A User-level Library for Fault Tolerance on Shared Memory Multicore Systems

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

***Abstract*—The ever decreasing transistor size has made it possible to integrate multiple cores on a single die. On the downside, this has introduced reliability concerns as smaller transistors are more prone to both transient and permanent faults. However, the abundant extra processing resources of a multicore system can be exploited to provide fault tolerance by using redundant execution. We have designed a library for multicore processing, that can make a multithreaded user-level application fault tolerant by simple modifications to the code. It uses the abundant cores found in the system to perform redundant execution for error detection. Besides that, it also allows recovery through checkpoint/rollback. Our library is portable since it does not depend on any special hardware. Furthermore, the overhead (up to 46% for 4 threads), our library adds to the original application, is less than other existing approaches, such as *Respec*.**

I. Introduction

Although the shrinking transistor size has made it possible to implement multiple cores on a single die, it has also made reliability a concern, as smaller transistors are more prone to both transient [1] as well as permanent [2] faults. However, the abundant processing resources of a multicore system can be exploited to provide fault tolerance through redundant execution.

One way to use the abundant processing resources to provide fault tolerance is by using the state machine replication approach [3]. For multithreaded programs running on shared memory multicore systems, it is required that threads of the replicas access shared memory in the same order. In other words, shared memory accesses should be deterministic. Our library ensures this. Overall it provides the following features.

- Efficient deterministic execution in presence of lock-based shared memory accesses.
- Optimized memory comparison of the replicas for error detection.
- Checkpoint/rollback to perform recovery from transient errors.

In Section II we discuss the background and related work. In Section III, we discuss the overview of our library, while in Section IV, we discuss its implementation. In Section V, we present and discuss our results. We finally conclude the paper with Section VI.

II. Background and related work

Fault tolerance is achieved by three major steps, error detection, isolation and recovery [4]. With redundant execution, an error is detected if the replicas diverge. Since any kind of divergence is used to detect an error, it is important to remove any source of divergence which is not due to an error. The only sources of non-determinism in a single threaded program are non-deterministic functions, such as *gettimeofday* and asynchronous signals, while for a multithreaded program, there is also non-determinism due to shared memory accesses. Moreover, shared memory accesses are usually much more frequent as compared to non-deterministic functions and asynchronous signals, which makes implementing efficient state machine based replication more difficult for a multithreaded program, running on a shared memory multicore system, than for a single threaded application.

Two main approaches, record/replay and deterministic multithreading, exist for this purpose. In record/replay, the order of shared memory accesses on the original processes are recorded so that they can be replayed by the other replicas. On the other hand, with deterministic multithreading, given the same input, a process always performs the same ordering of shared memory accesses. It has to be noted though that for non-deterministic functions, such as *gettimeofday* and asynchronous signals, record/replay is the only viable method. Our library uses the record/replay approach.

For record/replay, both hardware and software-based methods exist. Karma [5] and DeLorean [6] are examples of hardware-based approaches. While Karma intercepts the cache coherence protocols to record inter-processor data dependencies and later use these recorded data dependencies to replay, DeLorean uses a relaxed memory model, where each processor executes instructions in chunks concurrently and an arbiter is used to commit the chunks. Replaying is done by replaying the chunks in the order in which they were committed. Respec [7] is a software-based method. It logs the ordering of acquisition and release of synchronization objects, such as mutexes, to make replicas acquire the synchronization objects in the same order. It also performs checkpoint/rollback to perform recovery.

For deterministic multithreading, also, both hardware and software-based methods exist. An example of hardware-based approach is Calvin [8]. It executes a program deterministically by executing instructions in the form of chunks and committing them at barriers points deterministically. Kendo [9] is a software based approach that only deals with programs

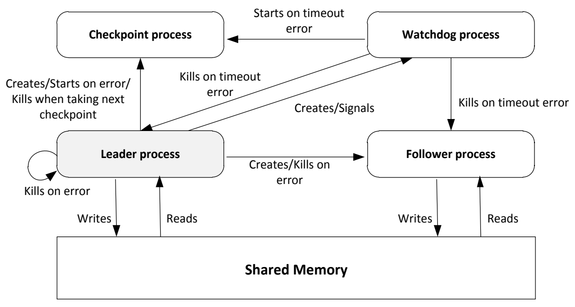

Fig. 1. Data flow diagram of our fault tolerance scheme

without data races. For efficient deterministic execution, it performs load balancing by only allowing a thread to acquire a synchronization object if that thread has executed less instructions than the other threads. The number of instructions are calculated by counting the retired stores.

III. Overview of the library

Our library 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 creates its replica (follower process) and the watchdog. The execution is divided into time slices (epochs). At the end of each epoch, the memories of the leader and follower processes are compared by the leader. 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 leader process signals the checkpoint process to start and kills itself and its follower. When the checkpoint process starts, it becomes the leader and creates its own follower. It might also happen that the leader or follower process 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.

IV. Implementation

This section discusses the implementation of our library. In Section IV-A, we discuss how the follower process is created. In Section IV-B, we discuss our memory allocation technique, which is followed by Section IV-C, where we explain how we are able to deterministically access the shared memory through mutexes. Section IV-D discusses our error detection mechanism, while Section IV-E discusses recovery. Lastly, in Section IV-F, we discuss how our library handles I/O and non-deterministic functions such as *gettimeofday*. Note that currently our library does not support deterministic replaying of asynchronous signals, which is left as future work.

A. Follower creation

Our library assumes that threads in the application are created once at the start of the application. Therefore, we create the follower process at point in the code where the threads are created. For this purpose, we replace the *pthread_create* function with *r_pthread_create*, which internally calls *pthread_create* function and indirectly calls the Linux's *fork* function.

To make sure that the follower threads have the same stack contents as the leader, our library itself allocates the memory used for thread stacks. These allocated stacks are then passed as attributes to the *pthread_create* function (called from *r_pthread_create*). When the leader calls the *fork* function, although all threads, beside the one calling it, die in the forked process, the stack contents are still present in the memory. Therefore, to recreate a thread with the same stack in the follower, we call *pthread_create* with the same stack attribute. 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.

For making sure that a follower thread also has the same register values as the corresponding leader thread, the thread's start routine passed as an argument to the *pthread_create* function (called from *r_pthread_create*), is not the start routine itself but wrapper *thread_start* that then calls the start routine, which is provided as a parameter to it. For the leader process, this function calls our barrier function *r_pthread_barrier_wait* in the beginning. When *r_pthread_barrier_wait* is called for the first time, each thread of the leader saves its registers, including the instruction pointer, by using the C *setjmp* function. The follower is itself created by the main thread by calling *fork* function, from within the *r_pthread_barrier_wait* function. The newly forked process then recreates the threads and each thread jumps to the same location as the thread in the leader process by using the C *longjmp* function. This is done by having the forked process also pass the *thread_start* function as start routine to *pthread_create* function. But unlike the leader process, the follower threads call *longjmp* at the beginning of the *thread_start* function.

Note that since the main thread is replicated by the *fork* process, we do not need to recreate it. However, *r_pthread_barrier_wait* needs to be inserted just after the code where the threads are created, so that all the threads are at a barrier point when the follower is forked.

B. Memory allocation

In an operating system with Address Sapce 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.

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

```
function R_PTHREAD_MUTEX_LOCK(ref pthread_mutex_log_t m)
    if $isLeader$ then
        $lock(m.mutex)$
        $m.leaderClock = m.leaderClock + 1$
        while $m.threadClock[tid] > 0$
        end while
        $m.threadClock[tid] = m.leaderClock$
    else
        while not $(m.threadClock[tid] == (m.followerClock + 1))$
        end while
        $m.threadClock[tid] = 0$
        $lock(m.mutex)$
    end if
end function

function R_PTHREAD_MUTEX_UNLOCK(ref pthread_mutex_log_t m)
    if not $isLeader$ then
        $m.followerClock = m.followerClock + 1$
    end if
    $unlock(m.mutex)$
end function
```

Fig. 2. Pseudocode for deterministic lock and unlock

memory, which is allocated with *mmap*. This memory is not compared for error detection.

C. 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. Since we assume that a program has no data races and all synchronization operations are done using mutexes, we provide functions *r_pthread_mutex_lock* and *r_pthread_mutex_unlock* for deterministically locking and unlocking a mutex. A mutex is enclosed in a special data structure, known as *pthread_mutex_log_t*, which also contains a pointer to clocks for that mutex to aid in deterministic execution. The memory region to hold the mutex clocks is shared between the leader and follower processes.

Our deterministic locking and unlocking algorithms for mutexes are shown as Figure 2. The benefit of this scheme is that it uses less memory as compared to schemes that use producer/consumer queues, such as Respec [7]. Secondly, by wrapping the pointer to the mutex clocks in the same data structure as the mutex, we avoid the overhead of using a hash table, which is used by Respec. Lastly, our algorithm is written such that it exploits the strict memory consistency model of multicore x86 (memory ordering respects transitive visibility and stores to the same location have a total order) and thus avoid using atomic variables (which incur significant overhead due to use of memory fences) on such systems.

D. Error detection

At regular intervals (epochs) of one second, the leader and follower processes calculate checksums by performing modular sums of the contents of the dirtied (modified) memory pages, which are then compared by the leader. If a discrepancy is found, a fault is detected. Follower keeps its checksum in the 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. If insufficient number of barriers are found in the program, the programmer can insert our library function *r_potential_barrier_wait* in

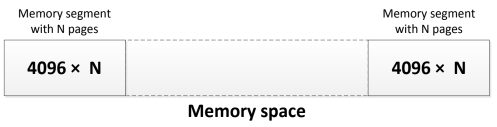

Fig. 3. Memory pages can be grouped into segments to reduce the overhead of memory comparison for error detection

the code. This function will create a barrier (by calling *r_pthread_barrier_wait*) only if the program has reached the end of an epoch.

To note down dirtied pages, at start of each epoch, we give only read access to memory pages, so that a page faults can be trapped to note down dirtied pages. To reduce the number of such page faults however, 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 V, where we discuss the performance evaluation.

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.

E. Recovery

For fault recovery, we use checkpoint/rollback. Checkpointing is done by forking a process and suspending it. If the leader process detects an error or the Watchdog detects a hang, a signal is sent to the checkpoint process to start execution. The leader and follower processes are also killed. The checkpoint process now becomes the new leader and forks its own follower. The checkpoint process also resets the mutex clocks (which exist in shared memory), since they could have been corrupted by an error.

Creation of the checkpoint process is very similar to creation of the follower (see Section IV-A), with the difference being that the checkpoint process is suspended in the beginning. Only when it is signalled to start, it recreates the threads and starts execution.

F. I/O and non-deterministic functions

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. Therefore, our library defines a structure *r_FILE*, which not only contains the FILE pointer, but also the buffer. The *r_fopen* function returns a pointer to this structure. The buffers are then committed at the end of an epoch after comparing the buffer contents of the leader and follower by using checksums. For sequential file reading, we provide functions *r_fread* and *r_fscanf*. The *r_FILE* structure also contains the file offset value at the last epoch, so that the file pointer can be rewinded to the previous value in case of rollback.

For non-deterministic functions such as *gettimeofday*, our library allows the programmer to create a deterministic wrapper

TABLE I
PERFORMANCE RESULTS OF OUR SCHEME FOR THE SELECTED BENCHMARKS

Benchmarks	Threads	Epochs	Locks	Pages compared	Original time (ms)	Deterministic exec time (ms) & Overhead	Overall time (ms) & Ovheread
Fluidanimate	2	3	4313983	33890	1988	2298 (16%)	2379 (20%)
	4	2	7466285	25380	1205	1696 (41%)	1712 (42%)
Ocean	2	4	5046	104504	2432	2459 (1%)	3091 (27%)
	4	3	10092	80618	1890	1895 (0%)	2119 (12%)
Water-nsq	2	3	125047	12624	1837	1859 (1%)	1861 (1%)
	4	2	188142	24912	1090	1112 (2%)	1170 (7%)
Radiosity	2	2	6124778	6920	920	1140 (24%)	1153 (25%)
	4	1	6170016	12616	589	854 (44%)	865 (46%)
Raytrace	2	1	121958	2344	1116	1216 (9%)	1260 (13%)
	4	1	121960	2348	663	700 (6%)	738 (11%)

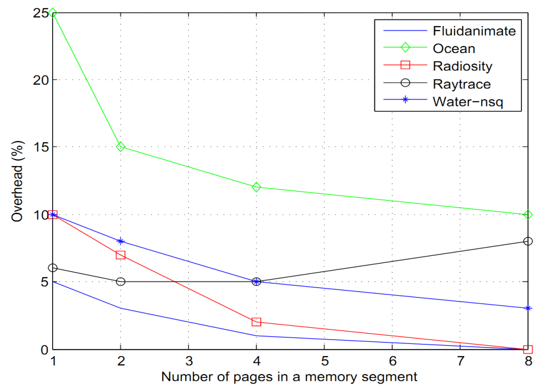

Fig. 4. Reduction in overhead by grouping memory into segments

by using functions *r_log_data* and *r_read_data*. *r_log_data* is called by the leader process to log the outputs of that function, while the follower process reads the outputs by calling the *r_read_data* function.

V. PERFORMANCE EVALUATION

We selected 5 benchmarks, one from the PARSEC [10] and four 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, running CentOS Linux version 5, with kernel 2.6.18. We used gcc 4.4.4 and optimization level -O3 to compile our results. The results are shown in Table I. For each benchmark, we show the results for 2 and 4 threads (For redundant execution, 4 threads means 8 threads in total). We compare the original execution time with deterministic execution time (excluding overhead of error detection, checkpointing and watchdog) and the overall time which includes all the overheads. We used memory segments of size 4 (See Section IV-D for discussion on memory grouping). For *fluidanimate*, which has high lock frequency our library only adds an overhead of 42%, while Respec adds an overhead of 67%. Also for *ocean*, which has high memory consumption, Respec adds an overhead of 43%, while our library adds an overhead of just 12% due to the optimized memory comparison scheme.

Figure 4 shows the impact of grouping memory pages on performance. We show the results for memory segment sizes of 1, 2, 4 and 8. Note that the overhead shown is after subtracting the overhead of deterministic execution. We can see that for an applications like *Ocean* which has high memory usage, we get significant performance gains using page grouping. However, grouping too many pages can also cause the application to compare more pages which have not been actually modified by that application, thus creating unnecessary overhead. This is evident for *Raytrace* which has lower memory usage than other benchmarks. However, for 4 pages, all five benchmark show performance gain.

VI. CONCLUSION

In this paper, we described the design and implementation of a user-level library for fault tolerance of multithreaded user-level applications running on shared memory multicore systems. Our library requires programmer to make little modifications to the program for providing fault tolerance. It allows creation of a multithreaded redundant process for detecting errors and provides facility of checkpointing and rollback for recovery. We also applied several optimizations to speedup the execution, like reducing memory for logging, which is required for record/replay, and optimizing memory comparison for error detection. Empirical measurements on tested benchmarks show that the overhead does not exceeds 46% for four threads.

ACKNOWLEDGMENT

This research is supported by Artemis through the SMECY project (grant 100230).

REFERENCES

[1] R. Baumann, “Soft errors in advanced semiconductor devices-part i: the three radiation sources,” *Device and Materials Reliability, IEEE Transactions on*, vol. 1, no. 1, pp. 17 –22, mar 2001.

[2] 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,” in *Proceeding of the 38th annual international symposium on Computer architecture*, ser. ISCA ’11, 2011, pp. 201–212.

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

[4] H. Mushtaq, Z. Al-Ars, and K. Bertels, “Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems,” in *Design and Test Workshop (IDT), 2011 IEEE 6th International*, dec. 2011, pp. 12 –17.

[5] A. Basu, J. Bobba, and M. D. Hill, “Karma: scalable deterministic record-replay,” in *Proceedings of the international conference on Supercomputing*, ser. ICS ’11, 2011, pp. 359–368.

[6] P. Montesinos, L. Ceze, and J. Torrellas, “Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently,” in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA ’08, 2008, pp. 289–300.

[7] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn, “Respec: efficient online multiprocessor replayvia speculation and external determinism,” in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ser. ASPLOS ’10, 2010, pp. 77–90.

[8] D. Hower, P. Dudnik, M. Hill, and D. Wood, “Calvin: Deterministic or not? free will to choose,” in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, feb. 2011, pp. 333 –334.

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

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The parsec benchmark suite: characterization and architectural implications,” in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT ’08, 2008, pp. 72–81.

[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*, vol. 23, pp. 24–36, May 1995.