for set-associative caches, the *Clean-first* replacement algorithm can significantly reduce checkpointing frequency, yet at the cost of an increased miss rate, especially for 2-way associative caches. This is because the selection of a clean line over a dirty line for replacement may overwrite the clean data that has just been brought into the cache. However, except for *mpeg2*, this increase in miss rate of the other benchmarks is negligible.

Writeback frequency: Compared to a traditional write-back cache, the proposed cache needs to write a modified block back into memory not only upon a **replacement**, but additionally upon a subsequent **write-hit** if the block is at the *retired* state. To show the impact of the latter, Table 2 reports the overall writeback rate as well as the ratio of the extra writeback requests for each cache configuration. As can be seen, the overall writeback rate decreases as the cache size or associativity increases. The average writeback rate of set-associative caches is 1.3% for the *LRU* policy, and 0.8% for the *Clean-first* policy. In most cases checkpointing only causes a quite limited amount of extra writeback requests. Although manyfold increases can be seen in *gsm* and *eon* in the case of direct-mapped and 2-way associative caches with an LRU policy, the *Clean-first* replacement policy can effectively reduce these manyfold increases. Meanwhile, since the overall writeback rate is quite small, these manyfold increases do not cause sizable degradation in the overall performance, as will be confirmed subsequently when the results of CPI increase are examined.

Thread performance: In addition to the extra write back requests caused by checkpointing, the proposed fault tolerance scheme also affects the overall performance of the MPSoC in that the L thread needs to await the T thread under the five conditions discussed in Section 5.2, while the T thread requires extra time in accessing the victim cache and checkpointing the regular cache. The overall performance overhead is furthermore strongly affected by the initial execution offset between the two threads. A too small offset would result in the T thread quickly catching up to the L thread while the latter is blocked on a cache miss, thus causing the T thread to also wait for the missing data. In contrast, a too large offset would result in the L thread to split a large number of cache blocks, thus causing the T thread to spend extra cycles constantly accessing the victim cache. Taking both effects into consideration, during simulation we initiate the execution of the T thread upon any of the following conditions: 1) The L thread splits a cache block; 2) Either the read counter of a block or the global access counter is half full; or 3) the L thread generates a checkpoint request.

According to the results of checkpointing and writeback frequency, we select two representative cache configurations, 16K-2way and a 32K-4way, for performance simulation. We furthermore simulate **three** distinct configurations, a 16-entry victim cache with 2-bit or 3-bit counters, and a 32-entry victim cache with 3-bit counters, to evaluate the impact of the victim cache size and the counter size on the overall performance. In all three configurations, an *8-bit* global access counter and the *asynchronous* checkpointing model discussed in section 4.4 are employed so as to attain best performance

Table 3: Overall performance degradation in terms of CPI increase

	Counter 2 + Victim 16		Counter 3 + Victim 16		Counter 3 + Victim 32	
	16K-2w (L/C)	32K-4w (L/C)	16K-2w (L/C)	32K-4w (L/C)	16K-2w (L/C)	32K-4w (L/C)
adpcm	0.801 / 0.801	0.800 / 0.800	0.797 / 0.797	0.797 / 0.797	0.797 / 0.797	0.797 / 0.797
epic	0.632 / 1.413	0.589 / 0.605	0.490 / 1.271	0.446 / 0.690	0.423 / 1.204	0.360 / 0.605
gsm	6.328 / 6.327	6.321 / 6.321	6.286 / 6.285	6.279 / 6.279	7.451 / 7.450	6.068 / 6.068
mpeg2	0.819 / 9.467	0.948 / 2.207	0.820 / 9.445	1.223 / 2.238	0.822 / 9.428	1.217 / 2.287
bzip2	12.10 / 11.92	12.18 / 12.15	15.67 / 14.79	15.82 / 15.70	15.70 / 14.79	15.99 / 15.76
eon	5.576 / 2.842	5.258 / 5.227	5.376 / 2.568	5.698 / 5.587	5.401 / 2.674	6.581 / 6.307
gzip	4.274 / 3.918	4.172 / 3.737	4.276 / 2.351	4.177 / 2.015	4.275 / 2.410	4.176 / 2.016

results. The writeback latency is set to 3 cycles, the L1 miss penalty is set to 20 cycles, while the checkpointing latency is set to 50 and 100 cycles for the 16K and 32K caches, respectively. The obtained results of CPI increase are listed in Table 3, with the baseline MPSoC adopting an LRU policy yet the fault tolerant MPSoC adopting both the LRU and the *Clean-first* policies to clearly show the impact of the latter.

The results in Table 1 show that compared to *LRU*, the *Clean-first* replacement policy sizably reduces checkpointing frequency, yet slightly increases miss rate. As can be seen in Table 3, the *Clean-first* policy imposes a negligible impact on *adpcm*, *gsm* and *bzip2*, sizably reduces the performance degradation of *eon* and *gzip*, yet degrades the overall performance of *epic*, and especially of *mpeg2*. These data confirm that the impact of the *Clean-first* policy on the overall performance is determined by both checkpointing frequency and miss rate. Meanwhile, the results in Table 3 also show that although an increase in the counter size and/or the victim cache size can relax the conditions for blocking the L thread, it does not necessarily reduce performance overhead. A larger counter and victim cache typically causes a larger execution offset between the two threads, thus in turn causing more cache blocks to be split. As a result, the T thread needs to spend more cycles accessing the victim cache, while the L thread needs to await the T thread for more cycles at each checkpoint.

In sum, the proposed fault detection and checkpointing scheme causes a 0.4% to 16% increase to the execution time of non-fault-tolerant single thread architectures. As usual, embedded applications exhibit a smaller performance degradation than *SPECint 2000* benchmarks, while significant degradation (>10%) is only reported for *bzip2*. These results confirm that the proposed scheme outperforms the lock-step based fault tolerance CMP, which typically exhibits a performance overhead of 15% to 19% according to the data reported in [8], and incurs additional hardware costs, furthermore.

7. CONCLUSIONS

We have presented in this paper a light-weight fault tolerance framework which attains fault detection and checkpointing through sharing a single cache between two identical threads. By allowing both threads to write results into registers and the cache without comparison, the strict instruction-by-instruction synchronization model used in multithreading processors can be relaxed. Meanwhile, the cache design is modified so that each data block will be compared for fault detection before being written back, thus protecting the memory from being polluted by execution faults. Accordingly, in the proposed framework only the processor state needs to be checkpointed whenever a dirty cache line needs to be written to the memory, and the two threads only need to synchronize when a checkpoint needs to be established. A *split* capability is furthermore incorporated into the cache design to selectively duplicate a cache block if the L thread tries to update a block of which the old value is pending for the T thread to access, thus effectively relaxing the execution synchronization imposed by cache block dependences. The simulation results show that the average checkpointing frequency is as low as 1 per 30K instructions, with only a slight increase in writeback rate and a 1.4% to 10.4% degradation in performance. This high efficiency therefore enables the incorporation of the proposed framework into various types of embedded MPSoCs to attain both fault detection and execution checkpointing.

8. REFERENCES

[1] W. Wolf, "The future of multiprocessor systems-on-chips," In *Proc. 41st DAC*, pp. 681–685, Jule 2004.

[2] International Technology Roadmap for Semiconductors (ITRS), 2007 Edition. "Process integration, devices, and structures".

[3] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," In *Proc. DSN'02*, pp. 389–398, 2002.

[4] K. D. Wilken and T. Kong, "Concurrent detection of software and hardware data-access faults," *IEEE Trans. on Computers*, 46(4):412–424, April 1997.

[5] S. K. Reinhardt and S. S. Mukherjee, "Transient-fault detection via simultaneous multithreading," In *Proc. 27th ISCA*, pp. 25–36, June 2000.

[6] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," In *Proc. 29th ISCA*, pp. 99–110, May 2002.

[7] T. N. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," In *Proc. 29th ISCA*, pp. 87–98, May 2002.

[8] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," In *Proc. 30th ISCA*, pp. 98–109, June 2003.

[9] A. Wood, "Data integrity concepts, features, and technology," *White paper, Tandem divison, Compaq Computer Corporation.*

[10] N. S. Bowen and D. K. Pradhan, "Virtual checkpoints: Architecture and performance," *IEEE Trans. on Computers*, 41(5):516–525, May 1992.

[11] P. A. Lee, N. Ghani, and K. Heron, "A recovery cache for the PDP-11," *IEEE Trans. on Computers*, C-29(6):546–549, June 1980.

[12] Y. Zhang and K. Chakrabarty, "Energy-aware adaptive checkpointing in embedded real-time systems," In *Proc. DATE'03*, pp. 918–923, 2003.

[13] D. B. Hunt and P. N. Marinos, "A general purpose cache-aided rollback error recovery (CARER) technique," In *Proc. FTCS-17*, pp. 170–175, 1987.

[14] K.-L. Wu, W. K. Fuchs, and J. H. Patel, "Error recovery in shared memory multiprocessors using private caches," *IEEE Trans. on Parallel and Distributed Systems*, 1(2):231–240, April 1990.

[15] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for Evaluating and Synthesizing Multimedia and Communications Systems," In *30th Micro*, pp. 330–335, Dec. 1997.

[16] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: an infrastructure for computer system modeling," *Computer*, 35(2):59–67, Feb. 2002.