Fault Tolerance on Multicore Processors using Deterministic Multithreading

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 software based fault tolerance approach for multithreaded programs running on multicore processors. 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. This is done by making sure that the redundant processes acquire the locks for accessing the shared memory in the same order. Instead of using record/replay technique to do that, our scheme is based on *deterministic multithreading*, meaning that for the same input, a multithreaded program always have the same lock interleaving. Unlike record/replay systems, this eliminates the requirement for communication between the redundant processes. Moreover, our scheme is implemented totally in software, requiring no special hardware, making it very portable. Furthermore, our scheme is totally implemented at user-level, requiring no modification of the kernel. For selected benchmarks, our scheme adds an average overhead of 49% for 4 threads.**

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 [10] as well as permanent [11] 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 [12]. 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 replicas perform non-deterministic shared memory accesses deterministically, so that they do not diverge in the absence of faults.

One way of making sure that the redundant processes access the shared memory in the same order is to perform record/replay where the leader process records the order of locks (to access shared memory) in a queue which is shared between the leader and follower. The follower in turn reads from that queue to have the same lock acquisitions order. This approach is used by Respec [4] and our previous work [6]. However, this requires communication between the leader and follower process, which decreases reliability, as the memory used for communicating might itself become corrupted due a soft error. Moreover, it requires extra memory.

In this scheme, instead of depending on record/replay, we use *deterministic multithreading*, that is, given the same input, a multithreaded process always have the same lock interleaving. This makes sure that the redundant processes acquire the locks in the same order without communicating with each other. We adapt the method used by Kendo [5] to do this, but unlike *Kendo*, our scheme neither requires deterministic hardware performance counters, which are not available on many platforms [8] (including many x86 systems), nor kernel modification for deterministic execution. The logical clocks used for deterministic execution are inserted by the compiler instead.

We can sum up the contributions of this paper as follows.

1) The scheme is implemented using a user-level library and does not require a modified kernel.
2) The scheme uses *deterministic multithreading* instead of record/replay to ensure that the redundant processes acquire locks for shared memory access in the same order. This eliminates the requirement of communication between replicas for deterministic shared memory accesses, making the system more reliable (by increasing isolation). Moreover, it consumes less memory.
3) The scheme is very portable since it does not depend upon any special hardware for determinisitic execution.

In Section II we discuss the background and related work, while in Section III, we discuss our fault tolerance scheme. This is followed by Section IV, where we discuss the implementation. In Section V, we evaluate the performance of our scheme, and we finally conclude the paper with Section VI.

II. Background and related work

A fault tolerant system which uses redundant execution needs to make sure that the redundant processes do not diverge in the absence of faults. In a single threaded program,

in the absence of any fault, the only possible causes of divergence among the replicas can be non-deterministic functions (such as *gettimeofday*) or asynchronous signals/interrupts.

However, 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.

One method to ensure redundant processes access shared memory in the same order is record/replay. Both software and hardware methods exist for that purpose. An example of hardware approach is Karma [14] which intercepts the cache coherence protocols to record inter-processor data dependencies and later use these recorded data dependencies to replay. Respec [4] 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.

The disadvantage of employing record/replay for deterministic shared memory accesses is that it requires communication between the replicas, making the fault tolerant scheme less reliable as the shared memory used for communication can itself become corrupted by one of the replicas. Moreover it requires extra memory.

To eliminate this communication and memory requirement, we can employ *deterministic multithreading*, where a multithreaded process have always the same memory interleaving for the same input. The ideal situation would be to make a multithreaded program deterministic even in the presence of race conditions, that is, provide *strong determinism*. This is not possible to do efficiently with software alone though. One can use a relaxed memory model where every thread writes to its own private memory, while data to shared memory is committed only at intervals. However, stopping threads regularly for committing to shared memory degrades performance as demonstrated by CoreDet [1], which has a maximum overhead of 11x for 8 cores. We can reduce the amount of committing to the shared memory by only committing at synchronization points such as locks, barriers or thread creation. This approach is taken by DTHREADS [7]. Here one can still imagine the slowdown in case of applications with high lock frequencies. Moreover, since in this case committing to the shared memory is done less frequently, more data has to be committed, thus also making it slow for applications with high memory usage. This is why hardware approaches have been proposed to increase efficiency of deterministic execution. An example of such approach is Calvin [3], which uses the same concept as *CoreDet* for deterministic execution but make use of a special hardware for that purpose.

Since performing deterministic execution in software alone is inefficient, one can relax the requirements to improve efficiency. For example, Kendo [5] does this by only supporting deterministic execution for well written programs that protect every shared memory access through locks. In other words, it supports deterministic execution only for programs without race conditions. The authors of *Kendo* call it *Weak Determinism*. Considering the fact that most well written programs are race free and there exist tools to detect race conditions, such as Valgrind [15], *Weak Determinism* is sufficient for most well written multithreaded programs.

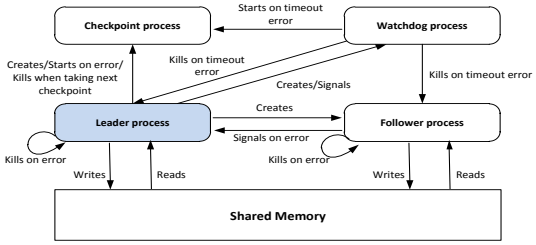

Figure 1. Block diagram of our fault tolerance scheme

The basic idea of *Kendo* is that it uses logical clocks for each thread to determine when a thread will acquire a lock. The thread with the least value of logical clock gets the lock. Though being quite efficient, *Kendo* still suffers from portability problems. First of all, it requires deterministic hardware performance counters for counting logical clocks. Many popular platforms (including many x86 platforms) do not have any hardware performance counters that are deterministic [8]. Secondly, *Kendo* needs modification of the kernel to allow reading from the hardware performance counters.

III. Fault tolerance scheme

Our fault tolerant scheme is intended to reduce probability of failures in the presence of transient faults. The block 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. 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, execution is restarted from the last checkpoint by resuming the checkpoint process and killing the leader and follower processes. This can also happen inside an epoch, if the follower sees that the parameters of system calls logged by the leader do not match those read by the follower, in which case it signals the leader process to restart execution from the last checkpoint. When the checkpoint process starts, it becomes the leader and creates its own follower. The watchdog process is used to detect timeout

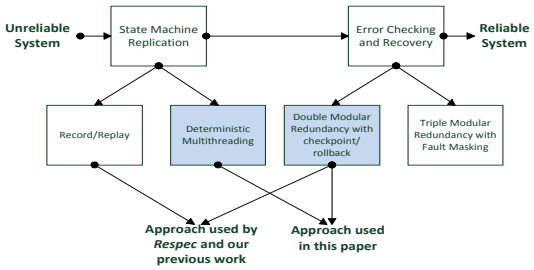

Figure 2. Steps used by our fault tolerance scheme

errors and recover from them. This is done by having the watchdog process signal the checkpoint process to start on a timeout error.

The approach used in this paper is different from *Respec* and our previous work [6] in the way it makes sure that leader and follower processes are identical in the presence of non-deterministic shared memory accesses. While *Respec* and our previous work used the record/replay technique, where the leader logs the synchronization operations (to access shared memory) in a queue which is then read by the follower to have the same order of synchronization operations, in this paper, we use *deterministic multithreading*, which does not require such type of communication, thus improving isolation and fault tolerance. The difference in these two approaches is illustrated in Figure 2. Note that in this approach, we still use shared memory between leader and follower, but only to log results from non-deterministic functions and type, input parameters and results of system calls, besides using it for memory comparison at the end of epochs for error checking.

At this moment, our fault tolerance scheme does not work with programs that use inter process communication (through pipes and shared memory for example). 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 [13].

IV. Implementation

In this section, we discuss the implementation of our fault tolerance scheme. We start by Section IV-A, where we discuss how deterministic execution of the replicas is performed. This is followed by Section IV-B which discusses error detection. Finally in Section IV-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 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 our own to make sure the threads of the replicas use the same memory addresses. More detail on this can be found in [6].

2) Memory allocation: We implement our own memory allocation functions to allocate memory deterministically. Our implementation replaces the locks in the original memory allocation functions with our own deterministic locks. 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 the *mmap* system call. This memory is not compared for error detection.

3) Deterministic shared memory accesses: We use Kendo's algorithm to perform deterministic execution. However, unlike *Kendo* which requires deterministic hardware performance counters, which are not available on many platforms, we insert code to update logical clocks at compile time. This also means that we do not need to modify the kernel which is required by *Kendo* to read from performance counters. Figure 3 shows the point of compilation where our compiler pass executes, which is between the point where the LLVM IR (Intermediate Representation) code is translated to the final binary code by the LLVM backend.

The unit of our logical clock is one instruction. For instructions which take more than one clock cycles, the logical clock is updated according to the approximate number of clock cycles they take.

The Kendo's method of acquiring locks deterministically works by giving lock to the thread with the minimum clock first. So basically our purpose is not only to reduce the code that updates the clocks but also to update the clocks as soon as possible, so that logical clock of the thread waiting for a lock becomes minimum more quickly. In fact, at compile time it is possible to increment the clock even before some instructions are executed, since at compile time we can count the number of instructions. Therefore in all optimizations we apply, besides trying to reduce the clock update overhead, we also try to increment the clock as soon as possible. Without any optimization, we update the clock at start of each of the basic block of LLVM IR. If there is a function call inside that block, we split that block, such that each block either contains no function call or starts and ends with a function call. Then we update the clock at the top of each block if that block contains no function calls, otherwise we update

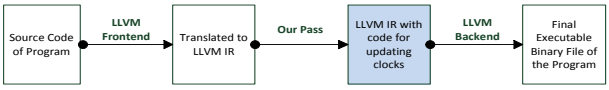

Figure 3. Our tool modifies the LLVM IR code by inserting code for updating logical clocks, used for *deterministic multithreading*

the clocks in between the function calls. By splitting blocks in such a way, we can more easily apply optimizations.

We apply the following optimizations to reduce the logical clock updating overhead as well as reduce waiting time for threads waiting for a lock, by incrementing clocks ahead of time.

Optimization 1 (Function Clocking) As discussed previously, the sooner the clocks are updated, the better, and leaf functions (functions that do not call any function) with only one basic block are perfect candidates for such an optimization. Clocks can be removed from such functions and instead be added to the basic blocks calling such functions. Besides functions with only one blocks, our method also considers leaf functions with multiple blocks, given that there are no loops in such functions. If our pass sees that all possible paths taken by such a function do not differ by much, we calculate the mean value for all possible paths and use that mean value to update the clock. We call such leaf functions as *clocked functions*. By intuition, we can judge that it is also possible to clock functions which call only *clocked functions*. In this way, we can even clock functions which are not necessarily leaf functions.

Optimization 2 (Conditional Blocks Optimization) This optimization is based on the principle that if a block has two or more successors, given that those successors are not merge nodes, we can make the successor with the least clock zero and subtract its original value from all its siblings, while also adding its original clock to the parent block. Another principle of this optimization is that if all predecessors of a merge block have that merge block as their only successor, the clocks could be shifted from the merge node to them. It should be noted that after having parsed all the blocks of a function and applying this optimization, if it is still possible to apply this optimization once more, it is applied.

Optimization 3 (Averaging of Clock) This optimization is based on the fact that paths emanating from a block in a function could be matching close together in total clock values. One can imagine it as a specialized case of *Function Clocking*. For *Function Clocking*, we just considered the paths emanating from the entry block, but here we also check for paths besides the entry block. When forming paths for a block, we only consider blocks dominated by it (execution must pass through the dominating block to reach its dominated blocks). We also make sure there are no loops or unclocked functions in such paths. If we find such a block in a function, we remove clocks from all the blocks in the averaged path and assign the mean clock value to that block.

Optimization 4 (Loops Optimization) This optimization considers the fact that loops are often executed multiple times. So for example, if you have a *for* loop, the increment operation will take place just before the next iteration. Therefore we check for back edges and if we see that the clock of the block from which the backedge is originating is less that a certain threshold value and is also less than the clock of the block it is jumping to, we merge its clock value to that block's clock and remove clock updating code from it.

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 of type, input parameters and output of the system calls. Type and input parameters are then read by the follower for error detection, while the output logged by the leader is used directly by the follower, instead of actually executing the system call. Working only with user-space wrappers of system calls is possible, because most of the system calls 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 *rand* and preload them using LD_PRELOAD. Follower uses the output logged by the leader for such functions. Each system call is protected by a deterministic lock to make sure that system calls occur in the same order in the replicas. For the system call *mmap*, which modifies the address space of the process, we take a special approach. The follower still uses the returned addressed by the leader, but use it as a parameter combined with *MAP_FIXED* flag to call the *mmap* function, to ensure that the follower uses the same memory addresses as the leader.

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 IV-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 (modified) memory pages 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 comparison is made even faster by assigning each thread to calculate hash-sum of different portions of the memory. The leader keeps its hash-sums in shared memory so that the follower 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. Otherwise we create barriers at system call points if necessary (at the end of an epoch that is).

Since our scheme runs at the user 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. Therefore, we take special steps to improve its performance. 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. 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.

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 and recover from timeout errors. Details can be found in [6].

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. 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. More detail on this can be found in [6].

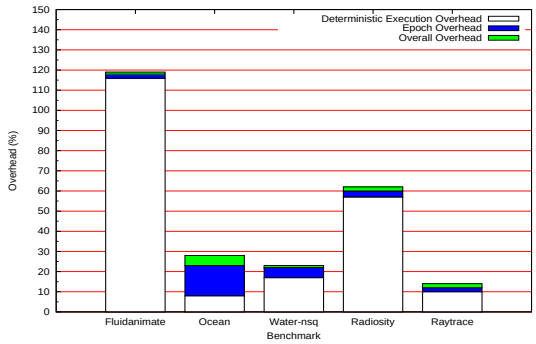

Figure 4. Overhead of our Fault Tolerance Scheme

V. Performance evaluation

We selected 5 benchmarks, 1 from the PARSEC [2] and 4 from the SPLASH-2 [9] 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 with maximum optimization enabled (level -O4 for clang/llvm). All benchmarks were run with 4 threads, meaning we had 8 threads overall for the 2 redundant processes.

A. Results

The results are shown in Table I. In this table, by *Redundant Exec*, we mean results obtained by allowing the leader and follower processes execute freely, without any deterministic execution and fault tolerance, while *Deterministic Exec* overhead is the overhead with deterministic execution only, whereas *Overall Exec* overhead includes all the components of our fault tolerance scheme. For overall execution, the results are shown with memory grouping size of 4. Figure 4 shows the different overheads separately. The epoch overhead here represents overhead of checkpointing, signals for noting dirtied (modified) memory pages and watchdog process. For benchmarks with high lock frequencies (*Fluidanimate* and *Radiosity*), the overhead of deterministic execution is expectedly quite large, whereas for *Ocean*, which has high memory usage, epoch overhead is the most. This is because for *Ocean*, large number of signals are received when memory pages are modified during an epoch.

B. Deterministic execution overhead

Figure 5 shows the deterministic execution performance improvement that we get by applying optimizations on the compiler pass that inserts code to update the logical clocks. The lower portion of the bars (in white color) in Figure 5 shows the overhead of deterministic execution with the optimizations, while the upper portion (in blue color) shows the additional overhead when optimizations are not applied.

Table I
PERFORMANCE RESULTS OF OUR SCHEME FOR THE SELECTED BENCHMARKS

Benchmark	Fluidanimate	Ocean	Water-nsq	Radiosity	Raytrace
Original Exec Time (ms)	1142	1870	1416	962	677
Locks/sec	6266655	5397	132798	6072689	225635
Pages Compared	3639	40195	380	8739	114
Epochs	3	3	2	2	1
Redundant (Redt) Exec Time and Overhead	1204 (5%)	1872 (0%)	1441 (2%)	980 (2%)	697 (3%)
Deterministic Exec Time and Overhead with No Optimization (w.r.t Redt Exec)	3072 (155%)	2022 (8%)	2102 (46%)	1892 (93%)	813 (17%)
Deterministic Exec Time and Overhead with Optimizations (w.r.t Redt Exec)	2603 (116%)	2014 (8%)	1688 (17%)	1534 (57%)	767 (10%)
Overall Exec Time and Overhead (w.r.t Redt Exec)	2639 (119%)	2391 (28%)	1771 (23%)	1584 (62%)	795 (14%)

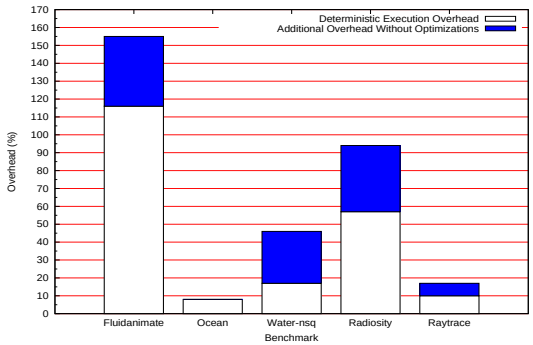

Figure 5. Improvement in Deterministic Execution Performance by applying optimizations

We get improvement in performance due to two reasons. Firstly, because of reducing the clock updating code and secondly by updating clocks ahead of time (See Section IV-A3 for discussion on the deterministic algorithm that we use and why updating clocks ahead of time is beneficial). For *Radiosity*, Optimization 1 (Function Clocking) is applicable on a large number of functions, with some of them being quite compute intensive. Therefore, by incrementing the clocks for those clockable functions ahead of time, we significantly reduce the waiting time for threads which are about to acquire a lock. For *Fluidanimate* and *Water-nsq*, the improvement was mostly due to the Optimization 4 (Loops Optimization) because these benchmarks contain compute intensive small loops. Optimization 3 (Averaging of Clock) worked well for *Raytrace* because our compiler pass could find such paths (for whom clocks could be averaged) in it. Furthermore, Optimization 2 (Conditional Blocks Optimization) was useful for reducing the clock overhead of most of the benchmarks, because such conditional paths are commonly found in programs.

VI. CONCLUSION

In this paper, we described the design and implementation of a user-level leader/follower based fault tolerance scheme for multithreaded applications running on multicore processors. Instead of using record/replay technique to ensure deterministic shared memory accesses by the replicas, we used *deterministic multithreading*, where the redundant processes do not need to communicate with each other for ensuring deterministic shared memory accesses. This improves isolation between the redundant processes, increasing fault tolerance and reliability, besides consuming less memory. To increase portability, we avoid using any special hardware for deterministic execution and modifying the kernel. We instead implemented a compiler pass that inserts code to update logical clocks for *deterministic multithreading*. We also applied several optimizations to reduce the overhead of logical clock updating code. In the absence of faults, our fault tolerance scheme, adds an average overhead of 49% for selected benchmarks, with 4 threads.

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

[4] 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.

[5] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. ASPLOS '09, pages 97–108.

[6] H. Mushtaq, Z. Al-Ars, and K. Bertels. A user-level library for fault tolerance on shared memory multicore systems. DDECS '12, pages 266–269.

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

[8] V. Weaver and J. Dongarra. Can Hardware Performance Counters Produce Expected, Deterministic Results? FHPM '10.

[9] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. ISCA '95, pages 24–36.

[10] 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, March 2001.

[11] 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.

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

[13] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. SIGMETRICS '10, pages 155–166.

[14] A. Basu, J. Bobba, and M. D. Hill. Karma: scalable deterministic record-replay. ICS '11, pages 359–368.

[15] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. PLDI '07, pages 89–100.