Towards Reliable Task Parallel Programs

Dimitrios Skarlatos[1], Polyvios Pratikakis[2], and Dionisios Pnevmatikatos[1 2]

[1] Technical University of Crete
[2] Foundation of Research and Technology – Hellas

Abstract. This paper presents a mechanism for fault-tolerance in task-parallel programs. We augment a task-parallel runtime system with support for transparent checkpoints of task data that may be written during task execution and seamlessly rerun failed tasks. The system can recover from transient errors during task execution within a single core by rerunning the failed task, as well as from permanent errors that disable a worker core by redistributing work among remaining cores. We have evaluated our implementation using six benchmarks and found that checkpointing incurs a performance overhead of 7.93% on average mainly due to the cost of memory copies, and only a negligible space overhead due to the recycling of checkpoint memory. We have tested the system for varying probabilities of transient or permanent errors, and found that the overall performance is consistently better than expected. We believe this to be due to cache warm-up effects that cause immediate re-execution of faulty tasks to be almost always faster than their first execution.

1 Introduction

Multicore and manycore processors are increasingly used in embedded and safety-critical applications. At the same time, the probability of failure increases with the number of cores and the complexity of interconnects in these processors. Fault-tolerance is thus a necessary property for any parallel application in this domain. Traditionally, software achieves fault-tolerance by (i) checkpointing the state of an application at points where it is safe to do so, (ii) relying on some mechanism (usually hardware) for detecting failures, and (iii) retrying computation from the last safe checkpoint on failure.

It is not straightforward, however, when a parallel system is at a consistent state, as all threads must synchronize to ensure the whole system is safe to checkpoint. This reduces the available parallelism, as all threads must wait for a global checkpoint. Per-thread checkpointing solves this problem, but it may require the programmer to reason about thread-local and thread-shared data, reason about the invariants of the program data structures and keep track of thread-changes transactionally. This makes the checkpointing of thread programs tedious and error prone.

Task-based programming models can express parallelism at a higher level of abstraction than thread programming. Tasks allow programmers to express the maximum parallelism in an algorithm without hard-wiring such parallelism into threads and making assumptions about the parallelism available in hardware. The compiler and runtime system can then extract parallelism as needed by mapping tasks to available cores, automatically detect dependencies and perform synchronization, and even produce deterministic parallel executions [11]. Task-parallel programming models offer a much

This work was supported in part by European Commission in the context of FP7 DeSyRe project (287611)

better abstraction for checkpointing an application, as tasks are atomic units of work that do not communicate or synchronize with other tasks and have a fixed set of inputs and outputs.

This paper presents RelyBDT, a fault-tolerant runtime system for the reliable execution of task-parallel programs. RelyBDT abstracts over checkpointing for the programmer by performing transparent checkpointing of parallel tasks. RelyBDT recovers from transient task faults by recomputing the faulted task, and from permanent core faults by rescheduling the failed core's tasks to different cores. We have implemented RelyBDT as an extension of BDDT [11, 5], a deterministic task-parallel runtime system.

The rest of this paper is organized as follows. Section 2 provides an overview of the fault model and our assumptions, Section 3 presents the design of the checkpoint mechanism, Section 4 presents the results of our implementation, Section 5 discusses related work and Section 6 concludes.

2 Assumptions and Fault Model

The semantics of task-parallel programming models make checkpointing easier and more efficient than in previous approaches targeting thread programs. Namely, a task is a computation that can run in isolation (usually a function call), has clearly defined input and output arguments, and is restricted to only access those. For example, the code `spawn f(input A, output B)` executes function `f(A,B)` in parallel to the rest of the program, taking into account that `f` reads its first and writes its second argument. Arguments can be either read-only (input), write-only (output) or read-write (`inout`). In implicitly synchronized task-parallel models like BDDT, moreover, tasks are guaranteed to have exclusive access to their output and `inout` arguments, because the runtime system automatically detects dependencies between tasks that access the same memory and schedules them correctly to avoid races. In BDDT, a "master" thread executes the main program and spawns parallel tasks, while a set of "worker" threads execute these tasks when all their dependencies are satisfied.

We take advantage of the task semantics and assume that tasks will respect their memory footprint (input and output arguments) even in the presence of faults. That is, we assume that the hardware can either detect faults as soon as they occur [7] or use memory protection mechanisms [8] to detect and stop a faulty task from corrupting any part of memory other than its arguments. This assumption greatly simplifies checkpointing, because the system need only restore task argument memory before retrying the task. Moreover, task arguments are accessed only by the task during its execution, meaning that both checkpointing and restoring from a checkpoint become thread-local operations, without requiring synchronization with other threads. In contrast, checkpointing a multithreaded application requires all threads to stop as any thread can access any memory address.

We assume two kinds of faults: (i) Transient faults that cause a computation to fail or compute the wrong result, while the hardware remains operational and it suffices to restart the computation. (ii) Permanent faults that cause hardware to fail and be unable to perform any computations. In both cases, we assume that the faults will occur during the execution of tasks, meaning that no faults can happen at the "master" core, nor when a "worker" is running the runtime system between tasks. In future work, we plan to extend the fault model and checkpointing implementation to support these cases.

3 Design

The BDDT runtime system uses a “master” thread to spawn tasks and a pool of “worker” threads to execute them. Under the assumptions described above, all checkpoint operations are run by the worker-threads. It is safe to checkpoint task data in parallel without synchronization, because of the semantics of tasks, namely that a task will never be scheduled concurrently with other tasks accessing the same data. Balancing the checkpointing, fault detection and fault-recovery workload across many threads gives scalability to our design and does not introduce bottlenecks.

3.1 Transient Faults

Worker threads checkpoint a task’s arguments just before executing it. Each worker thread uses a checkpoint buffer into which it copies the task arguments. We recycle the checkpoint buffer when possible, to avoid the overhead of allocating and deallocating checkpoint memory. The checkpoint only contains the contents of `inout` arguments, because input-only arguments can not get corrupted/altered during execution and output-only arguments are always rewritten during execution.

We assume that transient faults can be detected as they occur, so that task execution stops and the runtime system resumes control of the core. We simulated transient faults using a “coin-flip” test after every task to decide whether the task is faulty and needs to be re-run. We used a uniform-distribution pseudo-random number generator to generate faults with various probabilities. When a transient fault is detected, the runtime system restores the checkpoint by copying the contents of `inout` arguments from the checkpoint buffer to the original locations and retries the task. Note that in reality, a fault may be detected before the end of a task, causing it to be restarted earlier; our fault emulation thus overapproximates the overhead because it always allows the faulty task to terminate before retrying.

3.2 Permanent Faults

We assume permanent faults may completely disable a core while a task is running. We simulate permanent faults in the same way, using a “coin-flip” function based on uniform distribution to create various probabilities of failure. Recovery from permanent faults is more complex than for transient faults, because it involves different worker cores detecting the error and redistributing all work scheduled to the faulty worker core.

To detect permanent faults in worker cores we have the runtime in each worker core periodically check for permanent faults in other workers. Upon detection of a permanent fault, one of the live workers will take over recovery: disable the task queue of the faulty worker core so that no new tasks are scheduled there and redistribute all tasks in that queue to the remaining workers via task-stealing. The recovery worker also takes over the task that was running when the permanent fault occurred, restores its data from the checkpoint buffer of the disabled worker and executes the task locally.

4 Experimental Evaluation

We evaluate our implementation using six task-parallel benchmarksrunning on a 3.3GHz, 4-core i5-2500 CPU with 4GB RAM running Ubuntu 12.04 Server. All reported times are averages of 5 runs. We used the following benchmarks:

Black-Scholes is a mathematical model for financial markets, taken from the PARSEC [1] benchmark suite. We used 12,800,000 options and a 292MB dataset, resulting in 100,000 (dynamic) tasks. This application does not have any `inout` arguments, so it requires no checkpointing. The overhead shown only corresponds to the re-execution of tasks.

GMRES computes the generalized minimal residual method for solving non-symmetric systems of linear equations. We used 13,107 nodes and a block size of 128MB, which amounts to 249,717 tasks. This application models the worst case checkpointing scenario, where all tasks use `inout` arguments and memory copy operations become a bottleneck. Moreover, each task uses a small part of memory, requiring lots of small checkpoints. The overhead is the combination of the checkpoint functions and the re-execution of tasks.

FFT is a kernel with Blocking Transpose taken from the SPLASH-2 [13] benchmark suite. We used 16,777,216 Complex Doubles resulting in 28,864 tasks that have strided `inout` arguments that correspond to array tiles, causing overhead due to multiple memory copies per argument.

Cholesky is a linear algebra kernel uses 128×128 matrices, resulting in 5,984 tasks. The tasks have strided `inout` arguments corresponding to tiles of the total array, resulting in several copying operations per argument. Moreover, the tiles are large, and so the checkpoints have a large memory and time overheads due to copying.

Jacobi is a kernel using the Jacobian method for solving linear equation systems. We used a 7168×7168 matrix tiled into 128×128 blocks for 30 iterations, resulting into 94,080 tasks. Jacobi uses two matrices to avoid in-place computation; its tasks read from one and write to the other array, swapping the arrays for the subsequent iterations of tasks. Even though Jacobi uses strided arguments to describe array tiles it incurs no checkpointing overhead, because there are no `inout` arguments that require copying.

Multisort is a task-parallel version of Mergesort similar to Cilksort [2]. We used 20MB of data for 836 tasks. Multisort does not have `inout` arguments, each task uses read-only inputs and a write-only buffer to output its results. Effectively there are no checkpoints for Multisort tasks, and the only overhead corresponds to retrying failed tasks.

4.1 Time Overhead

To compute the overhead of checkpointing, we measure the running time of each benchmark using the original BDDT runtime without any support for fault tolerance and checkpointing. We also measure running time using RelyBDT for zero permanent faults, while varying the probability of transient faults.

Figure 1 shows the overhead incurred by checkpointing as a percentage over the baseline run. Note that the leftmost data points correspond to a transient error probability of zero, in effect measuring the "protection" cost of checkpointing all `inout` arguments in an application, even in the case where no checkpoint is ever used. Benchmark performance varies depending on the cost of allocating space and creating the checkpoint, relative to the cost of the task. As expected, benchmarks that do not require check-

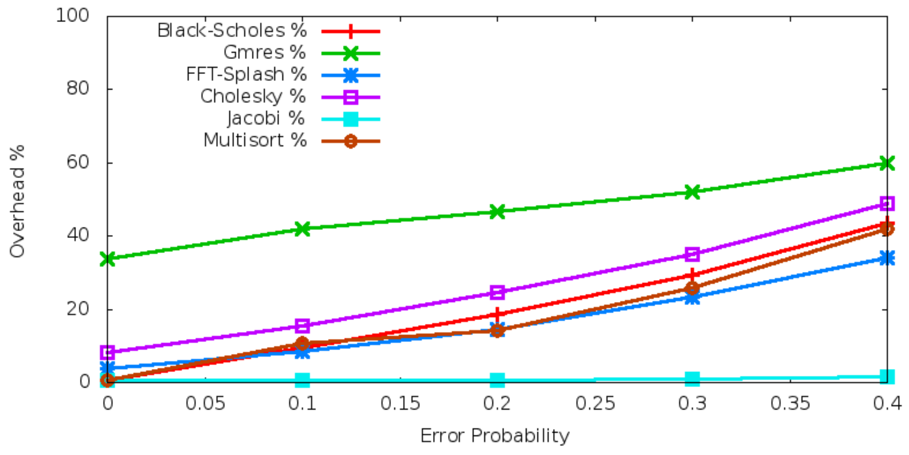


Fig. 1: Time overhead percentage over no-checkpointing with transient error probability ranging from 0.0 to 0.4 and no permanent errors.

points (have no `inout` arguments) do not show any checkpointing overhead. The overhead is similarly low on computationally-heavy benchmarks like FFT and Cholesky, because the cost of allocating the checkpoint and copying the data is dwarfed by the time spent in the actual task. On the other hand, applications like GMRES that have small, fast tasks operating on large data footprints suffer significant overhead, because creating the checkpoint takes time comparable to the actual task execution.

Figure 1 also shows three other data points per application, for executions with probability of transient error being 0.1, 0.2, 0.3 and 0.4 (10% to 40% error rate). Note that a probability of 0.1 for transient errors means that 10% of all tasks will fail and rerun, whereupon 10% of the reruns will also fail, etc.

In theory, a probability of $0 \leq x \leq 1$ of failure and rerun will execute a total number of tasks equal to

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

meaning that at probability of failure 0.1, 0.2, 0.3 and 0.4 we expect to perform respectively 11.11%, 25%, 42.86% and 66.67% more work due to retrying failed tasks. To test this hypothesis, we use a different configuration comparing the total execution time for these probabilities of failure, with the execution time using zero probability, to avoid counting the cost of checkpointing together with the cost of retrying failed tasks. Figure 2 presents the results, along with a curve showing the theoretically expected overhead just by rerunning the failed tasks. Note that in all benchmarks the cost is considerably lower than expected. We believe this to be because each worker core that detects a task failure immediately repeats the task execution, thereby taking advantage of warmed caches and locality. Almost consistently, the second time a worker core executes a task it is faster, as tasks are sufficiently small to fit in the processor cache.

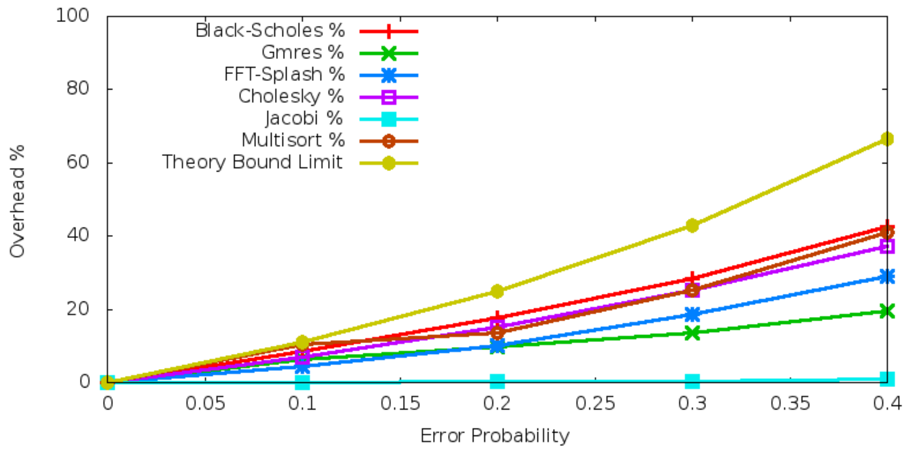


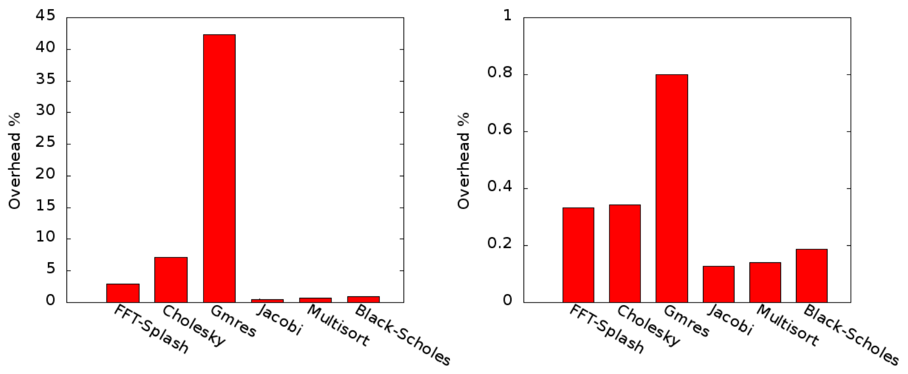
Fig. 2: Time overheads due to retrying failed tasks: comparison of total execution time overhead for various probabilities of failure with total execution time of checkpointing without failure.

We measure the overhead of permanent faults on total execution time similarly. As a baseline, we measure the total execution time of each benchmark using (vanilla) BDDT with three worker threads. We compare this to a RelyBDT execution with four worker threads, in which the first task always causes a permanent fault, causing one of the worker threads to stop. We set the probability of transient faults to zero. Note that even at a zero probability of transient faults, RelyBDT still entails the overhead of creating checkpoints. Figure 3a presents the overhead in total execution time. This includes the cost of checkpointing plus the cost of recovery from exactly one permanent fault. Note that this overhead is not directly comparable with the overhead of checkpointing reported in Figure 1 as it refers to executions with three worker threads, whereas Figure 1 refers to four worker threads.

To estimate the actual cost of a permanent fault, we perform another set of measurements. We measure the total running time for each benchmark executed using RelyBDT with three worker threads and zero probability of either transient or permanent faults; this baseline includes only the application time plus the overhead of creating checkpoints. We compare this against the total execution time using RelyBDT with four starting worker threads that immediately become three due to a permanent fault at the first task. Figure 3b shows that the overhead of recovering from one permanent fault is negligible, at worst 0.8% of execution time. Again, GMRES has a higher overhead mainly because it creates a lot of small tasks, meaning that recovering from a permanent fault introduces more task-steals from the faulty core to the remaining workers.

4.2 Space Overhead

Task-based checkpointing incurs little space overhead overall: only one task can run per worker at any given time and task semantics allow the system to only maintain



(a) Total overhead over BDDT (b) Overhead of a permanent fault

Fig. 3: Permanent faults

checkpoints for the arguments of running tasks. We measure the space overhead of checkpoints for each application per worker, that is, the maximum space used by any given worker at any given time to store a checkpoint. Figure 4 shows the maximum space required to checkpoint a task in each benchmark. Note that although GMRES creates many tasks, each task has a very small memory footprint (1 KByte), resulting in minimal space overhead for checkpointing all running tasks at any given time. On the other hand, FFT and Cholesky spawn tasks that operate on large tiles of large arrays, requiring a lot of space to store their checkpoints.

Table 1 shows the memory operations performed by checkpointing for each application. The second column shows the number of tasks per application. The third and fourth columns show the number of calls to `malloc()` and total size of allocated memory that stores checkpoints. The fifth column shows the number of `memcpy` calls needed to create the checkpoints. Note that this may be higher than the number of allocated checkpoints, due to strided arguments like array tiles that require multiple copies per task argument. The last four columns show the amount of memory that had to be copied back before rerunning a failed task, for four probabilities of fault.

Table 2 shows the memory footprint of the application and runtime, as well as the overheads due to keeping checkpoints. In cases where the memory footprint changes throughout execution, we present the high watermark: the maximum space required at any given point during execution. The first column shows the benchmark name. The second column shows the amount of memory reserved by the runtime system to contain task descriptors, the dependency graph, etc. The runtime takes a fixed amount of space, regardless of the application requirements. This is because it reserves a predefined maximum amount of memory that depends on the available hardware resources. The third column shows the amount of memory allocated by the application to store data processed by its tasks and the fourth column shows the maximum amount of memory used to store checkpoints at any given point during the execution. This amount is sensitive to the number of worker threads used. The number shown corresponds to four

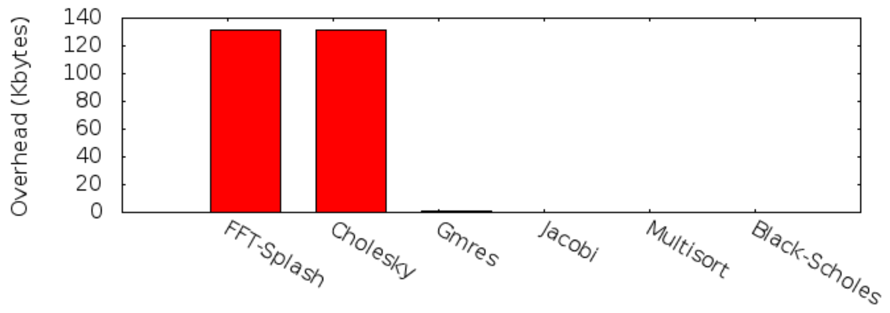


Fig. 4: Worst case space overhead per application for each worker.

Benchmark	Tasks	`malloc()` Calls	`malloc()` KBytes	`memcpy()` Calls	KBytes Restored / Probability 10%	20%	30%	40%
Black-Scholes	100,000	0	0	0	0	0	0	0
Cholesky	5,984	4,459	566,467	570,726	65,851	142,475	223,085	339,706
FFT	28,864	46,679	1,104,229	1,397,305	123,529	268,415	45,381	674,288
GMRES	249,724	39,442	39,372	39,442	4,444	9,817	16,539	24,842
Jacobi	94,080	0	0	0	0	0	0	0
Multisort	836	0	0	0	0	0	0	0

Table 1: Space overhead and checkpoint memory operations.

worker threads, meaning that, for instance, each worker thread in FFT needs 131072 bytes of memory to checkpoint the arguments of the task with the largest footprint. The fifth column presents the required checkpoint state as an overhead over the application data size. For all benchmarks it is much less than 1%. The last column shows the total maximum space required for checkpoints and corresponds to instances of the largest task running on all worker threads.

5 Related Work

Task Parallelism: Traditional task-parallel programming models try to abstract over threads and offer a higher-level abstraction for expressing parallelism. Cilk [2] is a task based, shared memory, programming model that allows the programmer to specify recursively spawned tasks, which are efficiently scheduled on threads using continuations. Sequoia [9] is a programming language used in development of parallel, hierarchy-aware and portable applications. In these models the programmer is responsible for synchronizing parallel tasks to avoid address space aliasing and races.

Dependence analysis in task-based programming models has been shown to improve performance of general purpose programs [12]. In these systems tasks define their effect *input*, *output* or both (`inout`) on their arguments. The runtime then uses versioned hyperobjects to track task argument dependencies and versioning of objects in order to increase parallelism. To break anti-dependence and output dependencies a versioning mechanism determines the view of an object for each thread.

Benchmark	Runtime Space	Data	Checkpoints	Overhead%	Total
Black-Scholes	404,111,456	358,400,000	0	0.00%	762,511,456
Cholesky	404,111,456	134,217,728	+524,288	0.39%	538,460,256
FFT	404,111,456	805,437,512	+524,288	0.07%	1,209,680,040
GMRES	404,111,456	73,453,000	+4,096	0.01%	477,565,480
Jacobi	404,111,456	881,852,416	0	0.00%	1,285,963,872
Multisort	404,111,456	167,772,160	0	0.00%	571,883,616

Table 2: Memory footprint (bytes) and overhead per application

*Reliability and Checkpointing:*RAFT [14] is a speculative runtime fault tolerance mechanism designed for single-threaded applications with deterministic output. It duplicates the original application and executes both versions in parallel, using double the amount of registers and memory. RAFT speculates the return values of system calls, avoiding synchronization barriers and only verifies values that escape the user space.

Shoestring [4] is a symptom-based instruction duplication technique that provides opportunistic soft error reliability. A compiler analysis is utilized to find vulnerable code, based on a specific set of symptoms (e.g., memory access exceptions). The instruction duplication mechanism selects a tree of instructions to duplicate. Checker nodes are injected along the code to compare the results of the leaf nodes from the original and the duplicated tree.

SWIFT [8] is a single-threaded, compiler-based fault tolerant technique which injects duplicate and comparison instructions at compile time to detect faults. The memory system is protected through the use of error correcting code (ECC). SWIFT uses an optimized control-flow checking mechanism which utilize control blocks with dynamic signatures, to avoid the cost of branch validation code.

ASSURE [10] provides a fault tolerance technique for server applications by implementing rescue points with a moderate performance overhead. Rescue points are locations in the application where error handling takes place and can be found offline. When a "live" error occurs for the first time,a new personal copy of the application is used to search for rescue points that are able to handle the error. Once a suitable point is found, a patch is applied trough binary injection to the online application. The patched version of the application is able to take checkpoints at the rescue point, so when the same error occurs a rollback can take place and recover from the fault automatically.

Static techniques have also been used in the past to identify *idempotent regions* of code that can be re-executed without checkpointing [3]. Such analyses are orthogonal to our work and can be used complementarily to identify and optimize unnecessary checkpoints, further reducing overhead.

Transactional Memory: The notion of rollback and retrying execution is inherent in Transactional Memory (TM) models [6]. Several software transactional memory runtimes use checkpointing or similar techniques to save and restore the state of transactional variables to consistent points. TM, however, is a lower-level model than task-parallelism; TM programs use threads to express parallelism and TM to enforce synchronization. Moreover, TM most often enforces a mutual exclusion semantics of synchronization, whereas runtime dependence analysis in task-parallel models enforces a dataflow semantics that deterministically produces the same result as the sequential program.

6 Conclusions

In this paper we describe a mechanism and a runtime system for the fault-tolerant execution of programs. We assume that a task-based programing model is used to describe the user application, and that faults (either transient or permanent) are detected in hardware. Our proposed runtime system can transparently re-execute failed tasks and achieve correct execution of the application. We found that for our benchmarks, saving enough program state to re-execute the tasks requires almost negligible storage space, while the performance overhead depends strongly on the benchmark behavior. These results are very encouraging and suggest that a task-based programming paradigm is very well suited for environments where fault-tolerance is required. Future extensions of our work include addressing distributed memory environments, and extending the fault coverage to the runtime itself.

References

1. C. Bienia, S. Kumar, J. Pal Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.
2. Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *PPoPP*, 1995.
3. Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static analysis and compiler design for idempotent processing. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, 2012.
4. Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *ASPLOS*, 2010.
5. Foivos Zakkak Hans Vandierendonck Polyvios Pratikakis George Tzenakis, Angelos Papatriantafyllou and Dimitrios S. Nikolopoulos. Bddt: Block-level dynamic dependence analysis for deterministic task-based parallelism. Tech Report 426, FORTH-ICS, February 2012.
6. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, 1993.
7. Joydeep Ray, James C. Hoe, and Babak Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *MICRO*, 2001.
8. George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. Swift: Software implemented fault tolerance. In *CGO*, 2005.
9. The sequoia programming language. http://http://sequoia.stanford.edu.
10. Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. Assure: automatic software self-healing using rescue points. In *ASPLOS*, 2009.
11. G. Tzenakis, A. Papatriantafyllou, J. Kesapides, P. Pratikakis, H. Vandierendonck, and D. S. Nikolopoulos. BDDT: Block-level dynamic dependence analysis for deterministic task-based parallelism. In *PPoPP*, 2012. Poster paper.
12. H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos. Parallel programming of general-purpose programs using task-based programming models. In *HotPar*, 2011.
13. S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.
14. Yun Zhang, Soumyadeep Ghosh, Jialu Huang, Jae W. Lee, Scott A. Mahlke, and David I. August. Runtime asynchronous fault tolerance via speculation. In *CGO*, 2012.