Efficient Software-Based Fault Tolerance Approach on Multicore Platforms

Hamid Mushtaq, Zaid Al-Ars, Koen Bertels
Computer Engineering Laboratory
Delft University of Technology
{H.Mushtaq, Z.Al-Ars, K.L.M.Bertels}@tudelft.nl

***Abstract*—This paper describes a low overhead software-based fault tolerance approach for shared memory multicore systems. The scheme is implemented at user-space level and requires almost no changes to the original application. Redundant multithreaded processes are used to detect soft errors and recover from them. Our scheme makes sure that the execution of the redundant processes is identical even in the presence of non-determinism due to shared memory accesses. It provides a very low overhead mechanism to achieve this. Moreover it implements a fast error detection and recovery mechanism. The overhead incurred by our approach ranges from 0% to 18% for selected benchmarks. This is lower than comparable systems published in literature.**

I. Introduction

The abundant computational resources available in multicore systems have made it feasible to implement otherwise prohibitively intensive tasks on consumer grade systems. However, these systems integrate billions of transistors to implement multiple cores on a single die, thus raising reliability concerns, as smaller transistors are more susceptible to both transient [12] as well as permanent [13] faults.

A common approach for providing fault tolerance is to perform redundant execution of the software. This is done by using the state machine replication approach [14]. In this approach the replicated copies of a process (known as replicas) follow the same execution sequence and produce the same output if given the same input. This requirement necessitates that the replicas handle non-deterministic events such as asynchronous signals and non-deterministic functions (such as *gettimeofday*) deterministically. This is usually done by having one replica log the non-deterministic events and have the other replicas replay them at the same point in program execution. In a shared memory multithreaded program, this also means that the original and replica processes perform non-deterministic shared memory accesses deterministically, so that they do not diverge in the absence of faults.

Different software-based solutions have been proposed, for deterministic execution of shared memory multithreaded programs on multicore processors, such as DTHREADS [9] and CoreDet [1], which are too slow to be used for practical purposes. On the other hand, Kendo [7], while an efficient solution, suffers from portability problem as it requires the use of deterministic hardware performance counters, which are not available on many platforms [10]. Respec [5] is a record/replay approach for fault tolerance, that requires kernel modification and also does not have highly efficient method of memory comparison for error detection. Moreover, it does not perform deterministic execution very efficiently for benchmarks with high lock frequencies.

978-3-9815370-0-0/DATE13/©2013 EDAA

In this paper, we describe a software based efficient fault tolerance scheme that performs the following.

1) The scheme is implemented using a user-level library and does not require a modified kernel.
2) Record and Replay of synchronization operations is made efficient and scalable by eliminating atomic operations and true and false sharing of cache lines.
3) The error detection mechanism is optimized to perform memory comparisons of the replicas efficiently in user-space.

In Section II we discuss the background and related work, while in Section III, we discuss the implementation. In Section IV, we evaluate the performance of our scheme. We finally conclude the paper with Section V.

II. Background and related work

For error detection of software running on a single core, fault tolerant systems commonly employ redundant execution at different levels of abstraction, at instruction, process or virtual machine level [15]. Schemes which work at instruction level have low error detection latencies, especially those which operate at hardware level. On the other hand, schemes which work at process and virtual machine level allow error to propagate before detecting it. Another important issue in process level and virtual machine level systems is that they need to cater for non-deterministic events, such as interrupts and non-deterministic functions, such as time of the day. They need to make sure that execution of the replicas is deterministic with respect to each other. Such schemes usually use the concept of primary and backup replicas, where the primary is responsible for logging information about the non-deterministic events to be used by the backups. For this purpose, non-deterministic events such asynchronous signals have to be executed at the same point in the code by the replicas. As an example, [16] defers asynchronous signal handling to known points in the code, such as function calls, system calls or backward branches.

In multithreaded programs running on multicore processors, there is one more source of non-determinism, which is shared memory accesses. These accesses are much more frequent than interrupts or signals. Therefore, efficient deterministic execution of replicas in such systems is much more difficult to achieve.

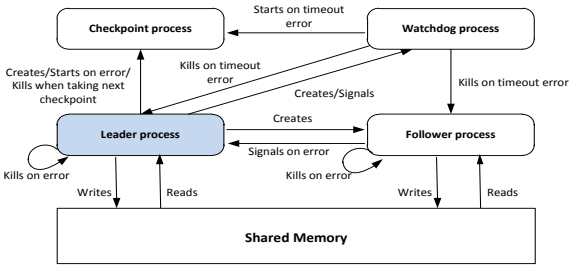

Fig. 1. Data flow diagram of our fault tolerance scheme

Lately, effort has been done to create deterministic languages, that ensure deterministic execution of a program. Examples of programming languages designed for deterministic parallel execution are StreamIt [8] and SHIM [3]. However porting programs written in traditional languages to deterministic languages is difficult as the learning curve is high for programmers used to programming in traditional languages. Therefore, deterministic execution at runtime is still the only viable solution to most users.

One such method for runtime deterministic execution is CoreDet [1] that uses bulk synchronous quantas along with store buffers and relaxed memory model to achieve determinism. Since this method requires bulk syncrhonous quantas, it has a very high overhead (1-11x for 8 cores) and limited scalability.

Kendo [7] is a software approach that works only on programs without data races, that is, those that access shared memory only through synchronization objects. It executes threads deterministically and performs load balancing by only allowing a thread to complete a synchronization operation when its logical clock, which is used to perform deterministic execution, becomes less than those of the other threads. Since this method requires global communication among threads for reading clock values, it also has limited scalability.

Respec [5] is a record/replay software approach that only logs synchronization objects rather than every shared memory access. If divergence is found between the replicas, it rollbacks and re-execute from a previous checkpoint. However, if divergence is found again on re-execution, a race condition is assumed. At that point, a stricter deterministic execution is performed. It uses producer-consumer queues. A queue is shared between a thread in the leader and its corresponding thread in the follower and is used to record logical clocks for mutexes. Each recorded operation atomically increments a clock. Since having a producer-consumer queue for each mutex will require a large memory, *Respec* only uses fixed number of clocks, that is, 512. The hash of the address of a mutex is used to point to its logical clock. A thread in the follower process only acquires a mutex when its logical clock matches that recorded by the corresponding thread of the leader.

III. Fault tolerance scheme

Our fault tolerant scheme is intended to reduce probability of failures in the presence of transient faults. The data flow diagram of our fault tolerance scheme is shown in Figure 1. Initially, the leader process (which is the original process highlighted in the figure) creates the watchdog and follower processes. The follower process is identical to the leader process and follows the same execution path. The execution is divided into time slices known as epochs. An epoch starts and ends at a program barrier. At the end of each epoch, the memories of the leader and follower processes are compared by the follower. If no divergence is found, a checkpoint is taken and output to files or screen is committed. The previous checkpoint is also deleted. The checkpoint is basically a suspended process which is identical to the leader process at the time the checkpoint is taken. If a divergence is found at the end of an epoch, the follower process signals an error to the leader process which in turn signals the checkpoint process to start and kills itself and its follower. This can also happen inside an epoch, if the follower sees that the parameters (of synchronization functions or system calls) logged by the leader do not match those read by the follower. When the checkpoint process starts, it becomes the leader and creates its own follower. It might also happen that the leader or follower processes are unable to reach the end of an epoch, due to some error which hangs them. In that case, the watchdog process detects those hangs by using timeouts and signals the checkpoint process to start. The watchdog process itself is less vulnerable to transient faults as it remains idle most of the time.

At this moment, our fault tolerant scheme does not work with programs that use inter process communication (such as through pipes and shared memory). The only form of I/O allowed is disk I/O and screen output. Moreover, our scheme assumes that there are no data races in the program. Lastly, we have not added functionality to handle asynchronous signals. However, this functionality can be added for user space by handling asynchronous signals at synchronous points, such as system calls, as done by Scribe [17].

In Section III-A, we discuss how we allow deterministic execution of the replicas. This is followed by Section III-B which discusses error detection. Finally in Section III-C, we discuss our recovery mechanism.

A. Deterministic execution

For deterministic execution, we need to ensure that replicas use the same memory addresses. We also need to ensure determinism in the presence of non-deterministic functions and shared memory accesses. Moreover, we need to make sure that the leader and follower processes use the same memory addresses. For this we need to have a deterministic memory allocation scheme. Finally we also need to make sure that we have deterministic I/O. Below we discuss how we handle these issues.

1) Replica creation: Our library creates a follower from the leader process by using *fork* system call, at the beginning and also when a rollback is done. This is because at a rollback, the checkpoint process becomes the leader and creates its own follower, which uses the same memory addresses as the leader process. We use our own version of *pthread_create* function to make sure that the leader and follower processes use the same stack addresses for the threads. For this purpose, the leader process logs these addresses to be consumed by the follower. For thread identification, we use a thread local variable, so that we can relate a thread in the follower process with that in the leader process.

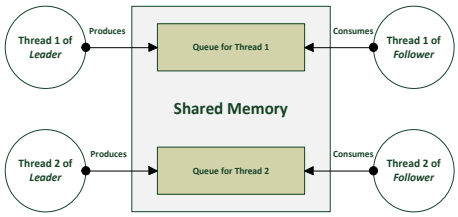

Fig. 2. Communication between the leader and follower processes for deterministic execution

2) Memory allocation: We implement our own memory allocation functions to allocate memory deterministically. In an operating system with Address Space Layout Randomization (ASLR), malloc can be non-deterministic. This is because *malloc* internally uses *mmap* for allocating memory blocks of large sizes and *mmap* can be non-deterministic. Therefore, whenever the memory allocator uses *mmap*, we make sure the follower has the same address returned for *mmap* by calling *mmap* with MAP_FIXED flag and the address returned by the leader process.We also make sure that threads of the leader and follower processes call the *malloc* function in the same order by internally using a mutex, which is locked and unlocked deterministically.

The variables used by our library (not related to original program execution) to perform deterministic execution, may have different values for the leader and follower processes, for example, the flag used to distinguish the leader process from the follower process. For these variables, we use a separate memory, which is allocated with *mmap*. This memory is not compared for error detection.

3) Deterministic shared memory accesses: For redundant deterministic execution, it is necessary that the leader and follower processes perform shared memory accesses in the same order. For this purpose, a mutex is enclosed in a special data structure, which also contains a pointer to clocks for that mutex to aid in deterministic execution. Whenever a thread in the leader process acquires a mutex, it increments the mutex's clock. A thread in the follower only acquires the same mutex in its execution, when its clock matches that for the corresponding thread in the leader.

We create our own deterministic versions of pthread's synchronization functions, such as *pthread_mutex_lock*, *pthread_mutex_unlock*, *pthread_trylock*, *pthread_cond_wait*, *pthread_broadcast*, *pthread_barrier_wait* etc. Since *pthread_mutex_lock* is the mostly used and is also used in our implementation of other pthread synchronization functions, we discuss our *pthread_mutex_lock* algorithm here, which is shown in Algorithm 1. We also have our own versions of data structures for representing the synchronization objects, for example, *pthread_mutex_log_t* instead of *pthread_mutex_t*. Here *m* represents an object of *pthread_mutex_log_t* structure which holds a mutex and its clocks. There is one such object for each mutex in the program. Therefore, deterministic access to a mutex is independent of other mutexes in the program, hence improving scalability.

When a leader thread acquires a mutex, it increments the leader's clock for that mutex and also records that value in a circular queue, so that the follower can acquire the thread when its clock reaches one less than the same value. The

Algorithm 1 Pseudocode for deterministic lock

```
function R_PTHREAD_MUTEX_LOCK(ref pthread_mutex_log_t m)
    q = GetQueue(tid)                                   ▷ There is a separate queue for each thread
    if isLeader then
        r = lock(m.mutex)
        if r == 0 then                                  ▷ Only if lock call is successful, increment the clock
            m.clock = m.clock + 1                       ▷ m.clock does not need to be atomic
        end if
        while !pushq(q, MUTEXLOCK, m.mutex, m.clock, r)
        end while
        return r
    else                                                                          ▷ Follower
        while not !popq(q, ref type, ref mutex, ref clock, ref r)
        end while
        if type != MUTEXLOCK and mutex != m.mutex then  ▷ Logged parameters do not match
            SignalErrorAndExit()
        end if
        if r != 0 then
            return r
        end if
        while (m.clock+1) != clock
        end while
        lock(m.mutex)
        m.clock = m.clock + 1
        return 0
    end if
end function

function PUSHQ(q, type, addr, clock, r)                                   ▷ Called by Leader
    lindex = GetLeaderQIndex(tid)
    if checkQElementsForZero(lindex) then
        q[lindex].type = type
        q[lindex].addr = addr
        q[lindex].clock = clock
        q[lindex].r = r + 1
        SetLeaderQIndex((lindex + 1) %QCAPACITY)
        return TRUE
    else
        return FALSE
    end if
end function

function POPQ(q, ref type, ref addr, ref clock, ref r)                    ▷ Called by Follower
    findex = GetFollowerQIndex(tid)
    if checkQElementsForNonZero(qindex) then
        type = q[findex].type
        addr = q[findex].addr
        clock = q[findex].clock
        r = q[findex].r - 1
        setQElementsToZero(findex)
        SetFollowerQIndex((findex + 1) %QCAPACITY)
        return TRUE
    else
        return FALSE
    end if
end function
```

communication between the leader and follower processes is shown in Figure 2. After acquiring the mutex, the follower also increments its clock for that mutex. Unlike Respec which uses a hash table of 512 entries to keep clocks for all the syncrhonization objects, we use a separate clock for each mutex. The benefit of this is that we can avoid using atomic variables for accessing the clocks, as clock can be incremented after acquiring the lock.

We also optimize the queue access by avoid using atomic variables and avoiding true and false sharing of cache lines. For that purpose, we use a lockless queue as shown by *pushq* and *popq* functions in Algorithm 1. This is unlike *Respec* which uses atomic operations if necessary to access the queue. The typical method of using a lockless queue (which we call *naive* in this paper) is to use shared tail and head indexes. Since in this method, producer and consumer read the head or tail indexes at the same time when the other is writing to it, this causes cache trashing. Hence it is a *true sharing* problem. We avoid this by having local indexes for producer (leader) and consumer (follower). The check for emptiness and fullness is done by checking the data value instead. Producer only writes to the queue when all the fields of the queue element it is about to write to, are zero, while the consumer only reads

when all the fields of the queue element are non-zero. Here, since the value of *r*, which represents the result returned by a synchronization function can be zero, We add one to its value while pushing and subtract one from it when popping. We make sure that the indexes for leader and follower do not share the same cache line by having sufficient padding between them. This makes sure that we do not have the problem of *false sharing*.

4) System Calls, Non-deterministic functions and I/O: We use LD_PRELOAD to preload the system call wrappers found in *glibc* with our own version which perform logging. This is possible, because most of the system calls can be and are usually called through their user-space wrapper functions. This method will not work however, if for example, a system call is made without using the wrapper function, for example, by using inline assembly. So, with our library, the programmer needs to make sure to not make a system call directly. Since the glibc library sometimes also make system calls directly, for example, by making the clone system call in *pthread_create* function, we provide our version of *pthread_create*. We also provide our own version of non-deterministic functions such as *gettimeofday* and *rand* and preload them using LD_PRELOAD. The leader performs logging of the parameters and output of these non-deterministic functions and system calls. The logged parameters are used by the follower to check for errors (by checking for discrepancies), whereas the logged output is just read by the follower. Furthermore, each non-deterministic function and system call is protected by a deterministic lock so that the leader and follower processes perform these calls in the same order.

For I/O, our library allows deterministic I/O for sequential file access and screen write. Write to a file or screen is only performed after making sure that no error occurred during an epoch. For that purpose, no output is committed during an epoch. Instead it is buffered. Our library overrides the *write* and *read* system call wrappers to allow buffering of the data. The buffers are committed at the end of an epoch after comparing the buffer contents of the leader and follower by using hash-sums. For this purpose, each file opened for writing is allocated a special buffer. It is important that addresses of these buffers are the same for the leader and follower process. For this purpose, we use a deterministic memory allocation scheme like the one described in Section III-A2. For sequential file reading, the file offset value is saved at the end of each epoch, so that the file can be rewinded to the previous value in case of rollback.

B. Error detection

At regular intervals of 1 second, known as epochs, dirtied memories of the leader and follower processes are compared. However, the epoch time is reduced to 100 ms if a file or screen output occurs during the epoch. Instead of comparing each memory one by one, the leader and follower processes calculate hash-sums of the dirtied (modified) memory pages, which are then compared. If a discrepancy is found, a fault is detected. The hash-sums are calculated much faster by using the CRC32 instruction of the SSE4.2 instruction set found on modern x86 processors.

The comparison is made even faster by assigning each thread to calculate hash-sum of different portions of the memory. Follower keeps its hash-sums in shared memory so that the leader can read it from there for comparison. We perform memory comparison at barriers which are already found in the program rather than stopping and creating a barrier. This improves the performance, as threads already wait for each other at barriers. If insufficient barriers are found in the program, the programmer can insert calls to function *potential_barrier_wait*, which is provided by our library. This function creates a barrier only when required, that is at the end of an epoch.

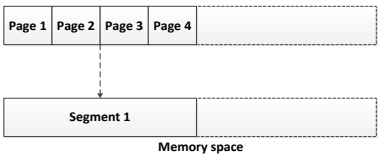

Fig. 3. Memory pages can be grouped into segments to reduce the overhead of memory comparison for error detection

Since our scheme runs at the user-space level, we cannot note down dirtied pages while handling page faults (from the kernel), the way *Respec* does, which is the most efficient method possible. We take special steps to improve its performance at user-space level.

At start of each epoch, we give only read access to allocated memory pages. Whenever a page is accessed for writing, the OS sends a signal to the accessing thread. In the signal handler, the address of the memory page is noted down and both read and write accesses are given to that memory page. In this way, we only need to compare the dirtied memory pages at the end of an epoch. Sending signals on each memory page access violation can slow down execution. Therefore, to reduce the number of such signals, we exploit the concept of spatial locality of data and segmented memory into multiple pages, as shown in Figure 3. A write on any part of a read protected segment of N pages is handled by giving write access to all the N pages in that segment. This improves the execution considerably, as discussed in Section IV, where we discuss the performance evaluation.

Some functions, like that for comparing memories, change the stacks differently for the leader and follower threads. For those purposes, we switch to a temporary stack, so that the original stack remains unaltered from such functions.

The watchdog process is used to detect hangs and recover from them. At the end of each epoch, the leader process sends a signal to the watchdog process to signal that it is not hung. In that signal, the process ID of the checkpoint is also sent, so that the watchdog is able to start the checkpoint process in case it detects hang of leader or follower process. Hangs are detected by using timeout. Besides sending the process ID of the checkpoint, the leader also sends process ID of itself and the follower process when it forks the follower, so that the watchdog process can kill the leader and follower processes before starting the checkpoint process.

C. Recovery

As discussed previously, for fault recovery, we use checkpoint/rollback. whenever the leader takes a checkpoint, it kills the previous checkpoint. If the leader process detects an error, or the Watchdog process detects a hang, a signal is sent to the last checkpoint process, so that the checkpoint process

can start execution. The leader and its follower are killed at that point. The checkpoint process then assumes the role of the leader and forks its own follower. It also creates a new checkpoint. Moreover, it resets the mutex clocks (which exist in shared memory), since they could have been corrupted by an error. Checkpoints are taken only at barrier points. For creating a multithreaded follower, we have implemented a special *multithreaded fork* function that replicates the leader process to create the follower.

IV. Performance Evaluation

We selected 8 benchmarks, two from the PARSEC [2] and six from the SPLASH-2 [11] benchmark sets. We ran all our benchmarks on an 8 core (dual socket with 2 quad cores), 2.67 GHz Intel Xeon processor with 32GB of RAM. All programs were compiled using gcc 4.4.4 with optimization level -O3. The results are shown in Table I.

For each benchmark, we show the results when the benchmark runs for 2 and 4 threads. Number of epochs executed are shown in the third column, while number of synchronization operations performed by each process is shown in the fourth one. This is followed by the number of barriers and total number of memory pages compared for error detection. Then the table shows the redundant execution time, which is the time to execute two instances of the same application. For redundant execution, each instance is executed on one of the two different quad core processors of the dual socket system. Next we show the time for deterministic execution scheme, which is execution without performing error detection and checkpointing but only deterministic locking and unlocking of the mutexes. This is followed by the overall execution (with error detection, checkpointing and Watchdog process). Next we show the overheads of the deterministic and overall execution with respect to the redundant time. For overall execution, the results are shown with memory grouping size of 4.

A. Results

Figure 4 shows the improvement that we get by avoiding atomic variables and having an optimized queue. The left bars are obtained by running the benchmarks with 2 threads while right bars are obtained with 4 threads. The lower portion of the bar shows the overhead with our lockless queue and our method for keeping the clocks for mutexes, while the middle portion shows the additional overhead that we get when we use Respec's method of using a Hash Table for mutex clocks. The upper most portion shows the additional overhead by using naive lockless queue.

We can see that for *fluidanimate*, which has a high lock frequency, we have a significant improvement in performance of deterministic execution. Furthermore, our method of using separate clocks for each mutex is more scalable than Respec's method of using limited clocks and accessing them through a Hash table, that requires atomic operations. The scalability here can be assessed by the fact that for two threads, our scheme and Respec's scheme perform similarly, while for four threads, our scheme performs far better. From this result, we can predict that our scheme will have even better results compared to Respec for larger number of cores. Furthermore, our lockless queue also shows much better scalability than a

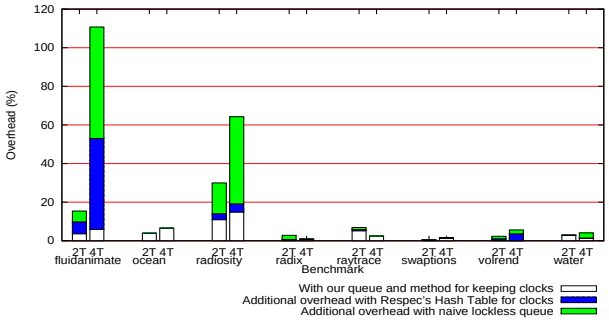

Fig. 4. Deterministic execution overhead

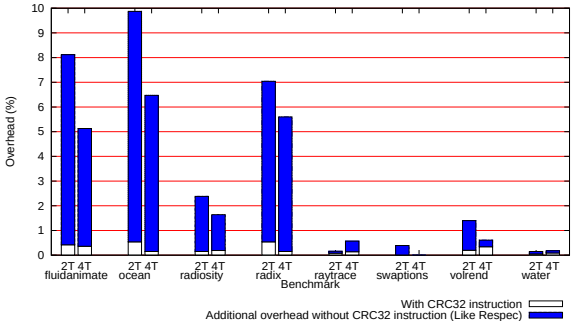

Fig. 5. Memory comparison overhead with and without CRC32 instruction

naive lockless queue due to avoiding true and false sharing of cache lines. Note that we do not get as much improvement for *radiosity*, which also has a high lock frequency, because it uses much fewer mutexes than *fluidanimate*, and hence fewer clocks, causing more contention in communication between the leader and follower threads.

Figure 5 shows the improvement that we get from using the CRC32 instruction (as opposed to Respec which does not use that instruction) for calculating hash sums for error detection. The results are especially impressive for benchmarks which modify higher number of pages, such as *fluidanimate*, *ocean* and *radix*.

In Figure 6, we show the overall results in the form of bar graphs, with each factor shown separately. Due to the optimizations we discussed and reduction in epoch overhead, which will be discussed in the next section, the overhead never exceeds 18% for four threads and is negligible for benchmarks with small memory usage and low lock frequencies.

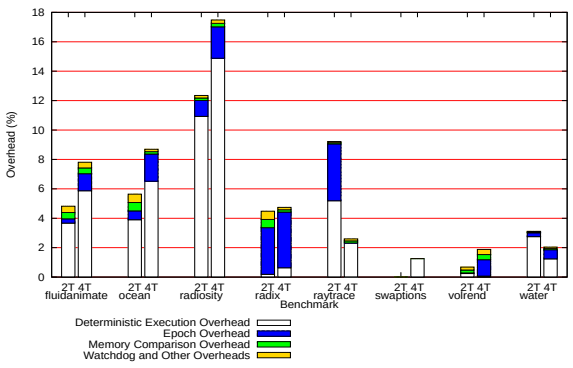

Fig. 6. Overall overhead

TABLE I. PERFORMANCE RESULTS OF OUR SCHEME FOR THE SELECTED BENCHMARKS

Benchmarks	Threads	Epochs	Synch Ops	Barriers	Pages modified	Redundant exec time (ms)	Deterministic exec time (ms)	Overall time (ms)	Det exec overhead w.r.t Redt exec time(%)	Overall overhead w.r.t Redt exec time (%)
fluidanimate	2	3	8689200	161	25238	2073	2149	2168	4%	5%
	4	2	16909680	161	25136	1294	1370	1392	6%	8%
ocean	2	3	10092	10763	103398	2819	2929	3008	4%	7%
	4	2	20184	10763	74518	1840	1960	2029	7%	10%
radiosity	2	2	8981938	19	10378	1199	1330	1347	11%	12%
	4	2	8900495	19	10344	887	1019	1043	15%	18%
radix	2	2	18	12	57706	1072	1074	1138	0%	6%
	4	1	54	12	35170	624	628	666	1%	7%
raytrace	2	2	243914	2	200	1214	1277	1325	5%	9%
	4	1	243918	2	148	690	706	702	2%	2%
swaptions	2	1	77020	1	6	510	510	510	0%	0%
	4	1	77020	1	8	239	242	242	1%	1%
volrend	2	3	459760	241	160	2490	2496	2503	0%	1%
	4	2	463922	241	154	1443	1444	1462	0%	1%
water	2	2	125047	664	474	1889	1941	1948	3%	3%
	4	2	188142	664	550	1125	1139	1148	1%	2%

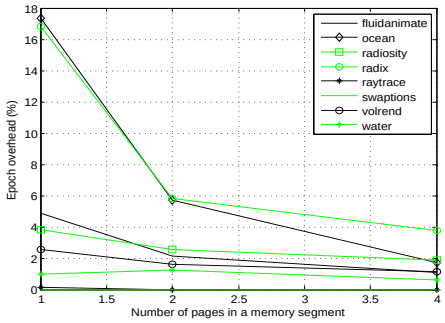

Fig. 7. Reduction in overhead by grouping memory into segments

B. Impact of Grouping Memory Pages

Figure 7 shows the impact of grouping memory pages (see Section III-B) on the performance. The overhead shown is the *epoch* overhead which mainly consists of the overhead of signals for noting down dirtied memory pages. We can see that for applications like *ocean* and *radix* which have high memory usage, we get significant performance gains using page grouping. However, it has to be noted that there is a limit to the number of pages which can be grouped for optimal performance, as grouping too many pages will cause the application to compare more pages which have not been actually modified by that application, thus creating unnecessary overhead.

V. CONCLUSION

In this paper, we described the design and implementation of a user-space level leader/follower based fault tolerance scheme for multithreaded applications running on multicore processors. We applied several optimizations to speedup the execution, like avoiding atomic variables, true and false sharing of cache lines for recording/replaying synchronization operations, reducing signals sent by the OS on page faults (used to note down dirtied pages) and using the CRC32 instruction from SSE4.2 instruction set to greatly improve error checking performance. Empirical measurements on tested benchmarks show that the overhead does not exceeds 18% for four threads. We compared our results with Respec and showed that our scheme is more efficient in performing deterministic execution and comparing memories for error detection. Moreover, we showed that by grouping memory pages, we considerably reduced the overhead of signals used for noting modified memory pages.

ACKNOWLEDGMENT

This research has been funded by the projects Smecy 100230, iFEST 100203 and REFLECT 248976.

REFERENCES

[1] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. *SIGARCH Comput. Archit. News*, 38:53–64, March 2010.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. PACT '08, pages 72–81.

[3] S. A. Edwards and O. Tardieu. Shim: a deterministic model for heterogeneous embedded systems. EMSOFT '05, pages 264–272.

[4] D. Hower, P. Dudnik, M. Hill, and D. Wood. Calvin: Deterministic or not? free will to choose. HPCA '11, pages 333 –334, feb.

[5] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient online multiprocessor replay via speculation and external determinism, ASPLOS '10, pages 77–90.

[6] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. ISCA '08, pages 289–300.

[7] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, 44:97–108, March 2009.

[8] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196.

[9] E. D. B. Tongping Liu, Charlie Curtsinger. Dthreads: Efficient deterministic multithreading. SOSP '11, pages 327–336.

[10] V. Weaver and S. McKee. Can hardware performance counters be trusted? In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 141 –150.

[11] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. *SIGARCH Comput. Archit. News*, 23:24–36, May 1995.

[12] R. Baumann. Soft errors in advanced semiconductor devices-part i: the three radiation sources. *Device and Materials Reliability, IEEE Transactions on*, 1(1):17 –22, mar 2001.

[13] S. Nomura, M. D. Sinclair, C.-H. Ho, V. Govindaraju, M. de Kruijf, and K. Sankaralingam. Sampling + dmr: practical and low-overhead permanent fault detection. ISCA '11, pages 201–212.

[14] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22:299–319, December 1990.

[15] H. Mushtaq, Z. Al-Ars, and K. Bertels. Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems. IDT 2011, pages 12–17.

[16] A. Shye, J. Blomstedt, T. Moseley, V. Reddi, and D. Connors. Plr: A software approach to transient fault tolerance for multicore architectures. *Dependable and Secure Computing, IEEE Transactions on*, 6(2):135 –148, 2009.

[17] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. *SIGMETRICS Perform. Eval. Rev.*, 38(1):155–166, June 2010.