A runtime heuristic to selectively replicate tasks for application-specific reliability targets

Omer Subasi[1,2], Gulay Yalcin[3], Ferad Zyulkyarov[1], Osman Unsal[1], Jesus Labarta[1,2]

[1]*Barcelona Supercomputing Center,* [2]*Universitat Politecnica de Catalunya, Spain,* [3]*Abdullah Gul University, Turkey*
{*omer.subasi, ferad.zyulkyarov, osman.unsal, jesus.labarta*}@*bsc.es, gulay.yalcin@agu.edu.tr*

Abstract—**In this paper we propose a runtime-based selective task replication technique for task-parallel high performance computing applications. Our selective task replication technique is automatic and does not require modification/recompilation of OS, compiler or application code. Our heuristic, we call App_FIT, selects tasks to replicate such that the specified reliability target for an application is achieved. In our experimental evaluation, we show that App_FIT selective replication heuristic is low-overhead and highly scalable. In addition, results indicate that complete task replication is overkill for achieving reliability targets. We show that with App_FIT, we can tolerate pessimistic exascale error rates with only 53% of the tasks being replicated.**

I. INTRODUCTION

As High Performance Computing (HPC) systems grow in size and complexity, they become more vulnerable to faults [9]. Moreover it is expected that hardware-only fault-tolerance solutions will not be adequate to handle the expected error rates [13] in the future. Thus, software-based solutions must complement hardware techniques to address the reliability of future HPC systems and applications. These solutions can be provided at programming model (PM) and/or runtime level. Currently task-based parallel PMs are becoming widely used to implement HPC applications for achieving higher performance [10]. In addition, it has been shown that dataflow execution model improves performance of HPC applications with asynchronous execution [4]. As a result, programming platforms such as OpenMP 4.0 [30] and Intel Threading Blocks (TBB) [2] have recently added support for task-based dataflow parallelism. Thus we find that it is important to provide software-based fault-tolerance for task-parallel dataflow HPC applications.

Redundant computation and checkpoint/restart are two well-known techniques to achieve fault-tolerance. In redundant computation, multiple replicas of a program are executed in parallel, such as task replication in a task-based HPC application. Redundant computation can be used for recovering from task failures as well as for detecting silent errors. It recovers from task failures since if a task replica fails, the remaining replicas can still continue their computations. It detects silent errors, such as data corruptions, by comparing the results of the replicas. A data corruption is called *silent* if it is undetected. Silent data corruptions

(SDCs) jeopardize the correctness of the results of HPC applications [18] and as a result they pose a significant threat. However, detecting SDCs is not sufficient, it is also necessary to recover from SDC errors. Checkpoint/restart can be utilized for SDC error recovery. In checkpoint/restart, the state of the computation, called checkpoint, is saved periodically and when a SDC error is detected, the computation restarts from the latest checkpoint thus recovering from SDC. In this work we combine redundant computation - in our case replication of application tasks - and checkpoint/restart to address SDCs and failures of task-parallel HPC applications while increasing reliability.

The straightforward way to achieve this goal is to replicate all application tasks, that is, complete task replication[1]. However complete task replication may be prohibitive due to the high resource cost and in fact might be excessive due to the uneven susceptibility of the different program parts to SDCs [24]. Therefore effective and efficient techniques are needed to selectively replicate tasks. However the optimal selective replication is NP-hard which can be formalized as a bounded knapsack problem [26]. Therefore, practical selective replication solutions must employ heuristics. In our main contribution, we propose a runtime-based, fully automatic and completely transparent heuristic, called App_FIT, to selectively choose tasks for replication. Our design does not require any modifications at all to application code or operating system.

With App_FIT, users can set the desired reliability in Failures in Time (FIT)[2] that their application requires and our heuristic transparently and automatically replicates tasks selectively to make sure that this target reliability is achieved. The App_FIT heuristic is useful when application users need the flexibility to specify the required level of reliability since different applications may have different reliability requirements as shown in [16].

In our experimental evaluation, we find that complete task replication over-allocates resources. In fact, we show that by using our selective task replication heuristic App_FIT heuristic, we can tolerate pessimistic exascale error rates with only 53% of task replication. Moreover, results show

---

[1]We use replication to refer to replication and checkpoint/restart together.
[2]FIT is a commonly used unitless reliability metric defined as number of failures per billion hours.

that App_FIT has very low overheads by smart replication of tasks. In fact, the fault-free performance overhead of App_FIT is found to be negligible.

We highlight two findings from our experiments and the analysis of the results. First, we find that complete replication is not needed to cope with the future HPC error rates. Second, our evaluation confirms our intuition that fault-tolerance based on task-parallel dataflow programming is efficient, scalable and low-overhead. Briefly, our contributions are:

- Design, implementation and evaluation of low overhead and highly scalable selective task replication.
- An automatic and efficient heuristic to selectively replicate tasks while reducing costs significantly.

The rest of this paper is organized as follows: Section II presents background and our motivation. Section III provides the design of task replication. Section IV discusses our heuristic. Section V presents the experimental evaluation. Section VI surveys related work. Finally, Section VII summarizes this work.

II. BACKGROUND AND MOTIVATION

This section introduces the error classification and failure model (Section II-A). Then it discusses task-parallel dataflow programming (Section II-B). Finally, it presents our motivation for selective replication (Section II-C).

A. Error Classification and Failure Model

Throughout this study, we refer to failures or errors as the manifestation of faults. Errors are classified into three categories based on their propagation (or lack thereof) from typical error detection/correction hardware. The first class is the Detected and Corrected Errors (DCE) where an error is detected and corrected by the hardware. The second class consists of errors that are Detected and Uncorrected Errors (DUE) where the hardware is unable to recover from the detected error. DUEs are expected to become more frequent in the future with the increasing likelihood of double-bit and multi-bit flips [15, 35] for caches and memory. Moreover, a single bit flip in parity protected processor structures such as register files could also lead to a DUE. DUEs typically result in the crash of applications since it is not possible for the faulted processor/hardware to recover [42]. The third class of errors consists of Silent Data Corruptions (SDCs). In SDC, the error is not detected, and the application terminates with wrong results. Recent research suggests that SDC can be a serious threat for HPC and exascale [18, 34]. A previous study at CERN found that SDC could be a serious concern since the observed SDC rate was orders of magnitude higher than manufacturer specifications [31]. Thus in this study we target SDCs and DUEs.
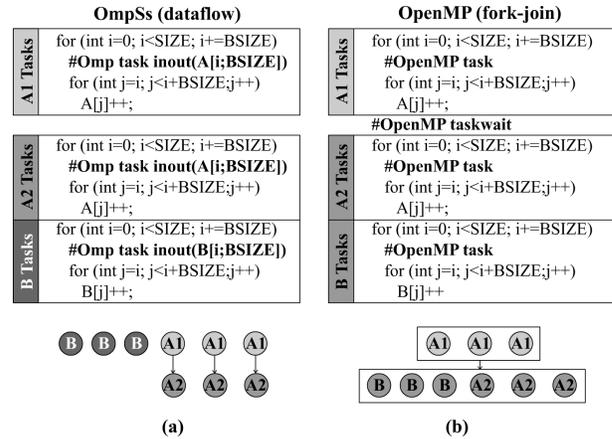

Figure 1. Example code in Dataflow and Fork-join.

B. Task-parallel Dataflow Programming

Tasks, as a higher level abstraction than threads, provide a more natural interface for expressing parallelism in parallel programs. Depending on how tasks are scheduled for execution and how they interleave during execution, task-based parallel programming can be either fork-join parallelism [11] or dataflow parallelism [21]. In both, tasks execute in parallel but are synchronized differently. Under fork-join parallelism tasks have to be explicitly synchronized with a join barrier whereas under dataflow parallelism tasks are synchronized implicitly depending on their inputs and outputs. Annotating these inputs and outputs of tasks in a correct and complete way is the programmer's responsibility as a programming discipline in dataflow programming models. However they are most often simply the inputs and outputs of functions. There are also tools for the automation of the annotations [39]. One can find example implementation of fork-join tasks in OpenMP 3.0 [12] and dataflow tasks in OmpSs [14], OpenMP 4.0 [30] and Intel TBB [2].

A recent comparison between dataflow and fork-join parallelism by Amer et al. [4] suggests that the dataflow execution model tends to perform better because it exploits the available parallelism better. This can be demonstrated with a simple contrived example in Figure 1. Figure 1 (a) is an example dataflow code in OmpSs PM with three tasks A1, A2 and B each incrementing the elements of an array. In tasks A1 and A2, the inout keyword is used to declare array A as both input and output. In similar way, array B in task B is declared as both input and output. Figure 1 (b) is the same example but for fork-join in OpenMP 3.0 PM. The difference between Figure 1 (a) and (b) is that the fork-join tasks do not declare explicitly their inputs and outputs and the fork-join code has the taskwait pragma directive between tasks A1 and A2. The figures under the code are the dependency graphs of the tasks for dataflow and fork-join, respectively. In dataflow the dependencies between the tasks are inferred from tasks' inputs and outputs whereas the dependencies in fork-join has to be enforced explicitly with a synchronization barrier like the taskwait directive.

Such synchronization in fork-join is necessary because the input of task A2 is the output of task A1 and task A2 cannot start execution before task A1 completes. However, using a `taskwait` barrier also prevents the execution of task B although it does not depend on neither task A1 nor task A2. In dataflow the dependence between the tasks is more accurately reflected indicating that task B can execute even before task A1 if its input is ready.

C. Motivation for Selective Replication

We propose selective task replication as a practical solution for real-life HPC programs that do not require complete fault coverage. At high level, we foresee the following scenario for using selective task replication. In this scenario, the user sets the desired application reliability in terms of Failures in Time (FIT). This scenario is related to the observation that different applications may have different reliability requirements. Hence the conservative approach of replicating the entire execution may not be necessary. Instead, replicating only the reliability critical parts of the application might be sufficient to satisfy the application's reliability requirements. In fact, the pioneering work of Fang et al. [16] finds that the algorithmic characteristics of parallel programs correlate with the error resilience properties. They report that different applications show different level of resilience and exhibit different SDC rates. Hence the application users can specify the level of resilience (FIT) for their applications and our heuristic then automatically decides at runtime the tasks to replicate such that the reliability of the application is always below the specified FIT threshold.

III. TASK REPLICATION

This section details our task replication implementation. We implement our framework in publicly available OmpSs PM and its Nanos runtime. However, our selective task replication heuristics are applicable for other task-parallel dataflow platforms. Nevertheless, the performance of OmpSs+Nanos is on par with the highly optimized commercial and open source implementation of OpenMP [5], [6] and it has successfully served as a pilot platform to push dataflow task parallelism to OpenMP 4.0. In the case of the distributed OmpSs+MPI model, it combines dataflow execution with the message passing model providing significant performance benefits. It hides the communication latencies and achieves higher performance compared to MPI only model [25].

Baseline Task Replication Design: Figure 2 shows our design in action. At the beginning of the task, the task's inputs are checkpointed ①. Then a replica is created by creating a duplicate *task descriptor* ②. In Nanos a task descriptor is an internal data structure which represents an instance of a task. It wraps the inputs and the outputs of the task as well as a pointer to its code. Both the original task descriptor and its replica are scheduled for execution. Idle threads from a thread pool poll the internal structures
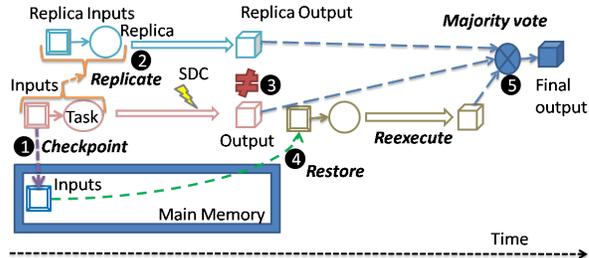

Figure 2.   Task replication design: SDC mitigation

which store the scheduled task descriptors and execute them asynchronously. The original task descriptors and their replicas are executed in parallel but they are synchronized at the end of their execution. This is the only synchronization point where the results of the task and its replica are compared ③. The inequality of the results signifies that SDC has occurred. In this situation, first the task's initial state is restored from its checkpoint and is re-executed ④. Then all three results are compared and the majority vote is selected as the task's result ⑤. Although we use bitwise comparison in this design, other comparators such as residue error checkers can easily be deployed in the runtime.

Selective Task Replication Design: The criteria for task selection is the rate that is estimated for a task based on its argument size. This rate indicates how likely a task will fail. At runtime the failure rates for a task are estimated before the execution of a task and the decision for the selection of the task for replication is taken by our heuristic. We will elaborate on how failure rates are estimated in Section IV-A.

IV. TASK SELECTION HEURISTIC

When selecting tasks dynamically at runtime, our goal is to avoid utilizing profiling information, which requires a prerun as well as to avoid collecting additional information at runtime since both are expensive. Therefore we propose our heuristic, called App_FIT, that makes use of only already existing information at runtime to achieve efficient, lightweight and near-optimal selective task replication. By using the free information, i.e. information about task inputs and outputs, from dataflow, we are able to design and implement App_FIT which does not require any profiling. We will now first present how we estimate failure rates for tasks and then we will present App_FIT.

A. The Estimation of Failure Rates

In this subsection we first present how we calculate failure rates in FITs for each application task and for the whole application/benchmark. We use the FIT rates for crashes (DUEs) and SDCs of Michalak et al. [29] for the Roadrunner supercomputer. Michalak et al. obtained these rates via accelerated neutron-beam test. We take the FITs of a Roadrunner TriBlade node and adjust them for each individual task and each benchmark proportional to their task argument sizes which are available at the beginning

of task executions. Let us call the task failure rates for crashes/DUEs and SDCs as $\lambda_F(T)$ and $\lambda_{SDC}(T)$ respectively. Consequently, the higher the argument size, the higher the estimated failure rate is. Similarly the benchmark FIT rates are estimated with respect to size of the benchmark input size. We use the benchmark FIT rates to calculate and specify the target reliability thresholds which are to be achieved by the App_FIT heuristic. For instance, if the crash failure is 2.22×10^3 for 32 GBs as given in [29], then for 32 MB program input the crash failure would be 2.22, or for a task argument of 32 KB the crash failure would be 2.22×10^{-3}. Finally a task's overall failure rates $\lambda_F(T)$ and $\lambda_{SDC}(T)$ are sum of all its arguments' failure rates respectively.

We now elaborate on how our framework is orthogonal to the failure rate estimation and analysis studies. Our framework is independent of the method for estimating/measuring the failure rates of each individual tasks or benchmarks. As stated above, we use DUE and SDC rates from the measurements of [29]. However, these rates can be obtained by any other methods such as the analysis of system failure (memory, storage, network) histories/logs or application/task-specific vulnerability analysis. Such studies are orthogonal and independent and can be seamlessly integrated to our heuristic. Moreover, these studies can account for various additional features that can affect the reliability factors of individual tasks. These features are in essence captured by refining task failure rates. For instance, the reliability factor of a task can be affected by the feature that the task contains many silent stores [23] which would mask any prior SDC at the memory location of the store operation. This will be captured by a vulnerability analysis in terms of a lower failure rate. Our heuristic, without any further modification, will simply make use of this refined task failure rate instead of the previous rate.

We assume that the checkpoints are stored in a safe memory region such that their failure rates can be ignored. For the voter, since it memory footprint is small, its failure rates are small enough to be considered safe. However, in any case, we can increase reliability by taking multiple checkpoints and using multiple voters without incurring too much performance penalty since the overhead of taking checkpoints and voting is low enough to perform them multiple times.

B. App_FIT heuristic

App_FIT heuristic is for the scenario where the user aims to run its application under a FIT threshold and while the application is executing, the threshold is never exceeded. Given that the user knows the FIT threshold, we assume it also knows the total number of tasks which the runtime takes as an input. Let N be total number of tasks. While the execution continues, when a task is about to execute, App_FIT checks atomically the following condition

Table I
DETAILS OF OUR TASK-PARALLEL BENCHMARKS

Shared-memory Benchmarks	
Sparse LU	LU decomposition Matrix size 12800x12800 doubles, block size 200x200
Cholesky	Cholesky factorization Matrix size 16384x16384 doubles and block size 512x512
FFT	Fast Fourier Transform Matrix size 16384x16384 complex doubles, block size 16384x128
Perlin Noise	Noise generation to improve realism in motion pictures Array of pixels with size of 65536, block size 2048
Stream	Linear operations among arrays Array size 2048x2048 (doubles), block size 32768
Distributed Benchmarks	
Nbody	Interaction between N bodies Array size 65536 bodies, block size depends on #nodes
Matrix Multiplication	Matrix Multiplication using CBLAS Matrix size 9216x9216 doubles and block size 1024x1024
Pingpong	Computation and communication between pairs of processes Array size 65536 doubles, block size 1024
Linpack	HPL Linpack Matrix size 131072 doubles, block size 256, 8x8 grid

to decide whether to replicate the task T:

$$current\_fit + (\lambda_F(T) + \lambda_{SDC}(T)) > (threshold/N) \times (i+1)$$
(1)

where $current\_fit$ is the current FIT of the computation at the time which is maintained by App_FIT, $\lambda_F(T)$ and $\lambda_{SDC}(T)$ is task T's estimated crash and SDC failure rates respectively, $threshold$ is the specified threshold by the user and i is the number of tasks that had been decided on so far. If the condition holds, it means that if App_FIT does not replicate the task, then after the task computation finishes the $current\_fit$ will surpass the intended threshold portion for the tasks (including the current task) finished by that time. Therefore, App_FIT decides to replicate the task. After the task finishes, App_FIT updates $current\_fit$ by adding the FIT of the task. If the condition does not hold, App_FIT does not replicate the task.

App_FIT, in its current design, only adds tasks to replicate. It could have been designed such that some replica tasks are removed dynamically however this has the drawback of losing the reliability obtained from and the computation of the removed tasks. In its current design, there is never such loss. In addition, removal of tasks would require dynamic inspection of task which would incur additional performance penalty.

V. EVALUATION

In this section we provide the evaluation and analysis of our technique. As stated earlier, we implement our ideas in OmpSs [14] and Nanos [40]. We perform our experiments in the Marenostrum supercomputer [3]. Up to 64 nodes and 16 cores per node are used in the experiments. Table I summarizes our benchmarks [1]. In addition to shared-memory benchmarks, we have distributed benchmarks to evaluate the performance overheads of task replication and our heuristics and their impacts on the application scalability at large scale and with high core counts. In shared-memory benchmark experiments all 16 cores in one node are used. In distributed benchmark experiments 1024 cores over 64 nodes are used. We run each experiment $10\times$ and report averages.
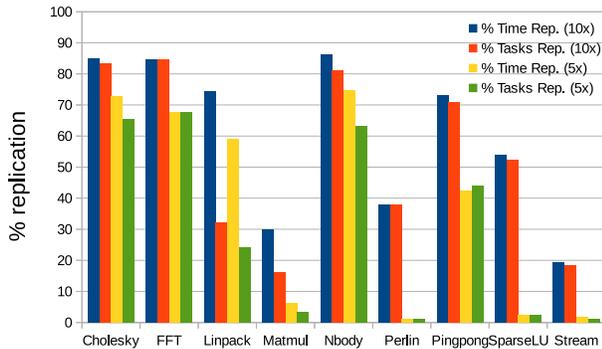
A. Experimental Results

Figure 3.   App_FIT results

1) Evaluation of App_FIT Heuristic: We evaluate App_FIT designed for obeying pre-specified FIT thresholds to see whether complete replication is needed to handle future error rates by specifying the thresholds such that their reliability matches today's systems. We set thresholds as follows: It is expected that in future HPC or exascale systems the error rates in a single node will increase about one order of magnitude [32]. To handle this increase, we decrease the current FITs of our benchmarks by $10\times$ so that the overall application FITs, thus their reliabilities, stay the same.

Figure 3 shows the replication results of App_FIT. The figure shows the percentage of computation time replicated due to the replication of tasks and the percentages of the number of tasks replicated for $10\times$ and $5\times$ error rates. On average App_FIT replicates only 53% and 30% of the tasks, and 60% and 36% additional execution time to keep the same reliability level at $10\times$ and $5\times$ rates respectively[3].

Takeaway-1: *Results show that complete replication is not required for the predicted exascale error rates to achieve the same reliability levels as today. Moreover the amount of replication can be decreased further by assuming modest increases in error rates or relaxing reliability requirements.*

Generally the percentages of the number of tasks replicated and the percentages of computation time replicated are close except for Linpack and Matmul. This is because they have some tasks that are clearly more distinctive than other tasks in terms of their FITs because of their memory usage and execution times. Moreover, the difference in terms of the percentages of replication across the benchmarks is mainly due to the task granularity and the number of tasks. That is, the more and the finer-grain the tasks are, the less the percentage of replication is. Coarser and low number of tasks restrains App_FIT to obey the threshold in a more efficient way. For instance, Cholesky, FFT, and Nbody have relatively coarser and low number of tasks and thus they incur more replication. In contrast, Stream, Matmul and Perlin have high number (25K-48K) of finer tasks. Another observation is for

---

[3]In our experiments App_FIT achieves FITs that are lower and close to the specified FITs. For brevity we omit the current FITs of benchmarks, the specified thresholds and the thresholds that App_FIT achieves.

Perlin and SparseLU there is a significant difference between the resulting replication percentages when $10\times$ and $5\times$ rates are used. This is because there is a few number of tasks whose reliability impacts are much higher than others and their selection for replication is sufficient to obey $5\times$ rates.

Finally, the performance overhead of App_FIT is not significant since it checks a single condition and calculates the FIT of a task through a tight code consisting of one branch and about 50 multiplication and addition instructions.

2) Evaluation of Task Replication: In this section we evaluate the overheads and scalability of selective task replication. In the experiments we replicate all tasks in an application. This way, if complete task replication (having high resource cost, more than %100) is shown to be scalable and to have low performance overheads, then it follows that selective replication (having lower resource cost) is also scalable and has low performance overheads. This is supported by the fact that the performance overheads of our heuristics are indeed very low. In addition, we use the McCalpin's artificial and memory-intensive stream benchmark [28] to stress-test our task replication design in terms of overheads and scalability. Task replicas are executed on spare cores.

Figure 4 shows the performance overheads of task replication with respect to the fault-free execution (wall-clock) times for all benchmarks. As seen, the overheads are very low and 2.5% on average.

Next we evaluate the effect of replication on the scalability of the benchmarks. We also assess the effect of fault recovery on the scalability with per task fixed fault rates. Figure 5 shows the scalability of complete task replication for shared-memory benchmarks i.e., speedups over 1 core with the given fault rate (each case has a different baseline). As seen, replication scales very well (except Stream). Stream does not scale with 16 cores even without task replication. It does not have much parallelism and mainly consists of memory operations which hinders its scalability even when there is no replication. Note that slight differences in speedups across different fault rates are due to the experimental noise (also holds for distributed applications). Figure 6 shows the scalability of complete task replication for distributed benchmarks i.e., speedups over 64 cores with the given fault rate (each case has a different baseline). We see that task replication is highly scalable for distributed applications. By evaluating these applications, we show task replication is well-suited for task-parallel distributed programs and is highly scalable at large scale and for high core counts.

Takeaway-2: *Overhead and scalability results indicate that task-parallel dataflow programming makes fault-tolerance affordable while enabling design of replication heuristics.*

VI. Related Work

Replication is a well-known technique that has been adopted in various domains from aviation to distributed
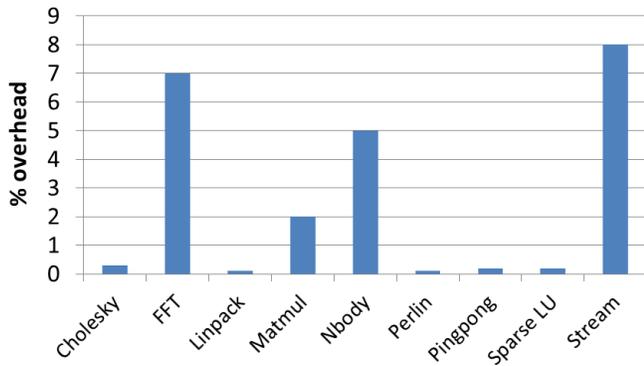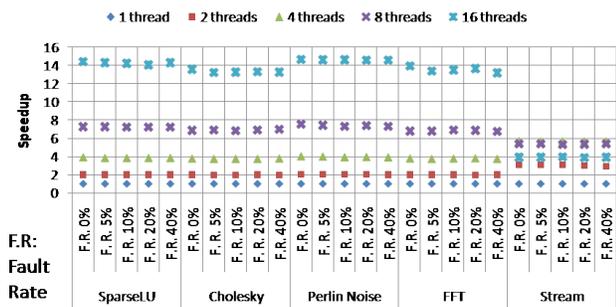
Figure 4. Task replication overheads

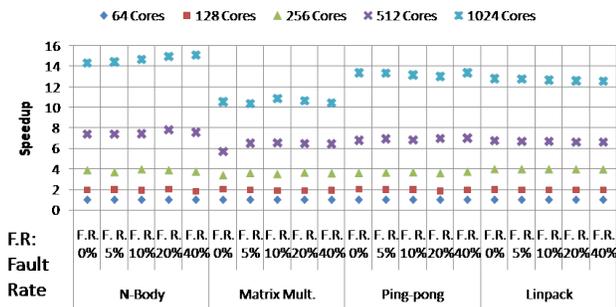Figure 5. Complete replication scalability (shared memory)

Figure 6. Complete replication scalability (distributed)

systems [33]. This technique has been used for reliability, performance and ensuring quality of service. However in most cases the complete replication of a system or an application can be prohibitively costly to achieve the intended purpose. As a result, selective replication becomes the only viable solution. For instance concerning the performance of systems, the work of Beckmann et al. [8] investigates selective replication to increase the performance of the caches of chip multiprocessors using a metric based on hit latency and misses. In case of aiming for better quality of service (QoS), Gruneberger et al. [20] propose a selective replication heuristic to increase QoS while keeping costs affordable for the distributed event-processing systems.

However selective replication as a way to address the trade-off between resource costs and reliability has not been investigated thoroughly, particularly in HPC community. Moreover, on one hand, the aforementioned studies [8, 20] cannot be employed to increase reliability while keeping cost affordable since those techniques and heuristics do not capture the reliability critical aspects of systems. On the other hand, there is the growing body of evidence showing that selective fault-tolerance support is of key-importance to decrease the resource costs while providing the required level of reliability. For instance, Luo et al. [24] and Fang et al. [16] find that different applications and different phases in applications (in our case tasks) exhibit different vulnerabilities. Although neither of these works state it explicitly, it follows that selective fault-tolerance is a natural fit to achieve a reasonable trade-off between costs and the required level of reliability for different applications. Thus, to the best of our knowledge, our selective replication heuristic is the first to address the trade-off between resource costs and application-specific reliability requirements, in particular for task-parallel HPC applications. The work of Subasi et al. [38] is based on programmer knowledge in order to achieve effective partial replication. The NanoCheckpoints [36] and the message logging protocol proposed by Martsinkevich et al. [27] address fail-stop errors of task-parallel computations. SSD [37] is designed by using machine learning techniques to mitigate silent errors in HPC applications.

Meanwhile even though the performance and efficiency advantages of task-based dataflow programming models [19] and runtimes [41] are well-established, to the best of our knowledge, there has been no research investigating selective replication in these programming models. As consequence, we strive to leverage the resilience advantages of such frameworks to develop fully automatic runtime selective task replication heuristics. Although our selective task replication framework does not provide complete failure coverage such as errors from the operating system or network/MPI communications, it is orthogonal and seamlessly integrable to system-wide fault-tolerance solutions such as [7, 18].

There has been work at compiler level to selectively duplicate instructions that may cause user-visible errors [17, 22]. Shoestring [17] uses the data flow and control flow graphs of programs to select vulnerable instructions for error detection by duplication. Laguna et al. use machine learning to learn code instructions that must be protected to avoid corruptions. They use compiler to protect only those learned vulnerable instructions through duplication. As opposed to our scheme, these techniques do not offer error recovery.

VII. CONCLUSION

In this study we propose low-overhead and scalable selective task replication for task-parallel programs to mitigate SDCs and DUEs. To achieve selective task replication, we develop a selective task replication technique for task-parallel programs. We present our automatic and transparent heuristic to select the tasks to replicate for keeping FIT of

a program under a given threshold. Results show that it has low overhead and selects tasks in an efficient way.

In this research, our key findings were: First, complete replication of HPC applications is not required to mitigate the foreseen exascale error rates while achieving the same reliability levels today which are typically sufficient for the applications to correctly finish their computations. Second, task-parallelism and dataflow offer key properties making fault-tolerance support for future HPC systems affordable.

VIII. Acknowledgments

References

[1] BSC Application Repository: https://pm.bsc.es/projects/bar/wiki/applications.

[2] Intel tbb 4.3 update 2. http://www.threadingbuildingblocks.org/documentation.

[3] Marenostrum III: http://www.bsc.es/marenostrum-support-services/mn3.

[4] A. Amer, N. Maruyama, M. Perics, K. Taura, R. Yokota, and S. Matsuoka. Fork-join and data-driven execution models on multi-core architectures: Case study of the FMM. In *Supercomputing*, volume 7905 of *Lecture Notes in Computer Science*, pages 255–266. Springer Berlin Heidelberg, 2013.

[5] M. Andersch, C. C. Chi, and B. Juurlink. Using OpenMP superscalar for parallelization of embedded and consumer applications. In *International Conference on Embedded Computer Systems (SAMOS)*, pages 23–32, 2012.

[6] M. Andersch, B. Juurlink, and C. C. Chi. A benchmark suite for evaluating parallel programming models. Workshop on Parallel Systems and Algorithms, pages 7–17, 2011.

[7] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. Fti: high performance fault tolerance interface for hybrid systems. In *SC 2011*, pages 32:1–32:32.

[8] B. M. Beckmann, M. R. Marty, and D. A. Wood. Asr: Adaptive selective replication for cmp caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 443–454, Washington, DC, USA, 2006. IEEE Computer Society.

[9] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *Int. J. High Perform. Comput. Appl.*, 23(4):374–388, Nov. 2009.

[10] D. Chasapis, M. Casas, M. Moretó, R. Vidal, E. Ayguadé, J. Labarta, and M. Valero. Parsecss: Evaluating the impact of task parallelism in the parsec benchmark suite. volume 12, pages 41:1–41:22, New York, NY, USA, Dec. 2015. ACM.

[11] M. E. Conway. A multiprocessor system design. In *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, AFIPS '63 (Fall), pages 139–146, New York, NY, USA, 1963. ACM.

[12] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan 1998.

[13] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, and B. et al. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, Feb. 2011.

[14] A. Duran, E. Ayguade, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.

[15] M. Ebrahimi, H. Asadi, and M. B. Tahoori. A layout-based approach for multiple event transient analysis. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 100:1–100:6, New York, NY, USA, 2013. ACM.

[16] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. Evaluating the error resilience of parallel programs. In *The 4th Fault Tolerance for HPC at eXtreme Scale Workshop (FTXS)*, 2014.

[17] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 385–396, New York, NY, USA, 2010. ACM.

[18] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 78:1–78:12, 2012.

[19] P. Ghosh, Y. Yan, D. Eachempati, and B. Chapman. A prototype implementation of openmp task dependency support. In A. P. Rendell, B. M. Chapman, and M. S. Muller, editors, *OpenMP in the Era of Low Power Devices and Accelerators: 9th International Workshop on OpenMP, IWOMP 2013, Canberra, ACT, Australia, September 16-18, 2013. Proceedings*, pages 128–140, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[20] F. J. Grneberger, T. Heinze, and P. Felber. Adaptive selective replication for complex event processing systems. In *BD3@VLDB*, volume 1018 of *CEUR Workshop Proceedings*, pages 31–36. CEUR-WS.org, 2013.

[21] J. L. Kelly, C. Lochbaum, and V. Vyssotsky. A block

diagram compiler. *The Bell System Technical Journal*, 1961.

[22] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson. Ipas: Intelligent protection against silent output corruption in scientific applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO 2016, pages 227–238, New York, NY, USA, 2016. ACM.

[23] K. M. Lepak and M. H. Lipasti. Silent stores for free. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 22–31, New York, NY, USA, 2000. ACM.

[24] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu. Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 467–478, Washington, DC, USA, 2014. IEEE Computer Society.

[25] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero. Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 5–16, 2010.

[26] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., 1990.

[27] T. V. Martsinkevich, O. Subasi, O. S. Unsal, F. Cappello, and J. Labarta. Fault-tolerant protocol for hybrid task-parallel message-passing applications. In *2015 IEEE International Conference on Cluster Computing, CLUSTER, 2015, Chicago, IL, USA, September 8-11, 2015*, pages 563–570, 2015.

[28] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *TCCA Newsletter*, 1995.

[29] S. E. Michalak, A. J. DuBois, C. B. Storlie, H. M. Quinn, W. N. Rust, D. H. DuBois, D. G. Modl, A. Manuzzato, and S. P. Blanchard. Assessment of the impact of cosmic-ray-induced neutrons on hardware in the roadrunner supercomputer. *IEEE Transactions on Device and Materials Reliability*, 12(2):445–454, June 2012.

[30] OpenMP Architecture Review Board. OpenMP api version 4.0, 2013.

[31] B. Panzer-Steindel. Data integrity. In *CERN/IT Draft 1.3*, 2007.

[32] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, VECPAR'10, pages 1–25, Berlin, Heidelberg, 2011. Springer-Verlag.

[33] D. Siewiorek and R. Swarz. *Reliable Computer Systems: Design and Evaluation*. 1998.

[34] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, and F. Cappello et al. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, 28(2), 2014.

[35] V. Sridharan and D. Liberty. A study of dram failures in the field. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 76:1–76:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[36] O. Subasi, J. Arias, O. Unsal, J. Labarta, and A. Cristal. Nanocheckpoints: A task-based asynchronous dataflow framework for efficient and scalable checkpoint/restart. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 99–102, March 2015.

[37] O. Subasi, S. Di, L. Bautista-Gomez, P. Balaprakash, O. S. Ünsal, J. Labarta, A. Cristal, and F. Cappello. Spatial support vector regression to detect silent errors in the exascale era. In *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016*, pages 413–424, 2016.

[38] O. Subasi, J. A. Moreno, O. S. Unsal, J. Labarta, and A. Cristal. Programmer-directed partial redundancy for resilient HPC. In *Proceedings of the 12th ACM International Conference on Computing Frontiers, CF'15, Ischia, Italy, May 18-21, 2015*, pages 47:1–47:2, 2015.

[39] V. Subotic, S. Brinkmann, V. Marjanovic, R. M. Badia, J. Gracia, C. Niethammer, E. Ayguade, J. Labarta, and M. Valero. Programmability and portability for exascale: Top down programming methodology and tools with StarSs. *J. Comput. Science*, 4(6):450–456, 2013.

[40] X. Teruel, X. Martorell, A. Duran, R. Ferrer, and E. Ayguadé. Support for OpenMP tasks in nanos v4. In *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, pages 256–259, 2007.

[41] 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 *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 301–302, New York, NY, USA, 2012. ACM.

[42] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 264–275, June 2004.