Cache Based Fault Recovery for Distributed Systems

Avi Mendelson
Dept. of EE, Technion, Haifa, Israel
medlson@ee.technion.ac.il

Neeraj Suri*
Dept. of CIS, NJIT, Newark, NJ
suri@cis.njit.edu

Abstract

No cache based techniques for roll-forward fault recovery exist at present. A split-cache approach is proposed that provides efficient support for checkpointing and roll-forward fault recovery in distributed systems. This approach obviates the use of discrete stable storage or explicit synchronization among the processors. Stability of the checkpoint intervals is used as a driver for real time operations.

1. Introduction

Recovery from transient faults to sustain a system's functional and temporal requirements constitutes a much researched area for dependable distributed systems. Various hardware and software checkpointing schemes are proposed to constrain the recovery times by rolling back to a stable system state and restarting. However, roll-back recovery, by its very nature of rolling-back to a prior consistent operational state and re-trying the operation, involves a time penalty of lack of forward progress while the retry operation is performed. Thus, for systems with real-time deadlines or other response critical systems, the roll-back recovery approach is not particularly attractive for its latency characteristics, and has resulted in the development of roll-forward fault recovery techniques.

We provide a brief introduction of the generalized roll-forward procedures before detailing the proposed approach. The classical roll-forward recovery techniques is based on a pair (or set) of synchronized processors providing task redundancy by executing an identical task code, i.e., duplex configuration. Periodically, each processor creates a checkpoint and stores the process state information at that instant in a stable storage unit, and continues operation till the next checkpoint. A duplex configuration requires a comparison mechanism to check for the consistency of the checkpointed information for the two processors. A discrete processor, termed the *checkpointing processor*, compares the state of the processes at the checkpoint, and if a inconsistency is discovered, the checkpoint processor directs a spare processor to load the processor state recorded at the last[1] consistent checkpoint and execute the uncertain interval again. Assuming single transient fault occurrence in the system, and for a fault-free spare processor, the spare processor is used to provide a reference task execution. The comparison of the results obtained from the spare processor to the results obtained by the duplex processors, helps determine which of the two processors in the duplex pair was faulty at the previous checkpoint. As a recovery procedure, the identified faulty processor, discards its state information and loads the state information from the fault- free processor for subsequent operations. If multiple faults are allowed, the spare processor may be faulty as well (with a very low probability). In this case, the information of the spare processor may not match any of the duplex processors, thus, both processors roll back to the last consistent checkpoint [2]. Figure 1 elucidates these operations.

As, primarily, transient faults are of concern, this procedure assures the data consistency and operational consistency to be maintained for subsequent operations. The process of roll-forward recovery does inculcate the additional overhead of requiring a spare processor. However, such recovery approaches mitigate the fault recovery time and performance degradation of the roll-back recovery schemes, as the duplex pair sustain their ongoing operations while the spare processor concurrently executes the comparison process.

Roll-forward recovery has generated much recent interest, and a variety [14, 10, 11] of the roll-forward protocols appear in literature. However, the traditional emphasis has been on obtaining performance metrics

*Supported in part by DARPA Grant DABT63-96-C-0044

[1] The prior checkpoint is discarded only if the current checkpoint is found to be consistent.

[2] The probability of that operation is very low, and in the worst case the performance of the roll-forward scheme matches that provided by the classical roll-back schemes.

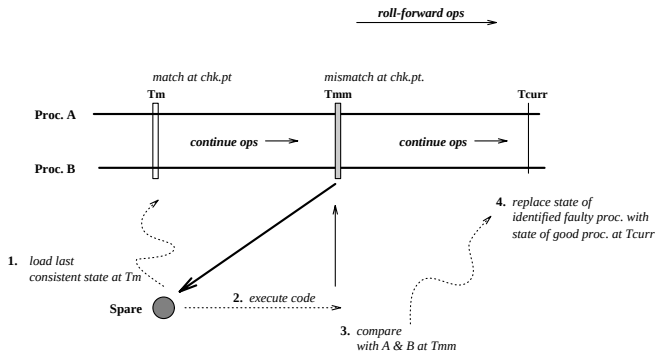

Figure 1. Generalized Roll-Forward Recovery

for the roll-forward implementations, and a number of important design and implementation facets, especially for real time systems, have not been adequately addressed. We highlight some considerations of interest, that constitute the focus of this paper:

(1) Can a cache based approach provide architectural support to facilitate roll-forward techniques? Is stability and predictability of checkpointing achievable using caches?

(2) How and when are checkpoints created and consistency achieved?

(3) What is the checkpoint content? Can a discrete "stable storage" for saving checkpoints be eliminated?

(4) Is checkpointing for roll-forward recovery feasible if the processors are not explicitly synchronized (maintaining synchronization causes significant overhead)

Furthermore, the prevalent approach is to save the state information in a stable storage unit and assume fault-free conditions on this mechanism. The cost and retrieval time parameters of the stable storage can significantly impact the recovery performance of the roll-forward mechanism, and are generally underemphasized in roll-forward implementations. This concern further motivates our utilization of a cache-oriented stable storage mechanism.

In this paper, we address the abovementioned set of concerns through a novel cache approach termed "bi-directional cache", which supports the roll-forward mechanism efficiently and provides natural run-time checkpointing capabilities. Our drivers are efficient checkpointing and obtaining a stable, predictable checkpointing operations using the cache to support response critical fault recovery.

1.1. The Cache Perspective to Checkpointing

Existing cache based checkpoint approaches need special enhancements of the processor and the cache subsystems (note that in modern processors, the first level cache is used as part of the processor architecture). The processor enhancement is needed for fast status capturing and the cache modifications needed to maintain the checkpoint creation protocols such as CARER [7].

In a recent paper [8], the performance liability and lack of predictability in using caches for a periodic checkpointing scheme was demonstrated. The major objection to the usage of the cache was the uncontrollably high variability in the cache based checkpointing intervals. For a real time system where predictability of recovery time is a prime driver, this characteristic of caches effectively restricts their usage. However, in the proposed bi-directional cache this specific cache-based system liability is alleviated by developing a new cache structure which provides for stable, low-variance checkpoint intervals. The checkpoints creation process is non-scheduled which is automatically established at run-time on a cache-line replacement instance. The use of a partitioned cache obviates the need for a discrete stable storage unit and also alleviates the associated physical and performance costs of such storage units. Furthermore, the techniques do not impose the strong assumption of requiring synchronized processors. The checkpointing strategy presented is a generalized one to support alternative fault-recovery mechanisms such as roll-back recovery too.

The organization of the paper is as follows. In Section 2 we describe the system model and also build up the rationale for the use of a cache based strategy furthering the basic cache usage discussions of [7, 2, 15, 13]. Section 3 details the proposed bi-directional cache architecture, and provides a discussion on some of its relevant properties before addressing roll-forward strategies based on this cache architecture. Sections 4 and 5 provide an operational analysis and simulation results for the proposed the roll-forward fault recovery strategies, and we conclude with a discussion section.

2. System Model

As we have mentioned earlier, the recovery mechanisms of interest apply to systems using redundancy of task executions. Also, the model is similar to the one used in [14] which considers duplex processor entities with a small set of discrete spares (either fixed or floating spares). Each system has a local bi-directional

cache with access to shared memory. The use of the new bi-directional cache architecture allows avoiding the use of discrete stable storage units which are necessarily required in virtually all other (cache and non-cache based) fault recovery schemes. The state information at the checkpoints stored in the main memory and in the caches. The state information at the checkpoints comprises of only the processor registers, the program counter and the write-cache dirty variables. These aspect are detailed in Section 4.

Usually, a discrete fault free unit is assumed to facilitate checkpoint comparisons across the system. Such a unit should identify the information that have been modified since last breakpoint and compare it with the other checkpoints as in [15]. The new bi-directional cache based architecture, keeps all the modified information in the write-cache, so that the comparison activity can be simplified to be implemented as a simple state machine as part of the controller. Also, the state comparisons can be accomplished by associating signatures with the checkpoint information so that only small amounts of information need to be compared between the duplex processors.

2.1. Caches for Checkpointing: Rationale

As memory caches are the repository of the information that is most recently used by a processor; if the following rules are observed, then the cache holds information that can be used for creating run-time checkpoints:

- The cache update policy is a write-back approach, with information written back to main memory only when a "dirty" line/variable (i.e., cache line that was modified.) needs to be replaced.
- At the first instance a dirty line is written back to the main memory, the cache is flushed; i.e., all the dirty cache lines are written at once to the main memory.

Lemma 1 *If the cache flush process as described above, is fault free, and the processor status (registers and flags) are kept as part of this process in a fault free memory location, then the process can always roll back to the last flush point by discarding the contents of its cache and through reload of the processor state.*

Proof: The cache contains all the differences between the program state at the last "flush point" and the main memory (since no information was written to the main memory from that time on). Thus, by invalidating the contents of the cache and reloading the processor state at that point, we establish consistency in the data content of the cache and the main memory. □

Cache models, such as [5, 1, 12], indicated the existence of working sets during the executing time of the program. These assume that the miss ratio within the working set can be considered as a constant and only when the program moves from one working set to another, a burst of cache misses can occur. If we assume a uniform distribution of the write operations among the memory references, and consider a fully associative cache (or a high degree of cache associativity), we can expect that the rate that write-cache lines are replaced from the cache will be uniform within the working set and may present shorter time intervals as the program switches working sets. This is very important designing checkpointing mechanisms that can operate "on the fly" and incur only a modest, if any at all, overall slowdown of the system.

3. The Bi-Directional Cache

Cache memories, classified according to their data updating policy are of two generic configurations, namely: the *"write-through"* cache updates the cache and the main memory simultaneously in order to keep both memory hierarchies consistent. This requires the processor need to idle until the main memory is updated. The *"write-back"* cache presents a different approach; it updates the main memory only when a modified cache line (termed *dirty*) is replaced in the cache. Thus, if the program displays a high degree of locality of memory references, the number of write operations are reduced considerably. The proposed architecture partitions the cache into two distinct subsystems, each responsible for discretely handling the *read* and *write* operations between the processor and the main memory. We refer to this partitioned entire cache as the *bi-directional cache*. It has been shown that the *bi-directional cache* causes lower write traffic rates than the write-through cache, and furthermore displays better performance features than even the write-back cache[4, 13, 9].

The bi-directional cache (Fig. 2) consists of two subsystems distinguished by their functions, namely: a *read-subsystem* and a *write-subsystem*. The *read-subsystem* handles fetching information from the main memory and the *write-subsystem* controls the updating of the main memory. We use a write-through cache as a *read-subsystem* and disconnect its write signals. The *write-subsystem* includes a small write-back cache that uses an allocate on a write miss and a no-allocate on a read miss policy. Based on earlier work, [9, 13], we locate write-cache between the processor and the write-buffer. This configuration allows us to add the *write-subsystem* to existing cache based systems without the

need for processor or cache enhancements.

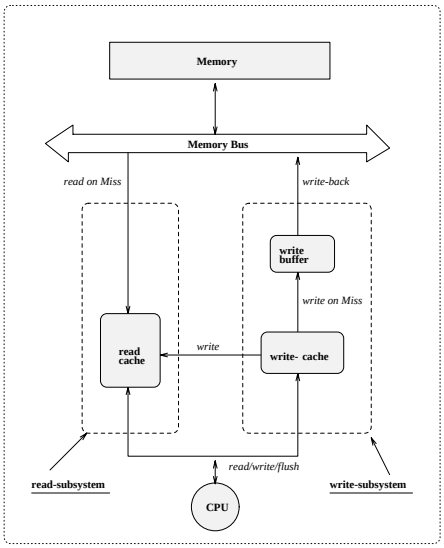

Figure 2. The Bi-directional Cache System

All the read and write references are issued to both subsystems. On a read reference, the *read-subsystem* has priority over the *write-subsystem*. On a write reference, the subsystem sustaining a hit, updates its entry (both subsystems may update their entries respectively). Thus, all the dirty lines of the *read-subsystem* are present in the *write-subsystem* as well. Splitting the cache into the read and write partitions also enables the read and write operations to have connotations of concurrency of operation. While a line is written to the main memory by the write-cache, the CPU can continue to read from the read-cache[3] Also, the split operations do not result in any overheads as the concurrency can occur in parallel to the CPU operations.

The design parameters (internal organization, line size, etc.) of the *read- subsystem* can be different from the design parameters of the *write-subsystem*. For a class of system operations (e.g., spice, lisp, TeX), simulations suggest an efficient organization of the *read-subsystem* to be 2-4 set associative with a 32-64 bytes cache line size. The size of the read-cache can range from 8 Kbytes to 1 Mbytes. The associated parameters for the *write-subsystem* are: line size of 4 bytes (1 word), and 8 ways set associative (or fully associative). The size of the write-cache subsystem will determine the frequency the checkpoints are created and will be discussed later.

[3]assuming data consistency being maintained by the inclusion property discussed later.

3.1. Operations of the Bi-directional Cache

We summarize the bi-directional cache operations by describing the processes the *read* and the *write* subsystems perform upon receiving a read or write reference from the processor:

Read Op:

- On a hit in the read-cache, the word is fetched from the *read-subsystem*. On a miss in the read-cache and a hit in the *write-subsystem*, the information is retrieved from the *write-subsystem*.
- On a miss in both subsystems, the information is fetched from the main memory to the *read-subsystem* and a data coherency protocol (see below) is applied to sustain data consistency between the read and write-caches (The line sizes of the read-cache and write-cache are different, which is another driver for the consistency check.).

Write Op:

- On a hit, the new value is written to the subsystem it hit in (the *read*, the *write* subsystem or both) and updated across the cache sub-systems.
- On a miss, the new value must be written to a new write-cache entry. If the write-cache is full, the CPU writes an entry to the write-buffer and replaces its value. If the write-buffer is full, the CPU stalls until it can replace the write-cache entry.

Ensuring Coherency:

- When a cache line is fetched from the main memory to the read-cache, its addresses should be checked against the content of the *write-subsystem*. If the cache line, or part of it, is found in the write cache, this *conflicting* entry must also be updated in the read-cache. This is a necessary operation to guarantee the data variable in the read-cache to reflect the current value of such a matching address. The difference in the line-size of the read-cache and the write-cache also necessitates this operation.

The bi-directional cache is shown [13] to provide better overall performance (throughput, response time, frequency of cache flushes) as compared to both the

write-through and the *write-back* caches. We refer the reader to [13] for more architectural and operational details. For the purpose of this paper, we focus on the results that support the roll-forward recovery and checkpointing considerations.

4. Support for Roll-Forward Recovery

This section describes the basis for different versions of the bi-directional cache that are aimed at supporting the roll-forward mechanisms. The techniques are based on a two-way/three-way partitioning of the *write-cache subsystem.*

We emphasize that the concept of checkpointing is meaningful only if the cache partitions (the read and write-caches) can maintain consistency of data (data "inclusion" property), both within each cache structure and also across the system cache of different processors. The inclusion property between the read and the write subsystems requires the consistency of data for valid data kept in the write-cache and read-cache of the bi-directional cache. For the purpose of checkpointing, we require all dirty variables of the read-cache to be present in the write-cache. Basically, the write-cache contains all the dirty variables/lines to be updated to the main memory. A simple strategy on achieving and maintaining the consistency property within a bi- directional cache is utilized, namely:

Algorithm 4.1 *Forcing inclusion between the read and the write subsystems.*

- *On a write-miss, both caches allocate information from the main memory.*
- *On a write-hit in the read-cache only, the dirty variable is also written into the write-cache.*
- *When a dirty line is replaced in the read-cache, the write-cache must first write its value to the main memory and flush its entry.*
- *When a cache line is written to the main memory, it clears its dirty bit in the read-cache.*

It is pertinent to note that the sizes of the cache lines that each cache allocates from the main memory (on a write-miss) may be different. In order to avoid the use of dirty bits in the read subsystem, it is sufficient to require that the read-cache report to the *write-subsystem* on any line replacement. Thus, the write-cache could write its value back to the main memory (a similar mechanism is used when a first-level cache needs to keep the inclusion with the second-level cache).

4.1. Creation of Checkpoints

The process for the selection and physical creation of the checkpoints[4] is not a trivial one. However, this important facet is generally under-emphasized with the assumption that this function is implicitly present. The bi-directional cache provides a procedure for natural determination of the checkpoints without resorting to any specialized supplementary mechanisms.

The write-cache in the bi-directional cache provides a logical basis for the triggering of run-time checkpoints. Other cache based checkpoint mechanisms suggest creating a checkpoint every time a dirty cache line should be replaced from the cache. The checkpoint information in this case contains the processor information and all the information that had been changed from the last checkpoint creation (all the dirty cache lines). CARER[7] tried to improve the performance of such mechanism and suggests changes in the replacement mechanism so that clean cache lines will be replaced before dirty cache lines. Such a modification can indeed slow the cache operation significantly. The bi-directional cache achieves the same functional effect by using the write-subsystem, especially since the dirty lines are handled in a special cache that specifically handles dirty cache lines. This provides a simple solution without the performance degradation implications.

The following sections briefly describe the basic process for the bi-directional cache checkpointing.

Definition 4.1 *A checkpoint contains all the necessary information needed for the process to continue its operation at a future time from that point.*

A checkpoint based recovery mechanism has to define **(a)** the time instant when the checkpoint is created, **(b)** the nature of information kept at the checkpoint, and **(c)** the process used to recover if an error is detected.

Algorithm 4.2 *Check point based recovery:*

Checkpoint triggering *: A checkpoint is created <u>each</u> time a dirty cache line is replaced from either the write-cache or from the read-cache. (This follows directly from the conditions required to sustain the inclusion property with the main memory)*

Checkpoint creation *: The checkpoint creation process requires that* **(a)** *the processor status information will be saved to a* "safe" *area, and* **(b)** *to save all the dirty cache-lines of the the write-cache and*

[4] In both roll-back and roll-forward recovery.

mark them as "clean" (this is important as unlike the conventional cache updating of the main memory, we will be requiring a time delay before the update operation).

Recovery *: This is the roll-forward mechanism described in subsequent sections.*

The implementation of these mechanisms may be dependent on the technology being chosen. We discuss some of these aspects in the next sections.

We note that as long as the cache does not update any of its dirty lines back to the main memory, we can roll-back to the last checkpoint by simply reloading the processor information and invalidating the read-cache. This invalidation of the read-cache is necessary to force all the information to be fetched from the main memory. Since the write-cache contains all the modified information at that point, the coherence mechanism between the write-cache and the read-cache will force the correct data to be fetched to the read-cache. Since only the processor's status is kept in the "safe" area during the checkpoint creation, we assume that area to be part of the processor or an internal buffer of the bi-directional cache architecture. It follows directly from the usage of a distributed cache-coherency protocol, and the fact that identical code is executed on different processors, that the checkpoints are consistently set for each duplex pair.

4.2. Performance Characteristics of Checkpointing

The design of the roll-forward mechanism is based on the assumptions that: **(a)** faults are infrequent and **(b)** only transient faults are considered. Thus, from the performance point of view it is especially important to optimize the processes of creating the checkpoint so that it could be done "on the fly"; i.e., to minimize the overhead caused by that process. One can observe that this requirement imposes that the entire checkpoint creation process must be completed before the computational interval that follows it. Otherwise, the system needs to be stalled until the checkpoint creation process is completed.

It is also important to determine the actual performance cost implied in setting the checkpoints. As the checkpointing is of a non-scheduled non-periodic nature, an immediate concern is to determine the number of checkpoints established relative to the total number of memory references. A high number of checkpoints is not desirable, as this involves continual updating of memory and imposes a natural performance limitation on the roll-forward process. We start our performance measurements with a comparison of how frequently the write-back caches create breakpoints versus the bi-directional cache systems.

The baseline configuration of the write-back cache was chosen (see [13] for detailed rationale) to be: cache line: 32 bytes, four-way cache associative with 128 sets (16K cache). The baseline configuration of the bi-directional cache was chosen as: cache-line: 4 bytes (this number was supported by performance results presented in [9]) and write-cache of size of 256 entries. (the write-cache is organized as fully associative or as an eight-way cache associative).

The first set of plots present the frequency the checkpoints are created and the time it takes to create a breakpoint under different software environments. We choose a set of 5 traces, given as part of the Dinero traces. The simulations of the write-back cache are marked as $B-*$, while simulations of the write-cache are marked as $C-*$. Figure 3 depicts the number of instructions executed before a checkpoint gets created for different applications and hardware configurations. The write-back configuration keeps the same cache lines and set associativity during this experiment. Note that the size of the write-cache was taken to be vary from 4 to 32 Kbytes, while the size of the write-cache was chosen to be between 256 bytes to 2K. The results indicates that for most of the applications presented, the use of write-cache significantly reduces the frequency for the caches need to create the breakpoints. The set of graphs in Figure 4 highlight that the frequency with which the bi-directional cache is flushed does not depend on the size of the read-cache but is a function of the size of the write-cache only. On the other hand, the frequency of the flushes in the write-back depends on both the applications and the size of the cache. It is again pertinent to point out the low variation in the cache flushing frequency for the bi-directional cache.

Figures 3 and 4[5] illustrate results from simulations which highlight the stability of the number of cache flushes, i.e., the number of checkpoints established. The variability in the number of cache flushes and the time required for cache-flushes in a conventional cache is one major reason why the existing checkpointing strategies are not based on the use of caches. The bi-directional cache not only provides stability in terms of the number of flushes, but also in the time required to complete the cache-flush operation (see Figure 5). The combination of these parameters supports our basing the checkpointing on the cache.

[5] A wider selection of benchmarks are used for simulations, over a variety of applications, for Fig. 5 and 6 than the other graphs simply to establish a greater confidence in the stability aspects of the cache.

Figure 3. Frequency of Cache Data Flushes

Figure 4. Frequency of Cache Data Flushes: *read-cache 2-way associative, 32byte block size; write-cache 256bytes, fully associative, 4byte block size.*

Figure 5. Average time required to flush the cache. *Cache/read-cache 2-way associative, 32 byte block size. Write-cache 256bytes, fully associative.*

It is also relevant to quantify the overhead required for the checkpointing strategy, and that the total overhead resulting from the checkpointing scheme be small enough not to impose performance degradation on the system. A variety of workload profiles ranging from TeX operations to *cc* compilation processes were studied to establish this metric. The overhead results from the CPU having to stall as it waits for the write-buffer to complete the updating activity to the main memory, as well as from the time required for the write-cache to write its values to the write-buffer. We do emphasize that the overhead is relative to the natural overhead entailed on the operations of bidirectional cache. The average overhead was observed to be 6.8%. The use of mechanisms (as in [13]) to reduce the CPU stall times can reduce the overhead to as low as .5 - 2 %. Overall, we have shown the simplicity of physically implementing the checkpoints and the procedures for maintaining their consistency across the processors. Also, the low time cost and low overhead readily support our choice for cache-based checkpointing.

Next, we consider how different design parameters of the write-cache, such as the cache size, affects the average, minimum and maximum sizes of the computational intervals, and so, the time the system has to efficiently create the checkpoints. We start with a comparison of the average time between checkpoint creation when the traditional write-back cache is used and when

Figure 6. Cost of breakpoint creation (SRAM model)

Figure 7. Cost of breakpoint creation (DRAM model)

the new bi-directional cache is used. Note that when the bi-directional cache architecture is considered, only the size of the write-cache affects the time between checkpoints; whereas when only the write-back cache is considered, the size of the entire cache gets factored in.

In order to estimate the average time it takes to manipulate the roll-forward algorithm and to create a checkpoint, we assume the following timing calculations: for the write-back cache, the time needed is: *time to search the entire cache (size of the cache) in order to create the signature + time to update the main memory (number of dirty lines in the cache) + communication time*. For the write-cache, the signature can be calculated during the update time, similar to the method used by the ECC mechanism, so the time required is: *time to update the main memory (size of the write-cache) + communication time*. As the communication time is of the same order of magnitude for both caching schemes, we can ignore it for the time being. The subsequent plots compare the time it takes to create a single checkpoint, given different cache size (write-cache size for the bi-directional cache architecture) Figure 6 presents a SRAM memory system where the access time to the main memory consist of 5 cycles set up time and a one cycle for each bus transaction (we assume 32bit bus). The second model, Fig.7, assumes a DRAM model where the setup time is reduced to a single cycle.

Note that the cost of creating checkpoints is the same for all the write-cache applications since the cache line size is fixed. We observe that when the DRAM model is used, the time to create a break point for the C-applications is larger. Thus, when the SRAM model is used, the write-cache architecture presents results in a faster handling of breakpoint creation. Also, as the creation of breakpoints under the write-cache is infrequent compared to the write-back cache, the overall handling of checkpoints using the write-cache turns out as the more efficient of the two.

In order to estimate the overall performance penalty caused by the different schemes, we consider a comparison of the average, min and maximum interval length, in conjunction with the average time it takes to create a new breakpoint. In most cases, the time required to establish a new breakpoint is lower compared to even the minimum interval for the write-cache architecture and the average interval time is much longer than the time needed to create the new checkpoint. Thus, the breakpoint can be created on the fly.

In the next section we provide enhancements to the basic bi-directional cache subsystem. These variations are designed to facilitate the the checkpoint creation and the roll-forward manipulation to be done "on the fly"; i.e., in parallel to the execution of the next computational interval. Thus, an overhead is incurred only when the checkpoint creation time is longer then the computational segment which is executed in parallel to it.

Results indicate, that a write-cache which is larger

then 1-2Kbytes can ensure the computational interval to be sufficiently long to ensure that the fault forward mechanism can be executed "on the fly"; at the same time large write-caches can result in computational intervals which are too long. Extreme differences between various computational intervals can also detrimentally affect the performance of the system in the instances of fault occurrence[11]. Thus, we may wish to limit the time span of the maximum interval. We do emphasize that the write-cache architecture can better take advantage of its size, while in many of the write-back applications, increasing the size of the cache does not improve the average interval's length.

5. Implementation of the Roll-Forward Bi-directional Cache Architecture

In order to avoid the penalty involved in copying the lines of the write-cache to the write-buffer, we propose enhancements to the basic write-cache architecture. We propose to partition the write-cache such that an implicit stable-storage unit is created. The number of partitions and the nature of the data movements across the partitions determines the properties of the roll-forward technique. We present two versions of modifications to the regular bi-directional cache architecture to implement roll-forward recovery.

The first enhancement calls to use a two-way partition of the write-cache. We use two sets of write-caches in the *write-subsystems* with one used as an *active* write-cache while the other is used as a *backup* write-cache. When a checkpoint needs to be created, the processor status is kept in a pre-designated safe area. If the backup cache is empty, the system switches between the active cache and the backup caches, and the processor can continue its operation. If the write-cache is full, the system switches the write-caches and continue its work by using the new write-cache. Concurrently, the "old" write-caches exchange signatures to validate that the information is consistent.

Meanwhile, the backup cache checks its validity with the backup cache of the other processors. As before, the comparison is based on exchanging signatures between the duplex write cache subsystems. If the caches were found to be consistent, the main memory is updated, the cache is marked as empty, although the last consistent processor status is still stored.

On the other hand, if the caches are not consistent, the write-cache does not update the main memory, so its update level remains the same as the last valid checkpoint. So, the backup processor can be loaded with the processor information (registers etc.) of the last valid checkpoint (which is kept in the active-cache area) we have, and continue its computation from that point. As soon as it reaches the checkpoint, it should compare its signature with the other two signatures and only the one which was found to be valid should update the main memory and the other processor. As the main memory data is consistent with the previous (valid) checkpoint, it can start performing the computations which were discerned as inconsistent among the duplex processor pair. As soon as the spare processor reaches the checkpoint (note that the structure of the write-cache automatically forces it to be the same checkpoint as the one for the duplex pair), by comparison its results (the spare forms the reference point) with the results of the duplex pair processors, it can help determine the faulty entity.

After the two processors reach the next checkpoint (in the case of a fault), they need to be stalled[6] until the spare processor has completed the determination of the faulty processor; and the correct cache will update the main memory and will also update the processor status of the processor identified to be faulty, with the state of the non-faulty processor. At this point, consistency of state is restored amongst the duplex pair and the two processors can continue their operations.

It is of interest to note that:

- In a fault-free scenario the performance of the system is not degraded. The checkpointing simply records the instances of the dirty line replacement, and is not detrimental to the cache operations.

- The time cost for establishing checkpoints does not affect the basic cache-access times, as the operations can be performed independent of each other.

The two-way partitioned bi-directional cache can help create checkpoints "on the fly", but it still requires the processors to stall while the spare completes execution. We can further enhance the checkpointing efficiency by using a three-way partitioned *write-subsystem*. One partition is used as the active write-cache, the second as a backup memory and the third is used to manage the memory update mechanism in parallel to the other system activities. The result is a minimal performance impact on the regular system operations.

This technique has connotations of a pipeline strategy. The improved mechanisms allows the system to continue two segments ahead before being required to idle if the write-cache is not empty. When no fault is present, both the two-way and three-way partitioned

[6] This is an inherent characteristic of any roll-forward strategy, as the duplex pair is allowed to continue its operation, only, till the next checkpoint. The next bi-directional cache strategy, we propose, alleviates this problem.

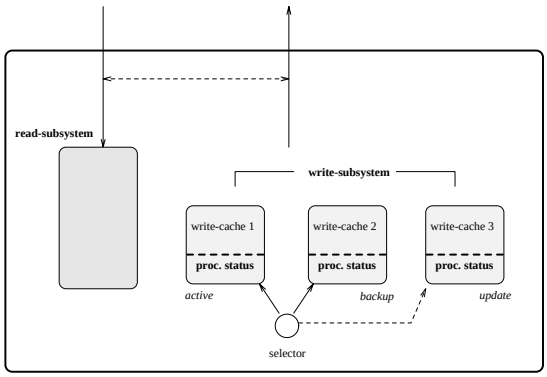

Figure 8. Roll-forward: 3-way partitioning

caches display similar operational behavior. When the processor reaches the point where it needs to create a checkpoint, it saves its internal status and uses the update-cache as the active-cache. Thus, the old backup-cache becomes the new update-cache and the old active-cache becomes the new backup-cache.

When a fault is discovered (the backup-caches are not consistent), the spare processor is activated as before. When the two processors reaches the next checkpoint, they do not have to wait for the spare processor to complete its operation or to the backup-cache to complete the update of the main memory. The processors can continue their operations by using the update-cache as another temporal storage entity. Since the memory update time is much faster then the time it takes the processor to reach the next checkpoint, the probability that the spare will made its decision and the main memory will be updated before the "regular" processor will form the next break point is virtually unity. We do mention that the primary purpose of allowing a two segment propagation is to provide the spare processor sufficient time to execute the inconsistent interval and identify the source of discrepancy in the duplex pair. The synchronization of the processors is again enforced automatically, and implicitly, based on the bi-directional cache operations. Furthermore, the *inclusion* stipulation, both within the cache partitions and across the system caches, directly implies the conditions of consistency of checkpoints and that of data to hold across the system; thus, the assertion of correctness of the roll-forward procedure.

5.1. Is Synchronization Across the Processor Pair Required?

It needs to be emphasized that we have not imposed the requirement of the two processors being clock synchronized with each other. As the processors execute identical task code, start a process with identical cache states, and their caches are filled at the same execution point; thus, it does not matter if one processor reached the checkpoint a little earlier or later than the other processor, and there is no data consistency violation. It is interesting to note that if one processor sets a checkpoint and the other processor fails to set its checkpoint, then this is automatically equivalent to the case of an inconsistent checkpoint. The nature of operation of the write-cache also implicitly forces the two system to determine the checkpoint at the same execution state. Hence, it suffices to implement a synchronization mechanism between the backup-cache of the two system so that we compare the correct versions, round # or any other suitable identifier for the checkpoints. Furthermore, in order to compare the consistency of the checkpoints we simply need to send any kind of signature to the other cache. This can be accomplished by the use of a CRC or any other form of signature for the state information.

The coherency of the data/variable updating in the main memory also follows implicitly. If one processor tries to update a data variable in the main memory, it establishes its checkpoint at this instance of cache-line replacement. Under fault-free conditions, the second processor also computes an identical value of the variable to be updated. For the checkpoints set by the two processors being consistent, only then does the data variable get updated in the main memory, else the inconsistent checkpoint triggers the fault recovery process. This is nothing but a form of coherency of data updating.

6. Discussion and Conclusions

We have presented a novel cache architecture which readily lends itself to the support of the roll-forward techniques. The natural checkpointing strategy provides an efficient approach which can also be used for conventional roll-forward and roll-backward protocols. The need for a discrete stable storage unit has been done away with. Furthermore, we have presented a logical synchronization between the duplex processors which avoids the use of explicit processor/checkpoint synchronization procedures required for conventional recovery approaches. We have currently provided simulation results in the paper, and we are developing analytical models for a more detailed performance analysis of the roll-forward mechanism.

We have discussed the checkpointing and fault recovery operations of the bi-directional cache based on the observed stability of (a) frequency of checkpointing, and (b) stability in cache flush timing over a range

of application programs, and (c) that the size of the write cache can be modified to obtain the desired checkpointing interval lengths. Another form of control is to deliberately cause checkpoints to be established. This can be as simple as sending a dummy read request from the processor to the bi-directional cache for a reserved memory variable thus forcing a checkpoint. A NOP operation or a similarly designated operation can be triggered based on the number of instructions from a prior checkpoint or simply based on a counter value.

Our recovery scheme tries to optimize the performance of the system. On one hand, we wish to set the checkpoints as far apart as possible, so that the potential overhead caused by the checkpoint creation will be as low as possible. On the other hand, we wish to set the checkpoints as close as possible so that the recovery time will be as efficient as possible. The use of the write cache seems to be a natural choice since most of the programs present a stable rate of writes to different locations in the main memory. Thus, if the interval is too long, we can always deliberately insert checkpoints to control the length of the interval. But, this technique can not be used to control the checkpoint creation when it is too frequent. For that purpose, a larger write cache can help.

It is interesting that most existing techniques require periodic, equi-distant checkpoints to be established, although the determination of the actual checkpoint interval based on real program information is not addressed. In [6] the authors select a 2-minute checkpoint interval without a rationale on how the 2-minute number is established, beyond simple observation.

In an actual system, the frequency of the checkpoint is very much dependent on the nature of the program application, and also on the type of system model under consideration. It is our assertion that the bi-directional cache and other cache-based checkpointing mechanisms provide a naturally optimized checkpoint interval i.e., if a periodic checkpoint is optimized for placement of checkpoints according to the program and memory access patterns. A simple performance analysis to compare the efficacy of such natural, aperiodic checkpoints with periodic checkpoint determination establishes this assertion.

The bi-directional cache structure also provides for handling of I/O and interrupt driven checkpoint insertion. We transform both I/O and interrupt calls to the as a dummy memory read operation inserted into the program stream of the duplex processors at the same PC location. The basic instruction code synchronization of processors is, thus, not perturbed while the I/O or interrupts get handled.

References

[1] A. Agarwal, M. Horowitz, and J. Hennessy, An analytical cache model, *ACM Transaction on Computer Systems*, May 1989.

[2] R.E. Ahmed, R.C. Frazier and Peter N. Marinos. Cache-aided rollback error recovery (CARER) algorithms for shared-memory multiprocessor systems. technique. *FTCS-20*, pages 82–88, 1990.

[3] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transaction on Computer Systems*, Vol. 4(No. 4), November 1986.

[4] Brian K. Breay and M. J. Flynn. Write-caches as an alternative to write-buffers. Technical report, Stanford University, April 1991.

[5] P. Denning, The working set model for program behavior, *Communication of the ACM*, vol. 11, pp. 323–333, May 1968.

[6] E.N. Elnozahy, D.B. Johnson and W. Zwanepoel. The performance of consistent checkpointing. *Proc. Reliable Distributed Systems*, 1992.

[7] Douglas B. Hunt and Peter N. Marinos. A general purpose cache-aided rollback error recovery (CARER) technique. *FTCS-17*, pages 170-174, 1987.

[8] B. Janssens and W. K. Fuchs. The performance of cache-based error recovery in multiprocessors. *IEEE Trans. on PDS*, pages 1033–1043, Oct. 1994.

[9] Norman P. Jouppi. Cache write policies and performance. *Proc. ISCA*, 1993.

[10] J. Long, W. K. Fuchs and J. A. Abraham. Forward recovery using checkpointing in parallel systems. *Proc. of ICPP*, pages 272-275, Aug. 1990.

[11] J. Long, W. K. Fuchs and J. A. Abraham. Implementing forward recovery using checkpoints in distributed systems. *IFIP DCAA*, 1991.

[12] A. Mendelson, D. Thiebaut, and D. K. Pradhan, Modeling live and dead lines in cache memory systems. *IEEE Trans. on Computer*, pages 1–23, Jan. 1993.

[13] A. Mendelson, N. Suri, and O. Zimmermann, Roll-forward recovery: the bi-directional cache approach. *IEEE Workshop on PDS*, 1994.

[14] D. K. Pradhan and N. H. Vaidya Roll-forward checkpointing scheme: a novel fault-tolerant architecture. *IEEE Trans. on Computers*, pages 1163–1174, Oct. 1994.

[15] K-L Wu, W.K. Fuchs and J.K. Patel Error recovery in shared memory multiprocessors using private caches. *IEEE Trans. PDS*, pages 231–240, 1990.