Reliability for Exascale Computing: System Modelling and Error Mitigation for Task-parallel HPC Applications

Omer Subasi

Department of Computer Architecture

Universitat Politècnica de Catalunya

A dissertation submitted in fulfilment of
the requirements for the degree of

Doctor of Philosophy / Doctor per la UPC

October 2016

Advisor : Jesus Labarta
Coadvisor: Osman Unsal

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola de Doctorat

Assessment results for the doctoral thesis

Academic year:

Full name

Doctoral programme

Structural unit in charge of the programme

Decision of the committee

In a meeting with the examination committee convened for this purpose, the doctoral candidate presented the topic of his/her doctoral thesis entitled

_____

_____.

Once the candidate had defended the thesis and answered the questions put to him/her, the examiners decided to award a mark of:

☐ UNSATISFACTORY ☐ SATISFACTORY ☐ GOOD ☐ VERY GOOD

(Full name and signature) Chairperson	(Full name and signature) Secretary

(Full name and signature) Member	(Full name and signature) Member	(Full name and signature) Member

_____, _____

The votes of the members of the examination committee were counted by the Doctoral School at the behest of the Doctoral Studies Committee of the UPC, and the result is to award the CUM LAUDE DISTINCTION:

☐ YES ☐ NO

(Full name and signature) Chair of the Standing Committee of the Doctoral School	(Full name and signature) Secretary of the Standing Committee of the Doctoral School

Barcelona, _____

International doctoral degree statement

As the secretary of the examination committee, I hereby state that the thesis was partly (at least the abstract and the conclusions) defended in a language commonly used in scientific communication in the field that is not an official language of Spain. This does not apply if the stay, report or expert is from a Spanish-speaking country.

(Nom, cognoms i signatura) Secretary of the Examination Commitee

This dissertation is dedicated to my father.
Always will be missed.

Acknowledgements

My advisors Jesus Labarta, Osman Unsal and Adrián Cristal gave me the opportunity for PhD studies. Their support, confidence, technical advise, and help have played a major role shaping my research ideas into the contributions expressed in this dissertation. I am very grateful to and thank Jesus, Osman and Adrián for their generosity and comprehension which helped me to go through the difficulties during this four-year long study.

I would like to thank Franck Cappello, who kindly invited me to a three-month internship at Argonne National Lab in the United States. I had a great and productive time at Argonne Lab thanks to Franck's always positive attitude and enthusiasm. At Argonne, I also benefited from a great working environment and met a number of colleagues that made my stay even more enjoyable. Sheng, Leonardo, Prasanna – thank you all for the helpful moments and interesting conversations.

I would also like to acknowledge all my friends and colleagues from the office that helped me throughout my PhD; for their insights and expertise in technical matters, and for their unconditional support. Many thanks go to Erdal Mutlu, Burcu Mutlu, Ferad Zyulkyarov, Gülay Yalçın, Oscar Palomar, Javier Arias, Oyku Melikoglu, Ismail Kuru and many others. I sincerely thank you all for your help and all the great moments we have had together.

I would like to thank my family for supporting me during this endeavour. My deepest thanks to my father, whom I lost in 2015, who was always supportive of me.

My graduate work has been supported by FI-DGR AGAUR 2013 scholarship, HiPEAC PhD Collaboration Grant, funding from Barcelona Supercomputing Center, the European Community's Seventh Framework Programme [FP7/2007-2013] under the Mont-blanc 2 Project, grant agreement no. 610402, and TIN2015-65316-P Project.

Abstract

As high performance computing (HPC) systems continue to grow, their fault rate increases. Applications running on these systems have to deal with rates on the order of hours or days. Furthermore, some studies for future Exascale systems predict the rates to be on the order of minutes. As a result, efficient fault tolerance solutions are needed to be able to tolerate frequent failures.

A fault tolerance solution for future HPC and Exascale systems must be low-cost, efficient and highly scalable. It should have low overhead in fault-free execution and provide fast restart because long-running applications are expected to experience many faults during the execution.

Meanwhile task-based dataflow parallel programming models (PM) are becoming a popular paradigm in HPC applications at large scale. For instance, we see the adaptation of task-based dataflow parallelism in OpenMP 4.0, OmpSs PM, Argobots and Intel Threading Building Blocks.

In this thesis we propose fault-tolerance solutions for task-parallel dataflow HPC applications. Specifically, first we design and implement a checkpoint/restart and message-logging framework to recover from errors. We then develop performance models to investigate the benefits of our task-level frameworks when integrated with system-wide checkpointing. Moreover, we design and implement selective task replication mechanisms to detect and recover from silent data corruptions in task-parallel dataflow HPC applications. Finally, we introduce a runtime-based coding scheme to detect and recover from memory errors in these applications. Considering the span of all of our schemes, we see that they provide a fairly high failure coverage where both computation and memory is protected against errors.

Contents

List of Figures

List of Tables

1 Introduction

As we get closer to the Exascale time-frame, fault-tolerance is becoming one of the main concerns along with power and energy consumption in High Performance Computing (HPC). The mean time between errors is expected to be on the orders of hours or even minutes [1] for future HPC and exascale systems. Moreover the extent and impact of Exascale failure types such as silent data corruptions (SDCs) or silent errors could present a significant threat for large-scale application reliability [2]: Such errors are not detected and applications produce wrong results. Another type of errors that is common in HPC systems is the fail-stop errors: The computing process/unit aborts execution and loses all computation and data. Hardware-only solutions are not expected to handle these two types of errors at Exascale error rates [3]. Thus software based fault-tolerance solutions are needed to complement the hardware based ones.

Meanwhile task-based and data-driven programming models (PM) are becoming widely used in HPC community [4]. As an example we see the adaptation of task-based dataflow parallelism in OpenMP 4.0 [5] and Intel Threading Blocks [6]. The performance of dataflow parallelism is shown to be higher than that of the traditional fork-join model due to amount of parallelism achieved [7]. Moreover, these models provide unique advantages in terms of fault

tolerance: In task-based dataflow programs, the failure containment is better than traditional parallel programs (for example those implemented with Pthreads) since an error is most likely to cause only the failure of the single task in which it occurs rather than the entire application. Furthermore, even if the error propagates to other tasks, having a dataflow model facilitates tracing the error through dependent tasks to its root cause. Additionally, the asynchronous nature of dataflow computation enables low-overhead error recovery by overlapping recovery of faulty task(s) with execution of other tasks that are independent of the faulty task(s). Therefore we find that it is important to provide fault-tolerance support for task-parallel data-driven HPC applications using data-driven execution model to mitigate fail-stop errors and SDCs.

In this thesis, we present fault-tolerance mechanisms to detect and recover errors that occur during the computations of task-parallel dataflow HPC applications.

1.1 Problem Statement and Contributions of Thesis

This thesis proposes several mechanisms for reliability of task-parallel dataflow HPC applications which we briefly list below. In addition, we present the contributions of this thesis.

1.1.1 Checkpointing and Message-logging for Task-parallel Dataflow HPC Applications

Checkpoint/restart is a well-known technique to mitigate fail-stop errors. However with the expected exascale error rates [1], this technique is projected to be not viable. Novel mechanisms are needed to handle fail-stop errors. In particular, with the introduction of new programming models (PMs), such as task-based dataflow PMs, there is no known mechanisms that can integrate task-level checkpoints and system-wide checkpoints.

Moreover hybrid PMs that combine message passing and task-parallelism to overlap computation and communication have emerged. There are no known fault-tolerance solutions for applications programmed in these new PMs.

In this thesis, we propose NanoCheckpoints which is a task-level checkpoint/restart framework for fail-stop errors that occur during the executions of task-parallel dataflow HPC applications. We show that our scheme has low overhead and is highly scalable at high core counts and large scale. Moreover, we develop a unified mathematical model to optimize the checkpoint interval of system-wide checkpointing when task-level checkpointing and system-wide

checkpointing are integrated to provide complete failure coverage. Furthermore, we extend NanoCheckpoints with a message logging protocol to provide reliability for hybrid distributed task-parallel programs. Finally, we develop a formal performance model for these hybrid PMs that shows the merits of our framework and protocol.

1.1.2 Selective Task Replication in Task-parallel Dataflow HPC Applications

The straightforward way to cope with SDCs or silent errors is to replicate all tasks in a task-parallel application. However this might be too costly or even not needed. That is, the complete task redundancy is only needed for very high error coverage. Smart mechanisms are needed for partial/selective replication of tasks. To the best of our knowledge, in the context of task-parallel dataflow programs, no such mechanisms exist which can trade-off between reliability and performance/power/energy costs.

In this thesis, we propose four different runtime-based selective task replication mechanisms that target both fail-stop errors and SDCs. The first one is a completely automatic, transparent-to-user, risk-based and lightweight heuristic. The second one incorporates programmer knowledge. Finally, the third and fourth ones are developed in a formal way and are also completely automatic and transparent to user. Experiments show that all four schemes are low-overhead, highly scalable and effective in SDC detection.

1.1.3 Memory Reliability for Task-parallel Dataflow HPC Applications

Memory reliability will be one of the major concerns for future HPC and Exascale systems. This concern is mostly attributed to the expected massive increase in memory capacity and the number of memory devices in Exascale systems. For memory systems Error Correcting Codes (ECCs) are the most commonly used mechanism. However state-of-the art hardware ECCs will not be sufficient in terms of error coverage for future computing systems in the Exascale era. Stronger hardware ECCs providing more coverage have prohibitive costs in terms of area, power and latency. Software-based solutions are needed to cooperate with hardware.

In this thesis, we propose a Cyclic Redundancy Checks (CRCs) based software mechanism for task-parallel HPC applications. Experiments that our mechanism incurs very low performance overhead with hardware acceleration while being highly scalable at large scale.

Our mathematical analysis demonstrates the effectiveness of our scheme and its error coverage. In addition, results show that our CRC-based mechanism reduces memory vulnerability significantly.

1.2 Thesis Organization

Chapter 2 presents background information on basic terminology, task-based programming and memory reliability.
Chapter 3 discusses the state of the art for fault-tolerance in HPC systems.
Chapter 4 introduces NanoCheckpoints, our checkpoint/restart framework for task-parallel dataflow HPC applications.
Chapter 5 presents our mathematical model which unifies task-level checkpointing with system-wide checkpointing. It additionally show the benefits of this unification experimentally.
Chapter 6 presents our message logging protocol to handle errors that occur in distributed task-parallel applications. Moreover, it presents our unified model for the integration of our protocol with system-wide checkpointing.
Chapter 7 discusses our selective task replication mechanisms and their in-depth evaluation.
Chapter 8 presents our cyclic redundancy checks based memory reliability scheme for task-parallel HPC application, and its mathematical and experimental evaluation.
Chapter 9 concludes this dissertation.

2 Background

In this chapter we discuss the relevant concepts and topics to our research which will help to understand the overall research and contributions of the thesis. We first introduce the error classification and failure model (Section 2.1). Then we discuss task-parallel dataflow programming and its advantages favoring fault-tolerance (Section 2.2). After that, we present OmpSs programming model (PM) and its runtime (Section 2.3). Finally, we present background on memory reliability and cyclic redundancy checks (Section 2.4).

2.1 Error Classification and Failure Model

Throughout this thesis, 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 [8, 9] 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 impossible for the faulted processor/hardware to recover [10]. Because of this, we consider DUEs as fail-stop errors. 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 [2, 11]. 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 [12]. Thus in this thesis we target SDCs and DUEs i.e. fail-stop errors.

2.2 Task-parallel Dataflow Programming

Tasks, as a higher level abstraction, provide a more convenient and friendly interface for expressing parallelism in parallel programs [4, 13]. Depending on how tasks are scheduled for execution and how they interleave during execution, task-based parallel programming can be either fork-join parallelism [14] or dataflow parallelism [15]. 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 [16]. One can find example implementation of fork-join tasks in OpenMP 3.0 [17] and dataflow tasks in OmpSs [18], OpenMP 4.0 [19] and Intel TBB [6].

A recent comparison between dataflow and fork-join parallelism by Amer et al. [7] 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 2.1. Figure 2.1 (a) is an example dataflow code in OmpSs 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 2.1 (b) is the same example but for fork-join in OpenMP 3.0. The difference between Figure 2.1 (a) and (b) is that the fork-join tasks do not declare explicitly their

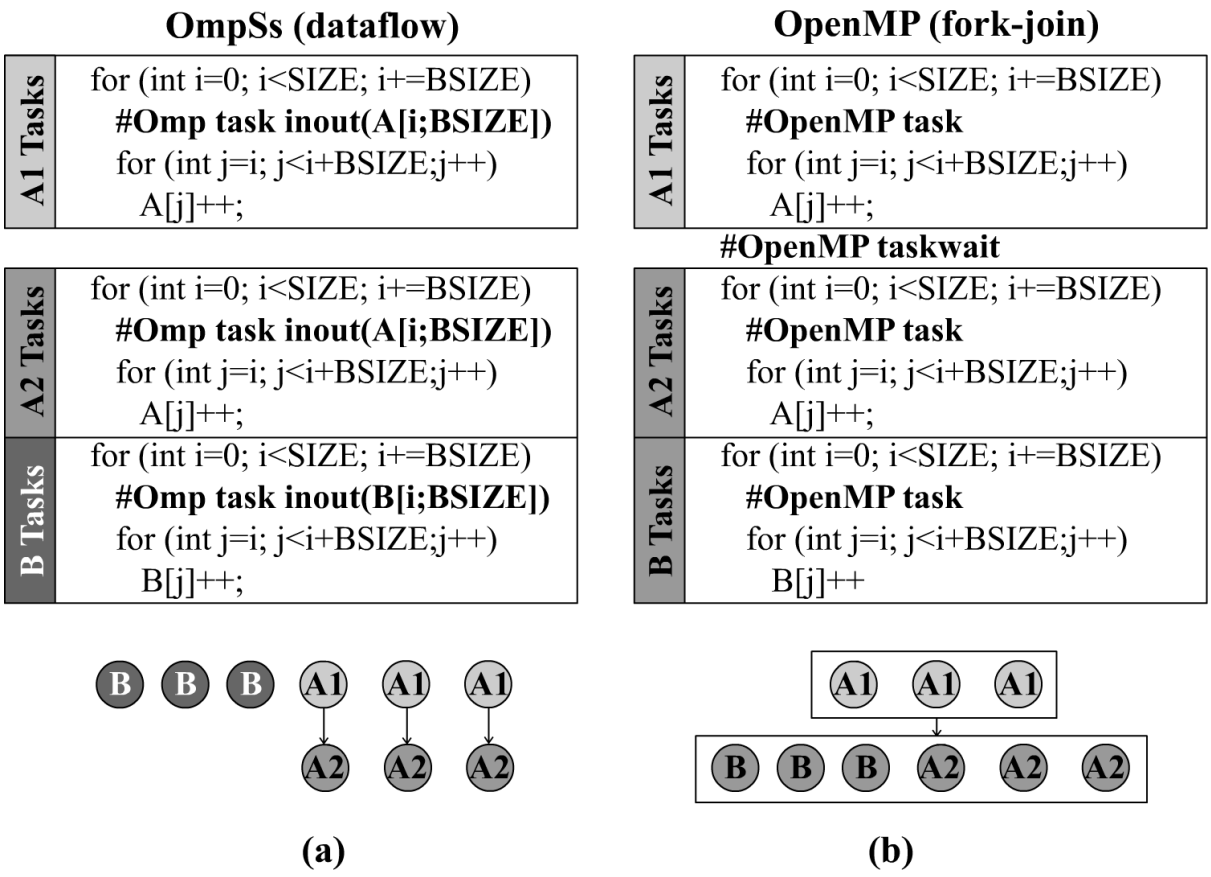


Figure 2.1: Example code in Dataflow and Fork-join.

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.

Furthermore, task-based dataflow programming has important inherent properties that favor

fault-tolerance:

1. Better failure containment: Tasks provide better failure containment since they contain failures within their computations rather than the entire application. The restart of the entire application is mostly avoided this way.

2. Asynchronous execution and overlap of computation and recovery: The asynchronous nature of dataflow execution enables low-overhead error recovery by overlapping recovery of faulty task(s) with execution of other tasks that are independent of the faulty task(s).

3. Minimal checkpoint size: As stated above, task inputs are available through programmer annotations which represent the minimal state of the tasks needed to be checkpointed for the fault recovery. In contrast, solutions in OpenMP typically checkpoint the entire address space of the application since the runtime is unaware of the exact application state to protect.

4. Easiness of SDC detection: Since the error propagation is only possible through the task outputs, the detection of SDC can be easily done by the comparison of the outputs. There is no need for an additional SDC detection mechanism such as the work of Di et al. [20] which converts SDC detection into a one-step look-ahead prediction problem.

5. Enabler for heuristics: With this knowledge of the task inputs and outputs at runtime, we are able to design and implement very efficient and low-overhead selective task replication heuristics.

2.3 OmpSs PM and Nanos Runtime

In this section we review the pure OmpSs PM and its runtime, and elaborate on the hybrid OmpSs+MPI programming.

2.3.1 OmpSs and Nanos

We now give a short introduction to the OmpSs programming model and the Nanos [21] dataflow runtime. OmpSs is a task-based parallel programming model derived from OpenMP. It extends OpenMP tasking constructs with a new data directionality (dependency) clauses to support asynchronous parallelism and heterogeneity, i.e. to support devices such as GPUs. OmpSs applications utilize source-to-source Mercurium [22] compiler and the Nanos runtime.

OmpSs uses a thread-pool execution model instead of the traditional OpenMP fork-join model. The master thread starts the execution and all other threads cooperate to complete the

work it generates. OmpSs has a memory model where multiple address spaces may exist, for example, due to the use of heterogeneous kernels. This means that shared data may reside in memory locations that are not directly accessible from other computational resources. Thus, the code can only safely access private data and the shared data that has been explicitly marked as such with the help of the extended OmpSs syntax.

OmpSs programs are parallelized by declaring and instantiating tasks. Tasks are pieces of code that can be executed on parallel resources. When the programmer taskifies an application, i.e. defines source code segments as tasks, he specifies task inputs and outputs to express the data dependencies. Based on this information, Nanos dynamically generates a task dependency graph of the program. When the program reaches a section of code declared as a task, an instance of the task is created and its execution is delegated to Nanos. A task is created in a waiting state, but, once all its input dependencies are resolved, it becomes ready and is executed by a thread from the thread pool as soon as there is an available core. After it finishes, the dependency graph is updated so that its dependent tasks become ready for execution. Figure 2.2 shows a sample program code and its task dependency graph generated at runtime.

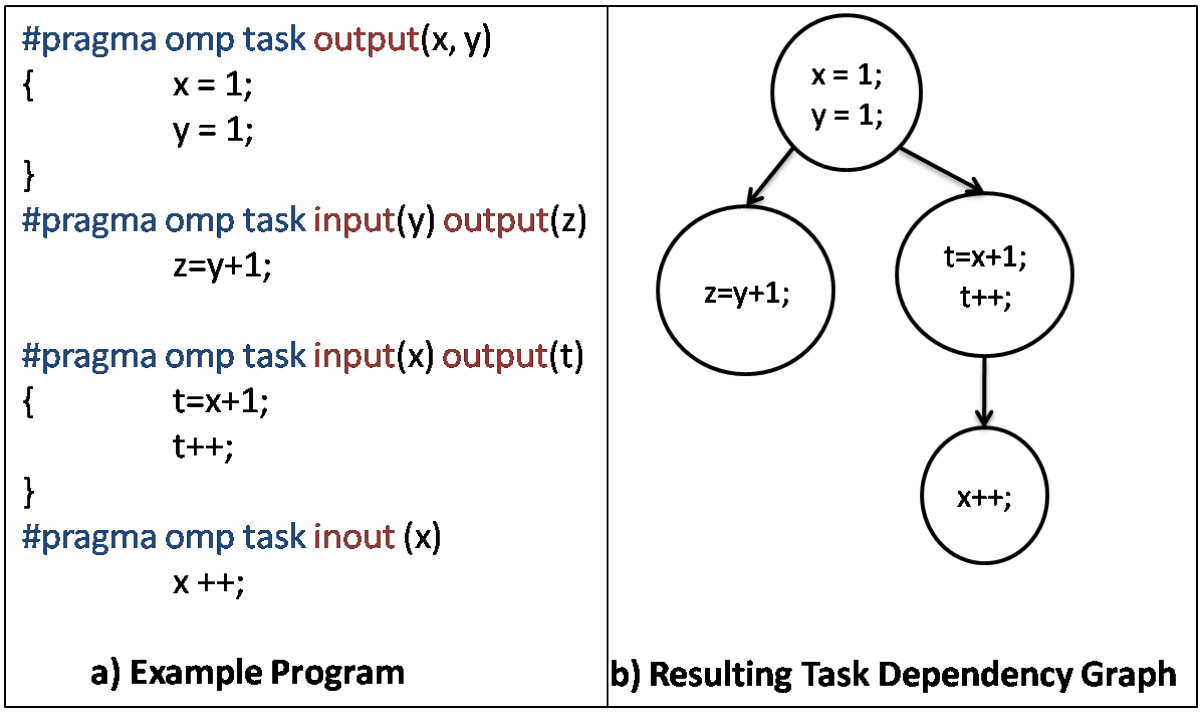


Figure 2.2: Sample OmpSs code and its task dependency graph

The main properties of the OmpSs PM are asynchronous parallelism and dataflow-based execution, because the order of task executions only follows the dependency graph but can be out of order from the point of view of the program.

Taskifying an application with OmpSs pragmas is easy because the programmer only needs to express data dependencies and/or target heterogeneous devices where particular tasks should be executed leaving everything else to the runtime. This greately facilitates his work and shifts the burden of parallelization from the programmer to the runtime. As experiments showed [23], [24], and [25], the performance of OmpSs and Nanos is on par or superior to other programming models and their associated runtimes because it exploits implicit and irregular parallelism.

We refer the reader to [18] for further details on the OmpSs PM.

2.3.2 The OmpSs+MPI hybrid programming model

Pure OmpSs PM targets shared-memory machines but it can be combined with MPI to allow execution on distributed memory systems. In such a hybrid model the work is distributed between the MPI processes and then parallelized inside each process by means of OmpSs. This approach is similar to the hybrid OpenMP+MPI model and it mixes a dataflow execution with the message passing model providing some performance benefits. In particular, it helps to hide the communication latencies and achieve higher Gflops/s performance as it was shown in [26].

In an OmpSs+MPI application, the programmer can taskify parts of code that include communication calls. This creates an opportunity to overlap communication and computation because such tasks can run in parallel with other non-dependant tasks. If a task includes an MPI call the programmer must list the buffer used for the call among the task input or output parameters so that the runtime knows which tasks depend on communication.

If a task has stalled waiting for completion of a blocking MPI call, the runtime can preempt such task and free the core for other tasks ready for execution. This helps to utilize resources more efficiently.

Figure 2.3 is a simplified code example taken from a matrix multiplication application that we used as one of our benchmarks in this paper. Each process multiplies blocks of a matrix and exchanges results with other processes. Figure 2.4 (a) shows what part of the code was taskified, and Figure 2.4 (b) schematically shows the overlapping of tasks in one process. The red nodes represent SendRecv tasks with MPI communication calls inside and the white nodes represent the mxm computation tasks. The parallelograms demonstrate the communication and

```
i = n*my_rank; //first C block

for( it = 0; it < nodes; it++ ) {

  mxm((double *)A, (double *)B, (double *)&C[i][0]);

  if (it < nodes-1) {
     SendRecv((double *) A, (double *)receivebuf);
  }

  i = (i+n)%m;   //next C block
  //swap pointers
  ptmp=a; a=receivebuf; receivebuf=ptmp;
     ...
}
```

Figure 2.3: Sample OmpSs/MPI code

computation tasks' overlap.

2.4 HPC Memory Reliability

We now discuss memory reliability in HPC systems and cyclic redundancy checks.

2.4.1 Background

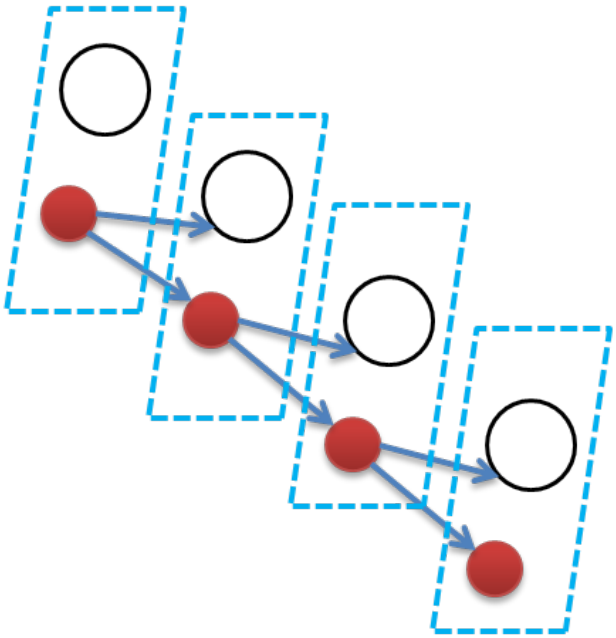
With shrinking feature sizes, error rates in main memory are expected to increase [27]. Moreover in future HPC systems the overall memory capacity and the number of memory devices will increase which will lead to higher error rates.

Memory in current HPC systems is mostly protected by hardware ECCs. Single bit-error correction and double bit-error detection (SECDED) [28] and Chipkill [29] are the most widely used ECCs. Chipkill [29] is used to mitigate DRAM chip failures. Considering future HPC and Exascale systems, on one hand Sridharan et al. [30] report that Chipkill will not be sufficient to mitigate the projected Exascale error rates. On the other hand, stronger ECCs, such as BCH [31], in hardware will incur prohibitive overheads given the predicted fault rates [32]. For BCH codes, Strukov et al. [33] show that the area (storage) overheads are linearly proportional to the length of code, i.e. data plus redundant information, and to the number of errors that can be detected and corrected. Moreover the latency or the performance delay is linearly propor-

```
#pragma omp task input(A, B) inout(C)
void mxm (double A[N], double B[N],double C[N]);

#pragma omp task input(A) output(receivebuf)
void SendRecv (double A[N],double receivebuf[N]);
```

(a)



(b)

Figure 2.4: a) Tasking MPI calls and b) Task dependency graph and overlap of communication with computation

tional to the number of errors that can be corrected. Furthermore, BCH codes have expensive encoding and decoding algorithms. These overheads will be prohibitive especially when the ECC support is not aware of the minimal, precise application data to be protected.

2.4.2 Cyclic Redundancy Checks

Cyclic Redundancy Checks [34] are cyclic codes that are typically used for error detection only. They are mostly used in communication, digital and mobile networks, and aviation. These codes are well-suited for burst errors, i.e. contiguous sequence of errors in data, over a digital network. The CRC computation requires a pre-specified polynomial, called generator polynomial, which becomes the divisor and the data becomes dividend which is considered to be a polynomial over Galois Field GF(2). The remainder of the polynomial division becomes the check value, also called CRC. A CRC is called an n-bit CRC, denoted by CRC-n, when its check value is n-bits. When a generator polynomial with degree n is used, the remainder will have length n since the polynomial has n+1 length. If generator polynomial has a degree of n, then all burst errors no longer than n bits will be detected for arbitrary data length. However the fault detection capability of a CRC depends on Hamming Distance (HD) achieved with the selected generator polynomial as well as the size of data it protects. For instance the 32-bit CRC polynomial standardized for Ethernet has a HD of 4 for data of size Ethernet Maximum Transmission Unit (MTU), i.e. maximum size of data that a protocol can pass onwards (Ethernet MTU is 1500 bytes). There had been research by exhaustively searching the optimal CRC polynomials i.e. those providing highest HD, for different data sizes [35] [36]. Therefore, one should use different CRC polynomials for different applications to provide flexibility to the user to adapt the CRC-polynomial according to the sizes of task arguments for an application to achieve the best fault detection capability, i.e. largest HD.

3

The State of the Art

In this chapter we present related work to our research in this thesis. We first discuss the state of the art in checkpoint/restart systems (Section 3.1). Then we present related work in checkpointing and performance modeling in HPC systems (Section 3.2). We then move onto discussing replication which is a well-known technique in fault-tolerance literature (Section 3.3). In Section 3.4, we present the state-of-the-art on silent data corruptions or silent errors which are the type of errors that we aim to mitigate in this thesis. Finally, we close related work by discussing the literature on memory reliability (Section 3.5).

3.1 Checkpointing Systems

Checkpoint/restart, or checkpointing, is a well-known *rollback recovery* technique deployed to mitigate fail-stop errors [37]. In checkpointing the application state, called checkpoint, is saved and the application is restarted by using the checkpoint when a failure occurs. The most common types of checkpointing are coordinated and uncoordinated checkpointing for distributed applications. In coordinated checkpointing, processes coordinate with each other to

reach a consistent global state before taking a checkpoint. The main drawback of coordinated checkpointing is the expensive coordination. Uncoordinated checkpointing was proposed to avoid the expensive coordination [37]. In uncoordinated checkpointing, the states of processes are saved independently, which avoids the coordination but supplementary application data is also logged such that a consistent global state could be formed in a restart. The logging of supplementary data is called message logging which is a technique used to prevent domino effect [37]. Domino effect is a condition in which, in the lack of message logging, the restart of a process causes other processes to restart; and in the end all processes have to restart losing all useful computation. Uncoordinated checkpointing has its own disadvantages. The log size, communication overhead, and fault-free execution overheads can be prohibitive.

To address these issues, hierarchical coordinated checkpointing schemes are proposed such as [38] and [39]. These schemes partition the application processes into groups in which each group can checkpoint independently. They limit the coordination among the processes in a single group and they perform costly global checkpoints less frequently than those within each group. However message logging is performed across the groups to be able for a global restart. Moreover the coordination among the processes belonging to the same group can still be expensive.

Basic checkpointing and message logging algorithms often suffer from low scalability and large memory footprint. Considerable work has been done to address these issues [40]. Some researchers suggest considering the knowledge on correlation between hardware faults in order to lower the message log memory overhead [41], [42]; others combine coordinated checkpointing with message logging for the same purpose [43]. To the best of our knowledge, however, no prior work has tackled message logging for hybrid PMs.

The Berkeley Lab Checkpoint/Restart [44] is a widely adopted approach in high performance computing for fault-tolerance. As a system-level solution, the most important drawback of this mechanism is the large memory footprint that is generated during checkpointing process due to the lack of knowledge of user-level structures. This mechanism provides a binary checkpoint solution that increases dependency with operating system, it generates the checkpoint using the I/O interaction at Linux kernel level. BLCR needs an operating system modification for a good performance. Our proposed solution in Chapter 4 is runtime-level, which does not create OS dependency and decreases the usage of memory.

Application-level approaches exist such as [45], however such solutions are not generic and have to be adapted for each single application with developer/programmer knowledge.

These approaches are more efficient than system-level ones. Our novel solution (Chapter 4) has the knowledge of the explicit asynchronous execution flow, which enables us to build a more accurate non-centralized incremental checkpoint and to share information through all the active threads.

System-level solutions have recently evolved into multi-level nature with the main idea to be performing expensive global checkpoints less frequently. Sato et. al. [39] proposes an approach based on multi-level checkpointing for petascale and exascale systems. Their checkpointing system merges non-blocking and multi-level checkpointing. This system is shown to be efficient on future post-petascale systems. Another hybrid solution proposed for hybrid petascale systems is the Fault Tolerance Interface by Bautista-Gomez et. al [38]. Our proposed framework is complementary to these system-wide solutions (Chapter 4).

Ma and Krishnamoorthy [46] propose an approach for task-parallel computations in the presence of work-stealing. The main objective of the work is to identify tasks to be recovered and to recover partial data updates accurately and efficiently. They track the communication operations and maintain an idempotent data store for this purpose. They target fault tolerance within a task parallel phase complementary to checkpointing and fault detection which constitute some parts of our work (Chapter 4). Chung et al. [47] introduce the Containment Domains (CD) for system resiliency. CDs are programming constructs for the programmer to express resilience concerns as well as to specialize fault detection and recovery according to application under consideration. In addition, CDs provides flexibility in terms of state preservation, location of preservation as well as the method of fault detection.

Algorithm-Based Fault Tolerance [48] requires additional programmer effort for generating specific redundant calculation on the algorithm.

Bronevetsky et al. [49] proposed an application-level checkpointing technique for OpenMP applications. The application state is regularly stored to disks and in case of a failure it is loaded and application is restarted. Fu and Yang [50] explore thread-level redundancy to detect and recover from transient errors. Tahan and Shawky [51] propose triple modular redundancy fault-tolerance for OpenMP programs via task replication at compiler level.

3.2 Modeling Checkpointing Systems

There has been a significant body of work to model checkpointing systems in order to find the optimal checkpoint interval to minimize the total waste. Young and Daly study the sequential

jobs [52] [53]. For parallel jobs, studies such as [54], [55] and [56] model the coordinated checkpointing protocols.

Considering hierarchical and partially coordinated checkpointing systems, the studies of Di et al. [57] and [58] investigate how to model and analytically determine the optimal checkpoint interval of each level in the hierarchy. In these studies Di et al. characterize the overheads of a multilevel scheme under analytically found checkpoint intervals. Balaprakash et al. [59] address the question of optimizing the checkpoint intervals while also considering the energy consumption of a multilevel scheme. As with the models for the coordinated schemes [54], [55] and [56], these hierarchical models cannot be used for non-hierarchical unified uncoordinated and coordinated checkpointing systems such as combined task-level and system-wide checkpointing. The unified model proposed by Bosilca et al. [60] models a range of checkpointing systems from fully coordinated scheme on one extreme to partially coordinated hierarchical schemes on the other extreme. However their model does not fit to the case of the non-hierarchical uncoordinated and coordinated unification since for the uncoordinated task-level checkpointing, a periodic checkpointing interval cannot be imposed. The data-driven and task-parallel execution model prevents periodicity which in fact turns out to be an enabler for affordable overall fault tolerance. Moreover, their assumption of tightly-coupled applications is not needed for our unified model (Chapter 5) and is not the case for a task-parallel application. If one of the computing resources, such as a processor, fails, then a task can simply be rescheduled to another processor whereas a tightly-coupled application would have to recover and restart. As a result, our mathematical framework [61] (Chapter 5) is the first to study the non-hierarchical unification of the uncoordinated checkpointing with the coordinated checkpointing.

3.3 Replication

Replication is a well-known technique that has been adopted in various domains from aviation to distributed systems [62]. 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. [63] 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. [64] propose a selective replication heuristic to increase QoS while keeping costs affordable in the context of 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 [63, 64] 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. [65] and Fang et al. [66] 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 heuristics in Chapter 7 are the first to address the trade-off between resource costs and application-specific reliability requirements, in particular for task-parallel HPC applications.

Meanwhile even though the performance and efficiency advantages of task-based dataflow programming models [5] and runtimes [67] 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 (Chapter 7). 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 [2, 38] (Chapter 7). As mentioned earlier, FTI [38] is a scalable multilevel checkpointing scheme addressing stop-failures such as DUE failures while Fiala et al. [2] propose SDC detection and correction methods for large-scale message-passing HPC applications. Ferreira et al. [68] investigates the applicability of process replication for exascale systems.

Research on reliability modelling, such as [69, 70, 71], has been conducted for various types of computing systems. However they all model the reliability of systems rather than that of applications to predict the failure rates of the systems. For instance, Thanakornworakij et al. [69] provide a model for HPC systems while Welke et al. [70] establish generic models

based on Markov modelling that are applicable for hardware or software systems having well-known architectures such as simplex and triplex modular redundant architectures. Elliott et al. [71] combine and analyse double and triple replication with checkpointing for message-passing HPC systems. However, none of these models can be utilized to achieve application-specific reliability with selective redundancy under application-specific rates of SDCs and crashes/DUEs.

3.4 SDC Detection and Mitigation

Research on SDC mitigation can be categorized mainly into three different categories: algorithm-based fault tolerance (ABFT) technique, replication of computation, and runtime analysis technique.

ABFT [48, 72, 73, 74, 75, 76] techniques are tailored solutions to specific numerical algorithms. As a result, they are usually efficient. However, they fundamentally lack the ability to be applicable to other algorithms than the specific numerical or algebraic kernel they are designed for.

Replication-based schemes [2] can be deployed for mission-critical situations. In such contexts, double or triple redundancy of computation is performed in order to detect SDCs by comparing the results of replica computations. The inherent drawback of the replication is its high cost; for instance, with double redundancy, the cost is 100%. Partial replication [77] has been proposed to decrease costs while providing required level of reliability. Partial or selective replication is shown to be promising (Chapter 7).

Runtime data analysis recently has gained attention in the HPC community. Studies [20, 78, 79] investigate and compare different prediction methods such as linear curve fitting or ARMA models, to detect SDCs. They convert the problem of detecting SDC into next-step prediction problem. Sharma et al. [80] utilize temporal features of data (in addition to spatial features) and provide a tailored SDC detector for stencil applications where they use support vector machines as a linear function approximator. The main drawbacks of temporal data analytics are the memory overhead and the computation cost of maintaining snapshot data. Subasi et al. [81] propose a spatial technique, called SSD, which incurs low memory cost while having low computation overhead.

3.5 Memory Reliability

We discuss memory reliability in two parts according to whether fault-tolerance is implemented mainly in software or hardware.

3.5.1 Hardware-based Memory Reliability

There have been hardware-based studies [82], [83] and [84] that propose stronger ECCs for HPC systems. For instance, Li et al. [82] propose memory protection by using BCH codes while decreasing the power overheads for certain applications; the proposed scheme incurs more overhead than Chipkill for computation-bound applications. Bamboo ECC [83] provides more flexibility in terms of data granularity and coding strength than Chipkill. As opposed to our design in Chapter 8, these techniques require custom hardware to deploy the proposed solution. Nevertheless, our framework (Chapter 8) is orthogonal to any hardware-based scheme, and can be used on top of the hardware schemes as a runtime-level scheme to protect critical and precise application data.

On a rather different research direction, Levy et al. [85] propose lightweight error correction for DRAMs by utilizing the existing similarities between the faulty page and some other DRAM pages for post-petascale supercomputers. However, this technique does not offer any error detection support and relies on the signals from the machine check architecture.

3.5.2 Software-based Memory Reliability

In comparison to hardware, software-based memory error fault-tolerance has not been investigated thoroughly for HPC. Our work [86] (Chapter 8) is the first to investigate different software techniques with respect to different costs (memory or computation) and different fault-tolerance capabilities. As a software-based technique, Maruyama et al. [87] introduce fault tolerance for GPUs that lack standard ECCs. They achieve error detection via data coding and recovery via checkpointing. This is similar to our approach however they do not utilize acceleration of coding. Fiala et al. [88] propose a software-based data corruption detection library which incurs 53% performance penalty on average, which is much higher than that of our CRC mechanism (Chapter 8). In contrast to our mechanism, their library is not immune to the overheads due to data access patterns within the application. In addition, we design and implement hardware acceleration of our CRC scheme, which is widely applicable in most of the state-of-the-art HPC

processors and incurs only 1.7% performance penalty (Chapter 8). The work of Borchert et al. [89] proposes flexible software-based ECCs which are in essence designed for operating system data structures.

4

NanoCheckpoints: A Task-Based Asynchronous Dataflow Framework for Efficient and Scalable Checkpoint/Restart

4.1 Introduction

In this chapter, we introduce NanoCheckpoints framework which consists of two checkpoint/restart mechanisms leveraging advantages of task-parallel dataflow programing. NanoCheckpoints is composed of two main designs: Baseline checkpoint/restart and the Singleton mechanism. Both NanoCheckpoints designs are at the runtime level, and do not require any modifications at all to application code or operating system (OS). Checkpointing and error recovery are task-local, asynchronous, automatic and completely transparent to the user. In the first design, we use the task directionality information to develop an efficient incremental checkpoint/restart scheme that only checkpoints minimal data to recover from errors. Our results indicate that the design is scalable even for high error rates. In addition, average checkpointing overhead

is about 6% of the fault-free execution times for shared memory benchmarks. In the second design, we develop a singleton copy mechanism that further decreases the checkpointing memory overheads. Singleton copy mechanism decreases the memory usage for checkpoints about 122× in comparison to the first design. Moreover, it has 3% performance overhead for shared memory benchmarks. Finally, in addition to the shared-memory applications, we evaluate NanoCheckpoints with hybrid OmpSs+MPI applications to demonstrate its very low overhead, on average 2%, and high scalability (with 1024 cores) for larger scales.

Our main contribution is the development of NanoCheckpoints, a task-local fault tolerance framework based on a dataflow runtime coupled with a task-based PM. To the best of our knowledge, this is the first work that explores the checkpoint/restart advantages of a dataflow runtime in detail. In summary, our contributions in this chapter are:

- A low-cost and scalable checkpoint/restart mechanism leveraging the asynchronous dataflow execution model with 6% performance overhead on average for shared memory benchmarks.
- A singleton copy mechanism to further minimize memory usage for checkpoints with 3% performance overhead on average for shared memory benchmarks.
- Evaluation of NanoCheckpoints for hybrid OmpSs+MPI benchmarks with 2% checkpointing overhead and high scalability.
- Analysis of checkpointing overhead in terms of time and space with respect to distinct computation and memory complexities of shared memory benchmarks.

4.2 NanoCheckpoints Design

In this section, we present the two NanoCheckpoints designs in detail. We implement our framework in publicly available OmpSs PM and its Nanos runtime. However, our framework is 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 [23], [25] 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 [26].

4.2.1 Baseline Checkpoint/restart Design

The first mechanism we propose is the baseline checkpoint/restart. We checkpoint the task inputs when the task is ready to execute, which means that all its data dependencies are guaranteed to be satisfied by the runtime. This is necessary for the mechanism to be correct since if we checkpoint before the data dependencies are satisfied, we may checkpoint incorrect task input data. We checkpoint to main memory. When a task is about to finish, in case of an error, the runtime restores and restarts its execution, otherwise releases the dependencies and removes the checkpoint and related software data structures. We detect errors via capturing signals propagated by OS/hardware at the runtime. Task computations are issued within try-catch blocks and signals are treated as exceptions. After catching the exception, the runtime restores the task and reissues it for a number of times which is provided as a configuration parameter. After these trials if the task computation does not succeed, the runtime schedules it to a new core, since most probably a permanent error is encountered in this case. If the computation does not succeed even after this migration, then the application aborts. When we evaluate our design we consider that the failure is detected at the end of a task, this way we penalize our scheme by assuming a relatively high error detection latency.

We support global modifying task computations, such as tasks updating a shared global variable, by the privatization of the modified data (although we did not encounter any mutable global state in the applications we studied in this work). In task-parallel dataflow programming, programmer expresses the global state modifications, such as a global variable modified within a task, as a shared variable by using data sharing attribute clause ***shared***. This shared variable is treated as an input and the variable is checkpointed just as the other inputs. Furthermore, the task works on a private copy of the variable. If the task successfully finishes its computation, then the value of the private copy becomes the value of the global variable at the end of the task and the private copy is deleted. If an error occurs, the task is restored as explained above where the private copy is restored by its checkpoint. We note that it is programmer responsibility to provide correct synchronization for globally shared variables among the tasks. However, this is mostly done in a straightforward way by simply using pragmas for atomic statements or for critical sections.

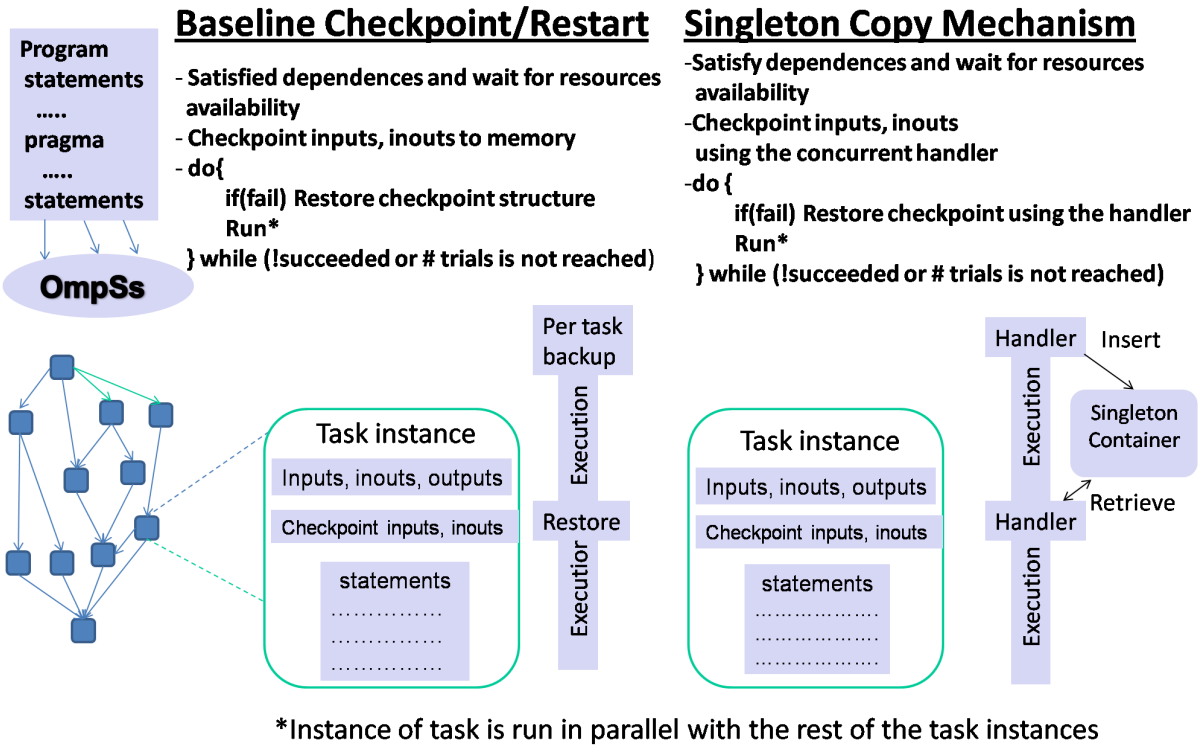


Figure 4.1: Designs of NanoCheckpoints

4.2.2 Singleton Copy Mechanism

In order to minimize the memory usage, we implement a singleton copy mechanism based on a concurrent hashmap. We exploit the fact that in a task parallel program, tasks are likely to share the same inputs. We implement a hashmap for maintaining the single checkpoints of task inputs. During the execution of a task parallel application, tasks insert or retrieve inputs into or from the map. Note that there is no additional synchronization needed for accessing a shared task input since by the construction of task dependencies no two tasks will be modifying a shared task input simultaneously. This mechanism enables significantly lower memory usage for checkpoints and lower checkpointing overhead as can be seen from our results in Section 4.3. Figure 4.1 shows the two designs of NanoCheckpoints.

4.3 Experimental Evaluation

In this section, we first present our methodology and experimental setup. Then we present our fault injection implementation. Finally, we discuss our results.

4.3.1 Methodology and Experimental Setup

We implement Nanos runtime counters through which memory/timing information is obtained by utilizing information from OS to measure time and memory consumption by our mechanisms. We run our experiments in Marenostrum supercomputer [90] hosted at Barcelona Supercomputing Center.

We conduct experiments to assess the behaviour of NanoCheckpoints in terms of scalability and the performance overheads with respect to fault-free executions. In addition, we perform experiments to see the improvements from our singleton copy mechanism in terms of memory usage and checkpointing overhead. We obtain all results by running each single case 10 times and take the average of overheads, speedups or execution times as the final results.

Our experiments are two-fold: The intra-node experiments on task-parallel shared memory benchmarks (listed in Table 4.1) are presented in Subsection 4.3.3. The inter-node experiments on hybrid task-parallel MPI applications adopting the OmpSs model (listed in Table 4.2) are presented in Subsection 4.3.3. With intra-node experiments, we aim to evaluate NanoCheckpoints in terms of overheads, scalability, efficiency and analyse it with respect to different characteristics of different applications in detail. With inter-node experiments, we will additionally show i) NanoCheckpoints is well-suited for such hybrid MPI programs and ii) it incurs low overhead, on average 2%, and is highly scalable at large scale and for high core counts. We run with upto 1024 cores and 64 nodes in these experiments.

For the intra-node experiments, we use four HPC benchmarks,-Sparse LU decomposition, Cholesky factorization, Fast Fourier Transform, Perlin Noise [91] and an artificial synthetic benchmark, the stream application. Table 5.2 shows information about benchmarks, in particular, the input data sizes, the complexity of the computation and memory operations and the block sizes which determine the input data sizes for a single task in the parallel program. For the artificial stream application, we rewrote and adapted McCalpin's implementation [92] by adding task pragmas and taskifying it to run in OmpSs. We choose this benchmark set such that we have different applications with different computational and memory complexity to assess

Sparse LU	LU decomposition O(N^3) computation, O(N^2) memory operations Higher ratio of computation to Memory Ops relative to Cholesky Matrix size 6400x6400, block size 100x100
Cholesky	Cholesky factorization O(N^3) computation, O(N^2) memory operations Matrix size 16384x16384 and block size 512x512
FFT	Fast Fourier Transform Array size 16384x16384, block size 128 O(NlogN) computation, O(N) memory operations
Perlin Noise	Noise generation to improve realism in motion pictures Array of pixels with size of 65536 (1500 iterations over it), block size 2048 O($(2^n) \times N$) computation, O(N) memory operations n is the number of dimensions of the noise function
Stream	Linear operations among arrays Array size 2048x2048 (doubles), block size 40K O(N) computation, O(N) memory operations

Table 4.1: Details of task-parallel shared-memory benchmarks

Nbody	Interaction between N bodies Array size 65536, block size depends on #nodes
Matrix Multiplication	Matrix Multiplication using CBLAS Matrix size 9216x9216 and block size 1024x1024
Ping-Pong	Computation and communication between pairs of processes Array size 65536, block size 1024
Linpack	HPL Linpack [93] ported to OmpSs Matrix size 131072, block size 256

Table 4.2: Details of distributed benchmarks

the impact of these complexities on the checkpointing overhead. In addition, with the synthetic stream benchmark, we aim to stress our mechanisms. We run with upto 16 cores (threads) in our intra-node experiments.

For the inter-node experiments, Table 4.2 shows the information about distributed benchmarks.

4.3.2 Fault Injection Mechanisms

We implement our fault injection mechanism to test the correctness of our mechanisms as well as to have an injection capability to corrupt the outputs of tasks at runtime. We inject faults in the outputs of the tasks by flipping bits of the outputs. We do multi-bit flips. Each flip is a random bit of a random element of the task output. We inject with different fault rates per task. We use fault rate x% to mean that the probability of a fault or faults occurring in a single task is x/100. This also corresponds to the injection of the faults approximately in x% of all tasks. We note that the fault rates in our experiments are high considering time frequency of faults in HPC systems, since the rates here are for *single* task. This effectively simulates higher fault rate for the overall system. The rationale for this is that if we achieve good performance for the upper bound case of high error rates, it would confirm our expectation that our implementations are what we call “failure scalable". Moreover, we do accelerated error injection, i.e. we simulate fault rates that are expected for future HPC and exascale systems and thus we stress test our mechanisms to assess how they cope with those boosted rates. We note that since we inject at runtime, the overheads reported in this work include also the fault injection overhead.

4.3.3 Evaluation and Discussion

In this subsection, we present the experimental results and discussion regarding them. We first present the intra-node experiments, and then present the inter-node experiments.

Checkpoint/Restart Results

Figure 4.2 illustrates the scalability of our mechanism for the benchmarks. We note that the speedups are over 1-thread case with the same fault rate. With the increasing number of threads, our mechanism scales very well except for stream. We see that it scales well up to 8 threads. The stream application does not even scale with 16 threads without any of our mechanisms enabled. This application does not have much parallelism other than operating on different parts of arrays with different threads. It mainly consists of memory operations. In addition, the dependencies are more tight compared to the other benchmarks. This is why it does not scale (even when fault tolerance mechanisms are disabled). We purposely chose and adapted it to analyze the behavior of such an extreme memory-intensive application in terms of performance. Moreover, since each single task has more inputs and larger input sizes relative to other

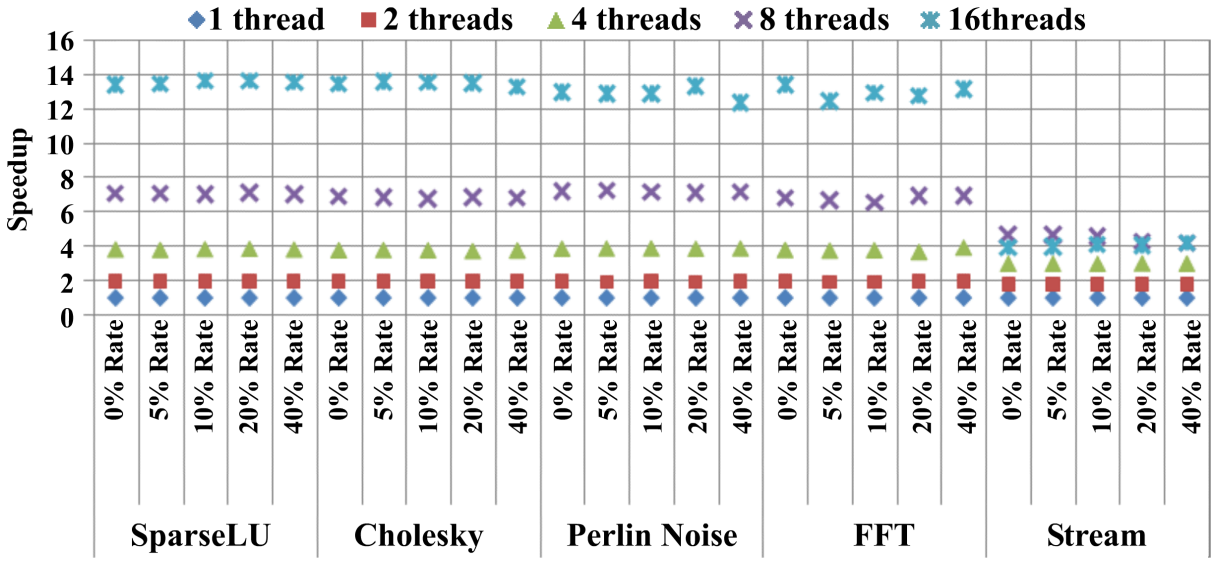


Figure 4.2: Scalability of Checkpoint/Restart

benchmarks, it is a good candidate to stress our design.

The checkpointing overheads of our design with respect to the fault-free execution times are 2%, 6%, 9%, 3% and 13% for Sparse LU, Cholesky, FFT, Perlin and Stream, respectively when 16 threads are utilized. Fault-free execution time refers to the execution time of a benchmark where no injections is performed and no mechanism is enabled. As we see, the checkpointing overheads are quite small with respect to total fault-free execution time. This shows that our design is quite efficient and does not degrade the performance of the applications. Our scheme improves usage of memory bandwidth due to the asynchronous and distributed nature of task executions. Tasks are distributed over threads, thus checkpointing is distributed over different threads of execution which constitutes a further reason of low overhead. The stream benchmark which we choose to assess characteristics of a memory-bound application incurs checkpointing overhead more 10% due to the fact that the benchmark consists of mainly memory operations rather than pure computation. Such applications have not only scalability problems but also checkpointing problems. Nevertheless, shortly we will show that the singleton copy mechanism reduces both memory usage for checkpointing and checkpointing overhead for the stream benchmark.

Our benchmarks provide a wide spectrum of computational complexity and memory operations. SparseLU and Cholesky benchmarks both have O(N^3) computation and O(N^2) memory

operations (see Table 5.2). Our experiments show that such a ratio of computation to memory operation incurs lower checkpointing overhead. In particular, the coefficient of the computational complexity of the SparseLU implementation we use is bigger than that of Cholesky with a factor of about 3. This further affects the checkpointing overhead. The FFT benchmark has a computation of O(NlogN) and memory operations of O(N) complexity. This lower ratio of computation to memory operations incurs a bigger ratio of checkpointing time to fault-free execution time. For Perlin Noise, even though it has an exponential coefficient in the computation complexity and tasks are computation-bound, the massive number of duplicate checkpoints leads to a higher checkpointing overhead with respect to SparseLU. Again, the singleton copy mechanism reduces the overhead to the lowest among our benchmarks and uses more than 380× less memory for the checkpoints. In the case of the stream, the complexities are the same in terms of computation and memory. This couples with higher input numbers and larger sizes of inputs of the tasks resulting in a higher overhead. Overall, except for the stream, checkpointing overheads are less than 10% of fault-free executions.

Figure 4.3 shows the scalability of checkpointing overheads in all applications. We see that as the number of threads increases, less overall time is used for checkpointing. The speedup of the checkpointing overhead depends not only on the size of the checkpoint but most significantly on the task dependency graph as it leads to different level of overlapping of checkpointing and computation in different dependency graphs. This is why we observe different speedups for different benchmarks at high number of threads. The special super-linear case for stream is due to the fact that with 8 and more cores the checkpoints fit in the caches leading to a boosted decrease in checkpoint times. Checkpoints may or may not be flashed to main memory later since at the end of the tasks checkpoints are deleted. Even though sometimes checkpoints are moved to main memory, this data movement is efficiently overlapped with the computation of tasks.

Singleton Copy Results

The scalability of the singleton copy mechanism is very similar to the baseline implementation, consequently we omit figures for brevity. For instance, even with 40% per task fault rate, the average speedup for 16 threads for the singleton copy mechanism is 13.6 excluding stream benchmark over the 1-thread case. For stream, these speedups are little over 3.

Our experiments demonstrate that when the Singleton copy mechanism is enabled, the over-

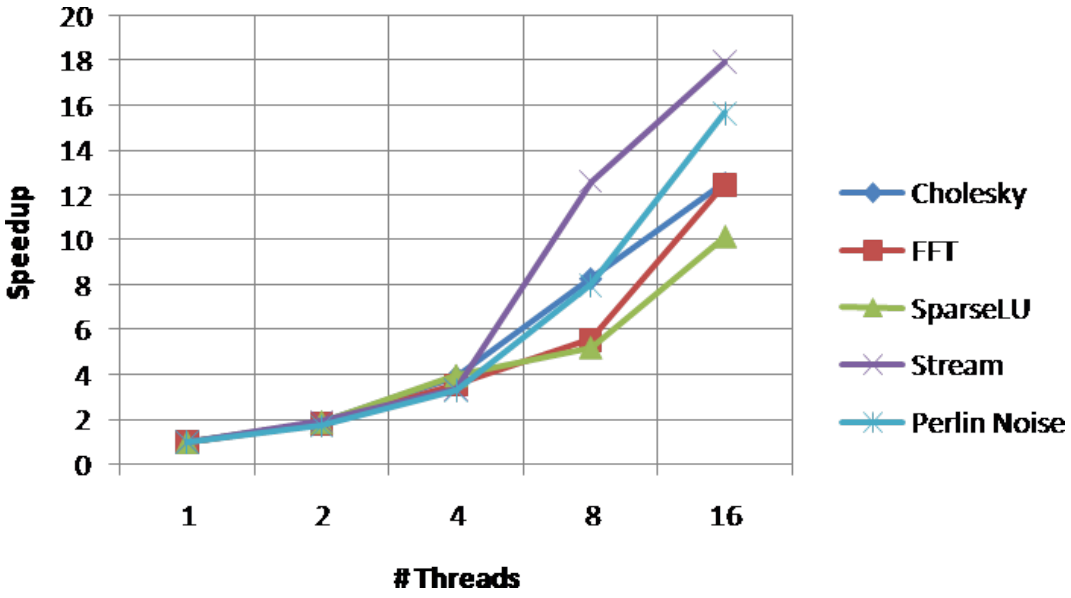


Figure 4.3: Scalability of Checkpointing Overheads

all execution times improve in comparison to the first design. The overall execution time improvements are not much for SparseLU, Cholesky, Perlin Noise and FFT. For Stream, the overall execution time improves and decreases dramatically when the singleton mechanism is used. For 1 or 2 threads, the overall execution time is almost halved. When it uses 4, the overall time is improves 60%. For 8 and 16, the effect is more visible than for the other benchmarks.

The value of the Singleton mechanism becomes evident when we evaluate the checkpointing overhead and memory usage of it. Table 4.3 (b) shows the checkpointing overhead with respect to fault-free execution times when Singleton copy mechanism is utilized (Table 4.3 (a) shows the checkpointing overheads of our first design discussed in the previous section) when 16 threads are used for benchmark executions. When compared to the checkpointing overheads of the first design, we see that for SparseLU the checkpointing overhead drops about 10×, for Cholesky drops about 20× and for FFT drops by about 0.23×. In FFT, tasks have a few common inputs and this prevents to improve more in terms of checkpointing overhead. For Perlin, the drop is about 30× with respect to the first design proving the effectiveness of the mechanism. We note that relative complexity of computation and memory operations is in effect in terms of checkpointing overhead. That is, the more relative computation (Perlin > SparseLU > Cholesky > FFT > Stream), the less the checkpointing time overhead (Perlin < SparseLU

Benchmarks	Sparse LU	Cholesky	FFT	Perlin	Stream
(a) Overhead	2%	6%	9%	3%	13%
(b) Overhead	0.2%	0.3%	7%	0.1%	8%

Table 4.3: Checkpointing Overheads (a) with First Design (b) with Singleton Copy Mechanism

$<$ Cholesky $<$ FFT $<$ Stream) provided that the checkpoints are handled in an efficient way. Stream also gains an improvement of about 1.6× over the first design. This demonstrates that the singleton mechanism improves the checkpointing overhead even for an artificial, memory-bound application with a relatively high memory access / computation ratio. The performance of the singleton mechanism is facilitated by our fast hash-table implementation which has, on average, insert and lookup times of 1 and 6 microseconds, respectively.

Lastly, the Singleton copy mechanism provides significant improvements in memory usage for storing checkpoints which indicates the value of it: Singleton copy mechanism not only improves the checkpointing overhead but also the memory usage for checkpoints. For SparseLU and Cholesky, the improvement is about 32× over the first design. As we explained before FFT tasks have a few common inputs and the improvement is 2×. However, for Stream application, the memory usage is reduced about 166× because of higher number of common task inputs. In Perlin Noise, the number of duplicate checkpoints is even higher. We have a significant reduction in memory usage that is about 382×.

Efficiency of Checkpointing Mechanisms

In this section, we demonstrate how our mechanisms are efficient. We define the efficiency of a mechanism as fault-free execution time when the mechanism is disabled divided by fault-free execution time when it is enabled. Basically, the efficiency definition captures the overhead of the mechanism when it is enabled but no faults occur. Note that the closer to 1, the more efficient the mechanism is. Table 4.4 shows the efficiency of our singleton copy mechanism. It is very efficient for all benchmarks except Stream on low thread counts. As stated before Stream is a purposely chosen benchmark to see effects of checkpointing in a extreme case. However, as the number of threads increases, the efficiency increases close to 1, which is due to the fact that larger number of threads overlaps checkpointing and computation more effectively. The results for the first design are similar to these results.

# Threads	SparseLU	Cholesky	FFT	Perlin	Stream
1	0.985	0.988	0.933	0.988	0.770
2	0.994	0.989	0.931	0.997	0.826
4	0.998	0.989	0.928	0.998	0.808
8	0.998	0.987	0.914	0.999	0.924
16	0.998	0.989	0.930	0.999	0.922

Table 4.4: Efficiency of Singleton Copy Mechanism

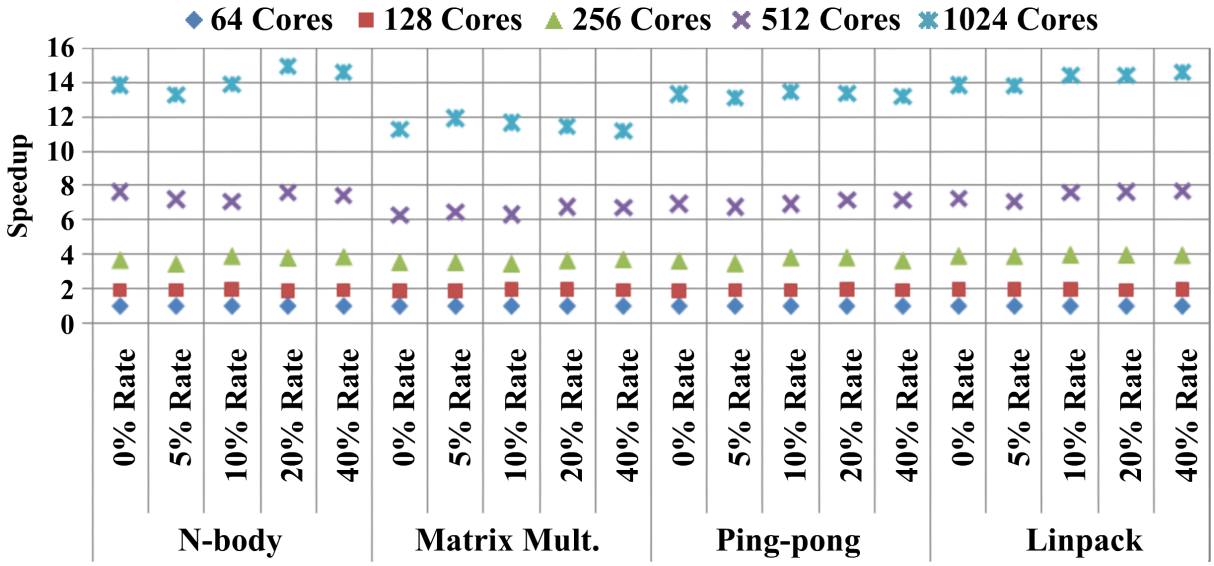


Figure 4.4: Scalability of Singleton in distributed benchmarks

Inter-node Results

Inter-node experiments performed on hybrid task-parallel OmpSs+MPI benchmarks. Figure 4.4 shows the scalability of the singleton mechanism for these benchmarks. The figure shows the speedups over 64 cores with the same fault rate. As seen, our mechanism is highly scalable for even much higher core counts. The checkpointing overheads in comparison to fault-free execution times when 1024 cores are utilized are 5%, 2%, 0.2% and 0.1% for N-body, Matrix Multiplication, Ping-pong and Linpack, respectively. These low overheads demonstrate that NanoCheckpoints incurs low performance overhead even for larger scales. The results for the first design are similar and omitted for brevity.

4.4 Conclusions

In this chapter, we introduce our fault-tolerance framework, NanoCheckpoints, a runtime-level checkpoint/restart library for task-parallel dataflow HPC applications to mitigate fail-stop errors. NanoCheckpoints offers two designs. We showed that our first design is low-overhead and is highly scalable. Then, we show that our second design, Singleton copy mechanism, improves memory usage and checkpointing overheads. Finally, we show NanoCheckpoints is also efficient and scalable with hybrid distributed applications at large scale with high core counts. In the next chapter, we will unify NanoCheckpoints with system-wide checkpointing to leverage lightweight task-level checkpoints while maintaining high failure coverage by means of global checkpoints.

5

Marriage Between Coordinated and Uncoordinated Checkpointing for the Exascale Era

5.1 Introduction

As HPC systems grow in capability and capacity, which are anticipated to have millions of cores and components, the state of the art checkpoint/restart techniques will be prohibitively expensive. These techniques are either coordinated or uncoordinated with message logging accompanying checkpointing. While coordinated schemes suffer high computation overheads due to synchronization to achieve a consistent global state, uncoordinated schemes incur high amount of memory consumption due to message logging. Moreover, the coordinated techniques can cause high pressure on I/O such as Parallel File Systems (PFS) thereby creating congestion. Furthermore, these techniques are mostly holistic, i.e. system-wide, in nature. As a result, they do not take into account programming model and paradigm specific aspects to

leverage so as to decrease the performance overheads to be viable for the Exascale era. In the previous chapter (Chapter 4), we proposed NanoCheckpoints, a checkpointing framework that leverages the properties of task-based dataflow programming models. In this chapter, we propose a unified approach to combine uncoordinated checkpoint/restart, i.e. NanoCheckpoints, with coordinated checkpointing. Our objective is to decrease the system-wide checkpointing overheads and to obtain a faster recovery mechanism while not sacrificing the complete failure coverage provided by a system-wide scheme.

To achieve our objective, we unify task-level uncoordinated, i.e. NanoCheckpoints, and system-wide coordinated checkpointing. Task-level checkpointing provides the following advantages when unified with a system-wide scheme: i) Since it mitigates a fraction of failures that was to be handled by system-wide checkpointing, the checkpoint frequency can be decreased. This way, the total number of system-wide checkpoints is decreased and as a consequence, the checkpointing overheads are reduced. ii) The failure recovery and restart become orders of magnitude faster with task-level checkpoints because of two reasons. First, task-level checkpoints are in-memory and fast as opposed to the global checkpoints that are stored on the disk if not on the PFS. Second, most often the task-level checkpoints are able to mitigate errors avoiding expensive global recovery. iii) With the unified approach, the scope of the failure becomes the task computation. As a result, task-level checkpointing provides fine-grain failure containment than system-wide checkpoints. Consequently, the amount of useful computation lost due to a failure becomes the computation since the beginning of a task rather then the beginning of a system-wide checkpoint interval. Because system-wide checkpoint intervals are much longer than a single task computation, the amount of useful computation lost is reduced significantly.

To materialize these observations, we provide a unified mathematical model to optimize the checkpoint interval in which the system-wide optimal checkpoint interval is increased. Our model quantifies this increase. Moreover, we analytically formulate the performance gain that is achieved by the unified approach. Figure 5.1 provides an overview of our analytical assessment and its implications. We see that as failure rate and task-level failure coverage increase, the performance gain increases.

Our main contribution is a unified approach that combines uncoordinated and coordinated checkpointing. *To the best of our knowledge, this is the first mathematical framework that unifies task-level uncoordinated checkpointing with system-wide coordinated checkpointing.* Briefly, our main contributions in this chapter are:

- Development, validation and evaluation of a unifying mathematical model for task-parallel HPC applications. We analytically derive a closed formula for the optimal checkpointing interval of the unified model.
- A closed formula to compute the performance gain when the task-level checkpointing is combined with a system-wide holistic checkpointing scheme. Simulations demonstrate that the performance gain can be as high as 306%. Our results for actual runs of real-world HPC benchmarks indicate that the performance gain is 282% of the total execution time on average and can be as high as 539%.
- A closed formula to compute the decrease in the checkpoint frequency of a system-wide checkpointing scheme. Our benchmark results show that the checkpoint frequency can be decreased 5.3× while not sacrificing complete failure coverage.

The rest of this chapter is organized as follows: Section 5.2 presents our motivation. Section 5.3 provides the formalization of our model. Section 5.4 presents the formulation of performance gain. Section 5.5 presents the experimental evaluation. Finally, Section 5.6 summarizes the chapter.

5.2 Combining Task-level Checkpointing with System-wide Checkpointing

We propose combining uncoordinated task-level checkpointing with coordinated system-wide checkpointing. This combination is *non-hierarchical* since task-level and system-wide checkpointing are not coupled together such as multilevel systems from Bautista-Gomez et al. [38] and Sato et al. [39]. One cannot enforce and specify a periodic checkpoint interval at the task level. Due to the data-driven semantics, tasks are executed out-of-order and asynchronously where only data dependencies are obeyed.

There are several advantages of the integration of task-level checkpointing with system-wide checkpointing. First, since task-level checkpointing mitigates a fraction of failures, the checkpoint frequency of system-wide checkpointing can be decreased to improve the overall application performance. Second, the restart overhead of task-level checkpointing is significantly lower than that of system-wide checkpointing since task-level checkpointing only restores in-memory task inputs while system-wide checkpointing will likely need to read a

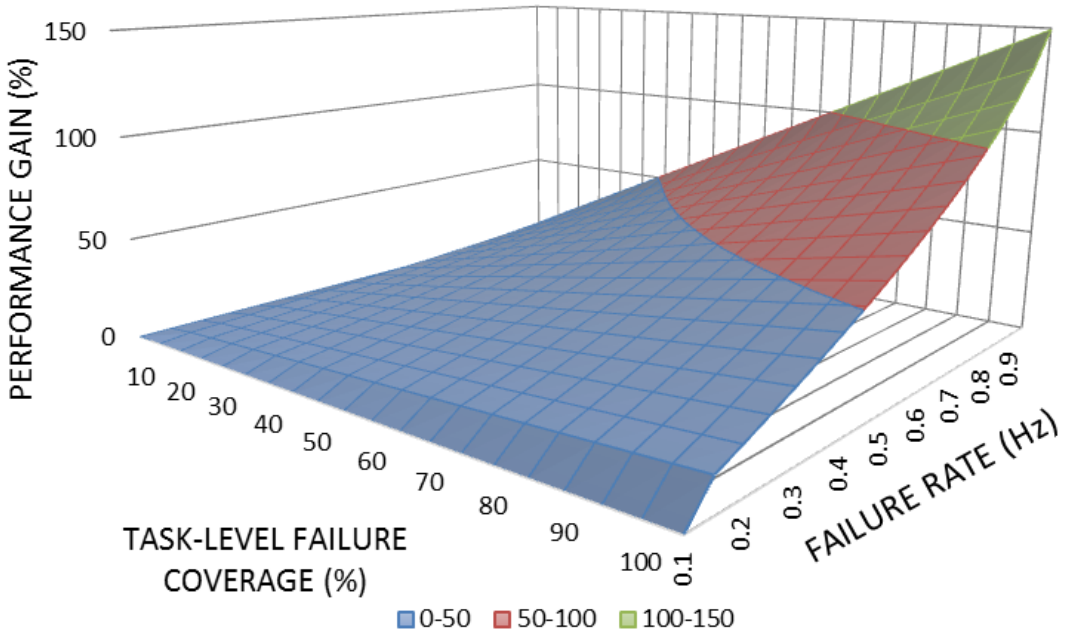


Figure 5.1: Performance gain predicted by our analytical model as a function of task-level failure coverage and failure rate. As seen, increasing failure rate - expected for the Exascale era - is the dominating factor for the performance gain (improvement).

checkpoint from the parallel file system (PFS) or a local disk. Third, the combined model can be instantiated with application and system specific parameters, such as the system-wide checkpoint and restart times for an application, so that the optimal checkpoint interval is tuned for the application and the total waste is minimized particularly for the application.

Task-parallel dataflow programming has its own inherent merits that enable affordable fault tolerance for future HPC systems, which we also discussed in the Background (Chapter 2). First, task-level checkpointing offers fine-grain *failure containment*, that is, limits the scope of failures to task computations instead of the entire application. This effectively means that task-level checkpointing recovers to the point of failure much faster since tasks are usually much shorter than the system-wide checkpointing interval. As a result, the amount of lost work is smaller for task-level checkpointing compared to system-wide checkpointing. Second, the

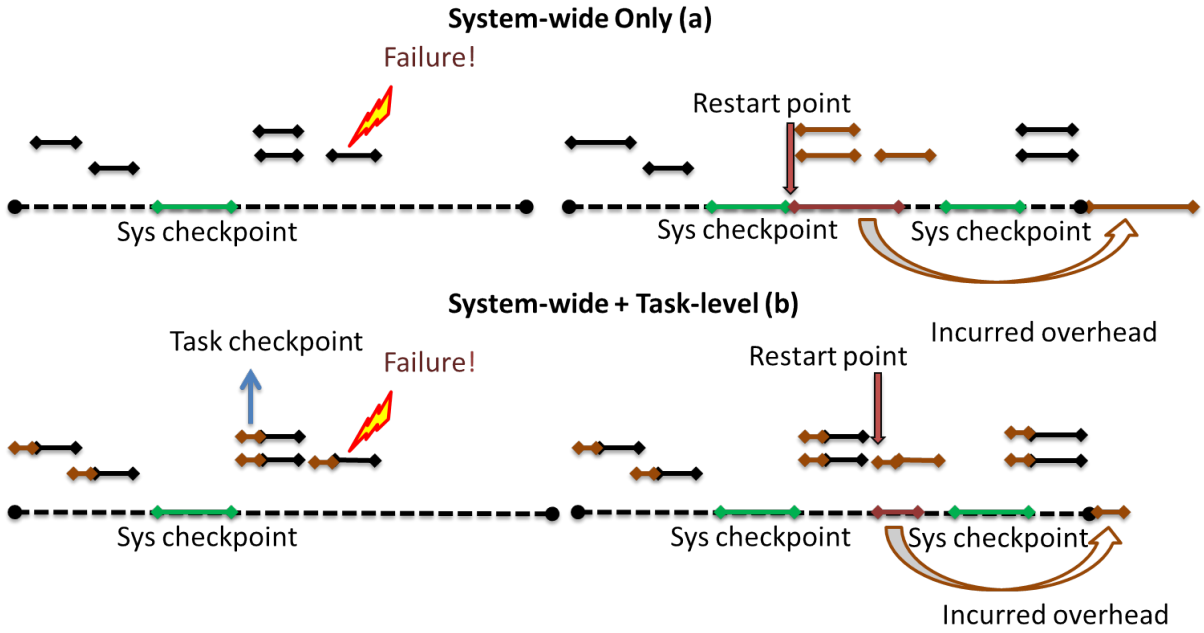


Figure 5.2: System-wide vs. System-wide + Task-level Checkpointing: Short line segments show task computations. In the system-wide only case, in (a), the restart point is the last global checkpoint. In the combined scheme, in (b), the restart point is the beginning of the failed task computation. Note that in the system-wide only case, we lose all computation since the last global checkpoint whereas in the combined scheme, it is only the computation since the beginning of the failed task.

asynchronous nature of data-driven execution enables low-overhead error recovery by overlapping recovery of faulty task(s) with execution of other tasks that are independent of the faulty task(s). This means that task-level checkpointing is uncoordinated. Figure 5.2 visualizes these observations. Third, task inputs are available through annotations which represent the minimal state of the tasks needed to be checkpointed for the failure recovery. In contrast, holistic system-wide solutions typically checkpoint the entire address space of the application since they are not aware of the minimal state. Finally, the performance of dataflow parallelism is shown to be higher than the traditional fork-join parallelism due to amount of implicit and irregular parallelism achieved [7].

5.3 Model Formalization and Analytical Assessment

We build a mathematical model to combine task-level checkpointing with a system-wide checkpointing scheme. If task-level checkpointing cannot handle the errors occurred, such as those that corrupt kernel, runtime and MPI library, then the system-wide scheme will recover from those errors. The model targets the detected errors that prevent the successful completion of the application execution, i.e. fail-stop errors. The undetected errors, called silent errors, are mitigated by our techniques explained in Chapter 7.

Table 5.1: Model Parameters

Parameter	Description
τ_s	Checkpoint interval of system-wide checkpoints
c_s	Time to take a system-wide checkpoint
r_s	Time to restart with system-wide scheme
T_{ff}	Failure-free computation time without fault tolerance
μ_s	Expected rate for fail-stop errors
$T_{overall}$	Total execution time including fault-tolerance
$CovTL$	Failure coverage of task-level checkpointing from fail-stop errors
T_{TL}	Total amount of time used for task-level checkpointing
T_{sys}	Total amount of time used for system-wide checkpointing
T_{wasted}	Total amount of time used for task-level and system-wide scheme
W_{TL}	Total waste per unit of time of task-level checkpointing
W_{sys}	Total waste per unit of time of system-wide scheme
C	Fraction of time that an application is performing task computations

Table 5.1 shows the model parameters. We first introduce the total time equations:

$$\begin{aligned} T_{overall} &= T_{ff} + T_{wasted} \\ &= T_{ff} + T_{TL} + T_{sys} \qquad (5.1) \\ &= T_{ff} + (W_{TL} + W_{sys})T_{overall} \qquad (5.2) \end{aligned}$$

The first equation shows the overall execution time which includes the wasted time due to task-level checkpointing and the wasted time due to system-wide checkpointing. The second equation shows the breakdown of the overall execution time with respect to the total waste per unit of time which we will minimize. We note that the *failure coverage* $CovTL$ refers to the fraction of time that a task-level scheme is able to recover from fail-stop errors. Formally, it is

defined as

$$CovTL = C \times e^{-\int_0^t \lambda(\theta) d\theta} \tag{5.3}$$

where C is a factor showing the fraction of time that a task-parallel application is performing task computations, that is, the execution time except the time consumed in runtime and MPI library, and in system calls. During the time in runtime and MPI library and in system calls, task-level checkpointing cannot mitigate errors which we need to factor out. In addition, $e^{-\int_0^t \lambda(\theta) d\theta}$ is the probability of the successful execution of a task-parallel computation from the beginning of the computation until time t with some failure distribution $\lambda(\theta)$ [94]. Our model does not assume any particular distribution for $\lambda(\theta)$.

We detail the waste per unit of time of system-wide checkpointing, i.e. W_{sys}, in Section 5.3.1 and then the waste per unit of time of task-level checkpointing, i.e. W_{TL}, in Section 5.3.2. In Section 5.3.3, we analytically find the optimal checkpoint intervals of the system-wide scheme and of the combined scheme. Finally, we conclude the model formalization with Section 5.3.4 where we discuss the generality of our model.

5.3.1 The waste time of a system-wide scheme

The waste per unit of time of a system-wide scheme, i.e. W_{sys}, is the sum of the checkpointing, rework and restart time per unit of time. Checkpointing time captures the overhead of taking checkpoints. The rework time is the lost computation from the latest checkpoint to the moment of the failure. The restart time refers to the time to restore the latest checkpoint. We include down times, if any, in restart times since they do not require any special treatment. Now we formalize checkpointing, rework and restart time.

The checkpoint overhead per unit of time of a system-wide scheme, denoted by W_{syschk}, is the product of the time to checkpoint c_s and the number of checkpoints which is $\frac{1}{\tau_s}$. Hence the checkpoint overhead per unit of time is

$$W_{syschk} = \frac{c_s}{\tau_s} \tag{5.4}$$

When a failure occurs, on average, half of the computation is lost since the last checkpoint. Thus, the expected value of rework overhead per unit of time of a system-wide scheme, denoted

by W_{sysrew}, is

$$W_{sysrew} = \frac{\mu_s \tau_s}{2} \tag{5.5}$$

Restart overhead per unit of time, denoted by W_{sysres}, is the product of the failure rate and the time to restore the checkpoint. Thus,

$$W_{sysres} = \mu_s r_s \tag{5.6}$$

Therefore the waste per unit of time of a system-wide scheme without task-level checkpointing is

$$W_{sys} = \frac{c_s}{\tau_s} + \frac{\mu_s \tau_s}{2} + \mu_s r_s \tag{5.7}$$

5.3.2 The waste time of task-level scheme

The waste per unit of time of a task-level scheme, W_{TL}, is constituted by the checkpoint per unit of time of all tasks, and the rework and restart per unit of time of the failed tasks during the program execution. Let $FailTs$ be the set of the failed tasks during the program execution due to fail-stop errors. In addition let α be a factor showing the fraction of the total wasted time of task-level checkpointing and recovery that is reflected in wall-clock time of the application execution. So for instance if $\alpha = 0$, then the application perfectly overlaps the computation with the task-level checkpointing and recovery and there is no overhead reflected in the wall-time of the application. On the other hand if $\alpha = 1$, then it means that task-level fault tolerance is sequential and fully reflected in the wall-time of the application. Note that $0 <= \alpha <= 1$.

The checkpoint overhead per unit of time, W_{TL}^{chk}, is the rate of taking task-local checkpoints. Hence the checkpoint overhead per unit of time of task-level checkpointing is

$$W_{TL}^{chk} = \sum_{\forall i, Task_i} Task_i^{chk} \tag{5.8}$$

where $Task_i^{chk}$ is the checkpoint time per unit of time of task $Task_i$.

The rework overhead per unit of time for a single task is the computation lost since the beginning of the task. Thus the rework overhead per unit of time, W_{TL}^{rew}, of task-level check-

pointing is

$$W_{TL}^{rew} = \sum_{Task_i \in FailTs} \mu_s \times Task_i^{rew} \tag{5.9}$$

where $Task_i^{rew}$ is the lost computation overhead per unit of time of the failed task $Task_i$ since the beginning of its computation.

The restart overhead per unit of time, W_{TL}^{res}, is the per unit of time of restoring the task-local checkpoint. Thus the restart overhead per unit of time of task-level checkpointing is

$$W_{TL}^{res} = \sum_{Task_i \in FailTs} \mu_s \times Task_i^{res} \tag{5.10}$$

where $Task_i^{res}$ is the restart overhead per unit of time of task $Task_i$.

Finally the waste per unit of time of a task-level checkpointing scheme is

$$W_{TL} = \alpha(W_{TL}^{chk} + W_{TL}^{rew} + W_{TL}^{res}) \tag{5.11}$$

In the next section we combine the two models.

5.3.3 The waste time of the combined model

If the task-level scheme provides the failure coverage $CovTL$, μ_s is decreased as

$$\mu_s' = (1 - CovTL) \times \mu_s \tag{5.12}$$

which leads us to parametrize the waste per unit of time of the system-wide scheme with respect to the expected failure rate.

Let $\mu_s' = (1 - CovTL) \times \mu_s$. We then denote the waste per unit of time of the system-wide scheme under failure coverage $CovTL$ with $W_{sys}(\mu_s', CovTL)$. As a result, note that $W_{sys}(\mu_s', 0)$ denotes the waste per unit of time of the system-wide scheme without task-level checkpointing. The waste per unit of time of the system-wide scheme under failure coverage $CovTL$ is

$$W_{sys}(\mu_s', CovTL) = \frac{c_s}{\tau_s} + \frac{\mu_s' \tau_s}{2} + \mu_s' r_s \tag{5.13}$$

Then the total waste per unit time of the combined scheme is

$$W = W_{TL} + W_{sys}(\mu_s', CovTL) \tag{5.14}$$

We now compute the optimal checkpoint intervals of both the system-wide only and the unified scheme. To find the optimal interval of the system-wide only scheme, we take the derivative of Equation 5.7 with respect to τ_s to and equate it to zero to find the global minimum. We compute it as

$$\tau_s^* = \sqrt{\frac{2c_s}{\mu_s}} \tag{5.15}$$

which is similar to the Young's formula [52].

The waste per unit of time of the task-level scheme W_{TL} is independent from τ_s of the system-wide scheme. That is, the task-level overheads and waste per unit of time is independent from the checkpoint interval of system-wide scheme. W_{TL} then can be treated as a constant - with respect to τ_s - such that the total waste per unit of time of the combined scheme in Equation 5.14 is still a convex function and thus we can find a global minimum. The first derivative of Equation 5.14 with respect to τ_s is given by

$$\frac{dW}{d\tau_s} = \frac{\mu_s'}{2} - \frac{c_s}{\tau_s^2} \tag{5.16}$$

where the derivative of W_{TL} with respect to τ_s is zero. Setting the derivative to zero, the solution is

$$\tau_{comb}^* = \sqrt{\frac{2c_s}{\mu_s'}} \tag{5.17}$$

Hence, if the task-level scheme has coverage $CovTL$, then τ_s^* is increased by a factor of $\gamma = \sqrt{\frac{1}{1-CovTL}}$, which we call the *gamma factor*, according to Equation 5.17. That is, the optimal solution of the combined model is

$$\tau_{comb}^* = \sqrt{\frac{1}{1-CovTL}}\sqrt{\frac{2c_s}{\mu_s}} = \gamma\tau_s^* \tag{5.18}$$

5.3.4 Discussion on the Generality of the Model

The unified model can be applied to any non-hierarchical scheme where there is an uncoordinated fault tolerance scheme, such as task-level checkpointing, that is used with a coordinated checkpointing scheme. The coordinated scheme can be any system-wide scheme where the uncoordinated scheme decreases the optimal checkpoint intervals in accordance to Equation 5.18. Moreover, this effectively also means that our model is orthogonal to studies done for coordinated or hybrid hierarchical systems such as Di et al. [57] and Bosilca et al. [60].

Our model enables application users tune the optimal checkpoint interval according to their applications. For instance, the system-wide checkpoint and restart times for application data differ from application to application, as evident in Table 5.5 from our results. They can tune the optimal checkpoint interval with respect to their applications' checkpoint and restart times. In addition, if their applications exhibit different failure rates as reported by Fang et al. [95], the optimal checkpoint interval can be adapted further.

5.4 Performance Benefits of the Combined Model

The *performance gain* of the combined model can be computed by calculating the difference between the overhead of the system-wide only scheme and the overhead of the combined scheme. In the combined model, even though the task-level scheme incurs overhead due to its task-local checkpointing, the model increases the optimal checkpoint interval and thereby decreasing the number of system-wide checkpoints. This effectively reduces the checkpoint overhead of the system-wide scheme which is much higher than that of the task-level scheme. The *performance gain* G_{ut} of the combined model is:

$$G_{ut} = W_{sys}(\mu_s', 0) - W_{sys}(\mu_s', CovTL) - W_{TL} \tag{5.19}$$

Further G_{ut} can be simplified as:

$$G_{ut} = (1 - \frac{1}{\gamma})\left(\frac{c_s}{\tau_s} + \frac{\mu_s \tau_s}{2} + (1 + \frac{1}{\gamma})\mu_s r_s\right) - W_{TL} \tag{5.20}$$

As an example, assume $CovTL = 0.75$, $c_s = r_s = 10$ seconds, and $\mu_s = 1/180$ Hz (1 error every 3 minutes). The system-wide only optimal checkpoint interval, τ_s, is calculated to

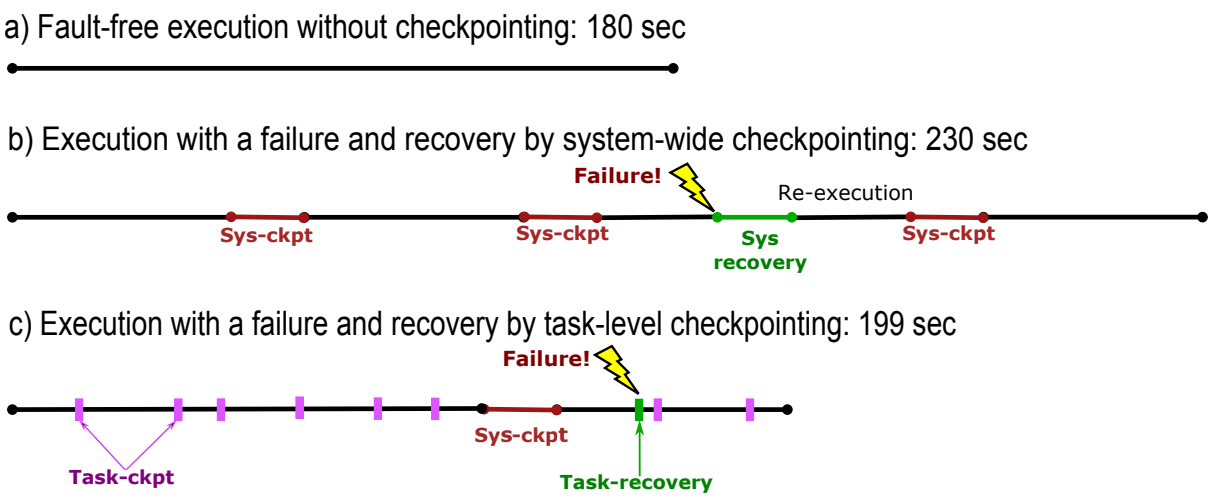


Figure 5.3: Comparison of the Unified Model to the System-wide-only Model

be 60 seconds. Since $\gamma = 2$, the optimal checkpoint interval for the combined model is 120 seconds. In addition, let the task-level waste time per unit time, W_{TL}, be 5% of the fault-free execution time. Then G_{ut} is calculated to be 11/120 seconds. If, say the fault-free execution is 180 seconds, with the combined model, the execution takes about 199 seconds, thus the overall gain is 19/120 × 199 = 31.5 seconds which is a significant gain. Figure 5.3 visualizes this example.

An important observation is that since $\mu_s' = (1 - CovTL) \times \mu_s = 1/720$ Hz (a failure every 12 minutes instead of every 3 minutes), the system-wide scheme does not encounter any failure for which it has to recover when combined with the task-level scheme. This way, a system-wide recovery and restart is avoided which is much more expensive than a task-local recovery and restart.

5.5 Evaluation

We use NanoCheckpoints (Chapter 4) [96] as our task-level scheme and publicly available FTI [38] as our system-wide scheme for our experimental evaluation. However, our model is applicable to any task-level and system-wide checkpointing scheme.

We present three-fold experimentation. First, in Section 5.5.1, we present the experimentation to evaluate the validity of our unified model. Second, in Section 5.5.2, we assess in detail the implications of the validated model through simulations. Finally, in Section 5.5.3, we eval-

uate the combined model for a set of task-parallel distributed applications. We will underline important takeaway lessons throughout the evaluation.

5.5.1 Model Validation

To validate our model, we perform failure simulation for which we wrote a simulator. The simulator generates failure sequences where failure arrivals follow Weilbull distribution. The shape parameter was 1 (exponential distribution) and the scale parameter was the number of tasks - for which we study a range - to have reasonable failure distributions in the simulations. Both the model and the simulator were fed with the same parameter values. We set parameter values to be on par with the predictions of studies [3] and [97]. We compare the total execution time predicted by our model to the total execution time of simulations in the presence of failures. We perform 10^4 simulations for each single case. Overall, the average difference between the total execution time predicted by the model and by the simulations is less than 0.24% of the execution time. The standard deviation was found to be 6.26 (over application execution time of 10^5 seconds) showing the consistency that the simulations and the model are predicting close values.

Moreover, we investigate the impact of different parameters on the accuracy of our model via simulations. Figure 5.4 a) shows the effect of the failure rate. The standard deviation is on the left axis and the average difference in percentage is on the right axis. We see that failure rate does not compromise the validity of the model. Likewise Figure 5.4 b) shows the effect of the system-wide checkpoint time to local storage on the accuracy of the model. We conclude that the system-wide checkpoint time does not interfere with the predictions of the model. Figure 5.4 c) shows the effect of the task granularity while Figure 5.4 d) shows the effect of the application execution time on the accuracy on the model. We see that these parameters do not affect the accuracy of our model.

The adaption and usage of our unified model is straightforward. First, given the failure rate of the system, we set the optimal checkpoint interval according to Equation 5.18 in the configuration file of FTI. Moreover given the failure distribution $\lambda(\theta)$, the runtime dynamically computes the task-level failure coverage as the application execution progresses. We can set the C factor in the Equation to be zero in the beginning. Recall that the C factor, defined in Section 5.3, refers to the fraction of time that the application execution is in a task computation. Then as the application execution progresses, NanoCheckpoints measures and maintains on-the-fly

the C factor at runtime. NanoCheckpoints' first update will set the C factor some value other than zero, for instance from 0 to 0.9. As the application progresses, if the C factor changes significantly, such as from 0.9 to 0.8, then NanoCheckpoints simply recomputes the optimal checkpoint interval and resets the value in the FTI configuration file.

5.5.2 Analysis of Unified Scheme

In this section, we investigate the impact of different parameters on the performance gain that our model provides via simulations. Figure 5.5 a) shows the impact of system-wide checkpoint time on the performance improvement. We see that the performance improvement increases as the checkpoint time increases. The impact of system-wide restart time is similar to checkpoint time and omitted. For these simulations, the application execution time is fixed as 10^5 seconds and the failure of the system is fixed as 0.001 Hz. Figure 5.5 b) demonstrates that the effect of the failure rate with respect to total execution time. We see that as the failure rate increases from every 10^5 seconds to every 2 seconds, the performance gain increases and can be as high as 306%. Given that exascale error rates can be on the orders of minutes or seconds [97] [3], the unified model provides significant reduction in performance overheads as well as decreases the demand for I/O which is a main source of bottleneck. For these simulations, we fix the application execution time as 10^5 seconds and the checkpoint and restart time as 10 seconds. The impact of task-level failure coverage in Figure 5.5 c), i.e. the fraction of time that task-level checkpointing recovers a failure, is similar to the impact of the system-wide checkpoint time.

Figure 5.6 presents the amount of performance gain or improvement in seconds as the parameters change. From these simulations, we conclude that it is important to taskify a parallel application as much as possible not only for performance - as a result of parallelism - but also for reducing the checkpointing overheads. This is because the more an application is taskified, the more likely a failure is mitigated by task-level checkpointing. Overall the results show the merit of the unified model for the Exascale era.

Takeaway 1: *Analysis on the simulations points that as the system-wide checkpoint time and failure rates increase, which is expected for the Exascale era, so does the performance gain of the unified approach. This corroborates the adaption of the unified approach so that checkpointing can be a viable technique for the Exascale era.*

Takeaway 2: *It is important to taskify an application not only for the performance gained from parallelism but also for the performance gain from the reduced checkpointing overheads.*

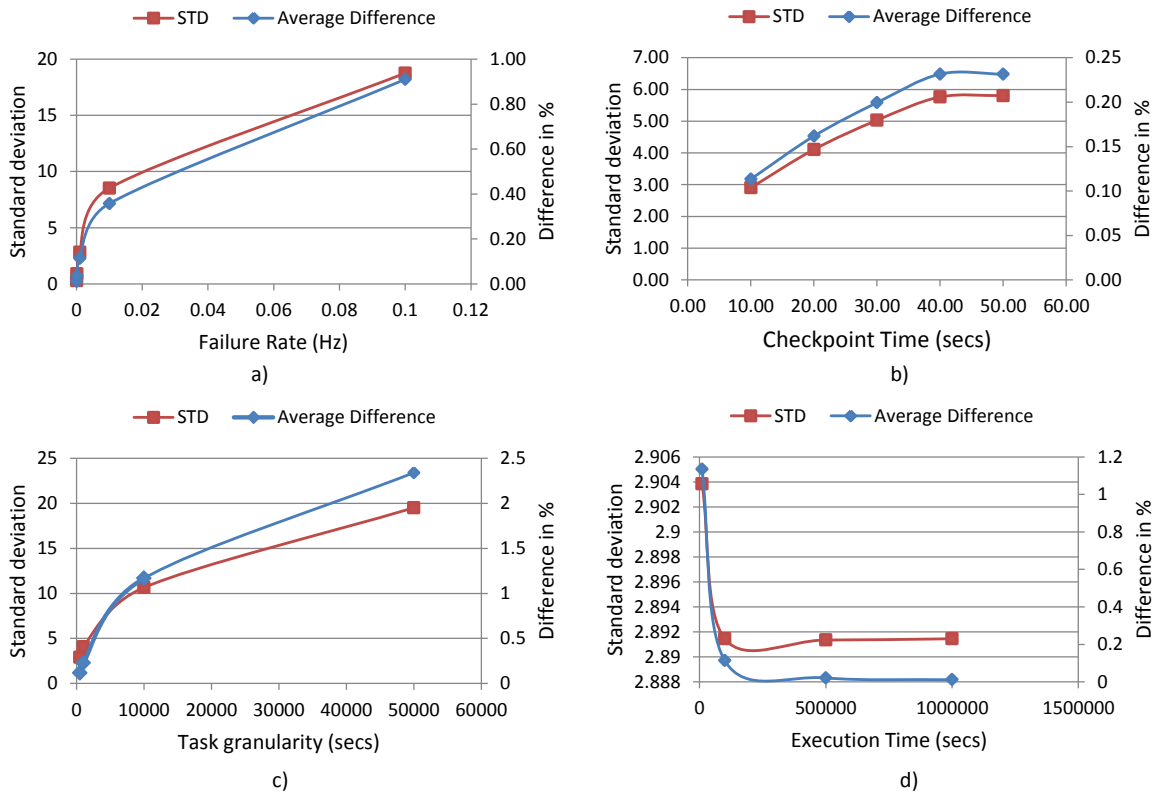


Figure 5.4: Model validation results

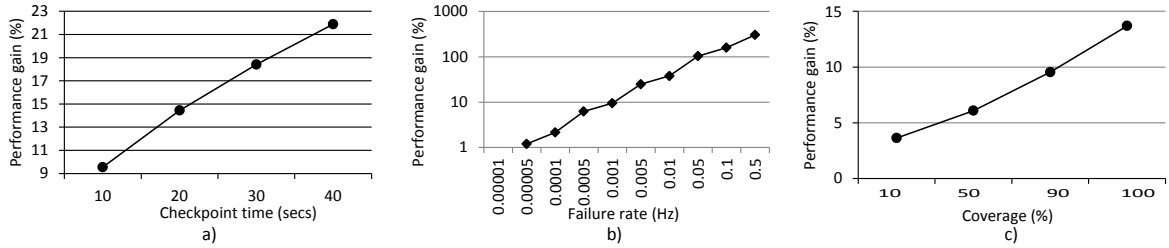


Figure 5.5: Performance gain percentages

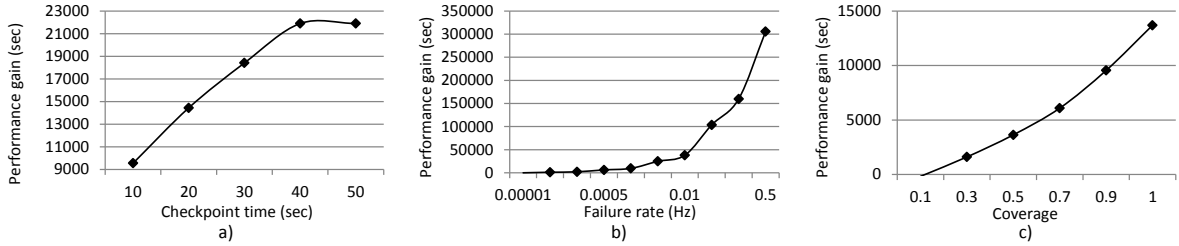


Figure 5.6: Performance gain amounts in seconds

Table 5.2: Benchmark Set

Linpack	Matrix size 131072, block size 256
Heat	Resolution 32768, 2 heat sources, Jacobi
Matrix multiplication (mxm)	Matrix size 9216x9216
Nbody	65536 particles
Specfem3d	Seismic wave propagation, 8x8 grid

Table 5.3: Coverage Coefficients: C factor (%)

HPL	mxm	Heat	Nbody	Specfem3d
87.3	91.31	97	98.7	97

Table 5.4: Walltime overhead (%)

HPL	mxm	Heat	Nbody	Specfem3d
7.25	6.01	1.03	0.32	0.3

5.5.3 Benchmark Evaluation Results

In this section we present the results for our benchmark set. We perform our experiments on Marenostrum III Supercomputer [90]. Table 5.2 summarizes our task-parallel distributed benchmarks [98]. In our experiments 1024 cores over 64 nodes are used. Our aim is to evaluate the unified model with real-world applications and assess the practical aspects of the model.

Table 5.3 shows the fraction of time that the application execution is in a task computation. This fraction corresponds to the C factor we defined in Section 5.3. This is important since the higher this fraction is, the more likely that task-level checkpointing will mitigate the failures affecting the application. As seen, for all benchmarks the fraction is high.

Table 5.4 presents the wall-clock time overheads of task-level checkpointing. For HPL and mxm, the overheads are relatively high due to communication across nodes. However, overall, all overheads are low and less than 8%. Note that in our model, these percentage overheads correspond to the W_{TL} values, total waste per unit of time of task-level checkpointing.

Figure 5.7 shows the performance gain of benchmarks when the combined model is used instead of system-wide scheme only up to 10 failures per minute (1/6 Hz.). First, we see that as the failure rates increase, so do the gains. Second, these gains are crucial given the expected

error rates for Exascale and extreme scale HPC systems. Third, the difference in behavior of benchmarks is due to the system-wide checkpoint times for each benchmark. Table 5.5 shows the measured system-wide checkpoint (or restart since they are very close) and task-level checkpoint (restart) times. With Figure 5.7 and Table 5.5 together, we see that the higher the system-wide checkpoint time is, the higher the performance gain.

Another observation is that compared to the γ factor (in Figure 5.8), the factor by which we decrease the system-wide checkpoint frequency, the system-wide checkpoint time is the dominating factor in terms of the performance gain. Even though the values of the γ factor are higher in some benchmarks, such as Nbody, their system-wide checkpoint times are low enough such that their performance gain is less than the gain of other benchmarks having higher system-wide checkpoint times. Moreover, the γ factor values converge to 2 since task-level failure coverages converge to 0.75 as the failure rate converges to 1/6 Hz.

Takeaway 3: *These experiments show that the performance gain is 282% of the application execution time on average and can be as high as 539%. Additionally, the decrease in checkpoint frequency, i.e. the γ values, is 5.3× on average.*

Table 5.5 shows the average task restart time and the FTI restart time. From these results we conclude that our model can provide orders of magnitude faster restart if combined with system-wide checkpointing. Our model uses in-memory task-level checkpoints rather than system-wide FTI checkpoints resided on the local disk, if not on the PFS.

Finally, we present our results in which we measure the peak memory sizes that task-level checkpointing incurs in Table 5.6 in a single node. We see that the memory sizes are reasonably low considering HPC systems' memory capacities.

Takeaway 4: *We conclude that the memory overheads of task-level checkpoints are low and become relatively insignificant considering the potential performance improvement that can be achieved with the unified model, while coping with the high failure rates anticipated for the Exascale era.*

5.6 Conclusion

In this chapter, we propose to unify task-level checkpointing, which we proposed in Chapter 4, with system-wide checkpointing for task-parallel HPC applications. We provide closed-form formulas for computing performance improvement and the optimal checkpoint interval of the unified model. Our simulation results show the validity of our model as well as the effects of

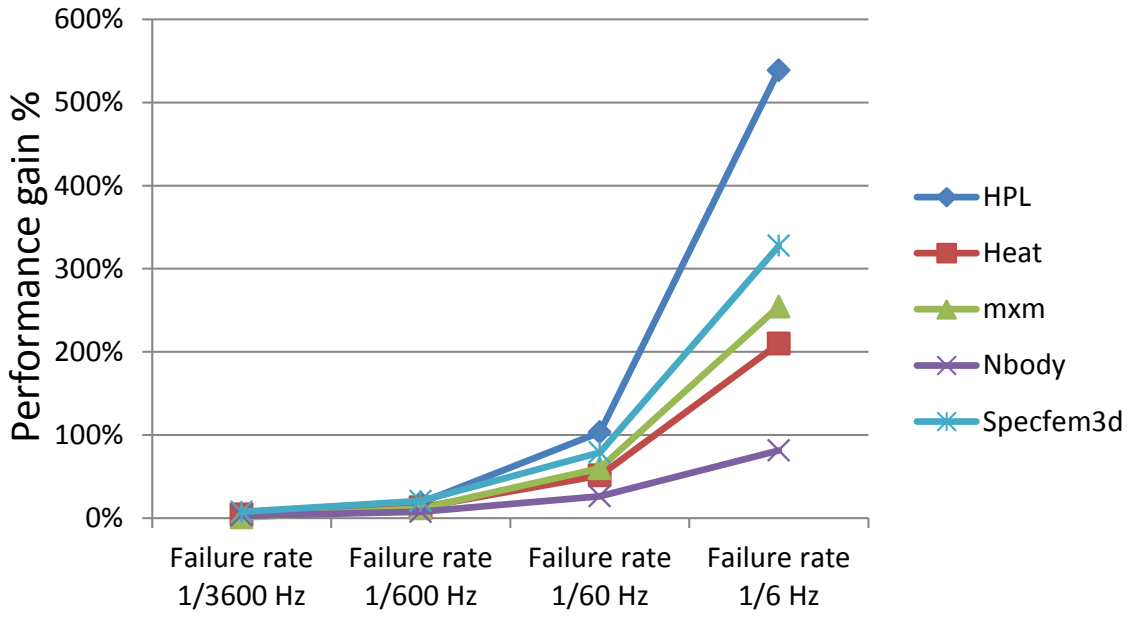


Figure 5.7: Performance gain of our benchmarks

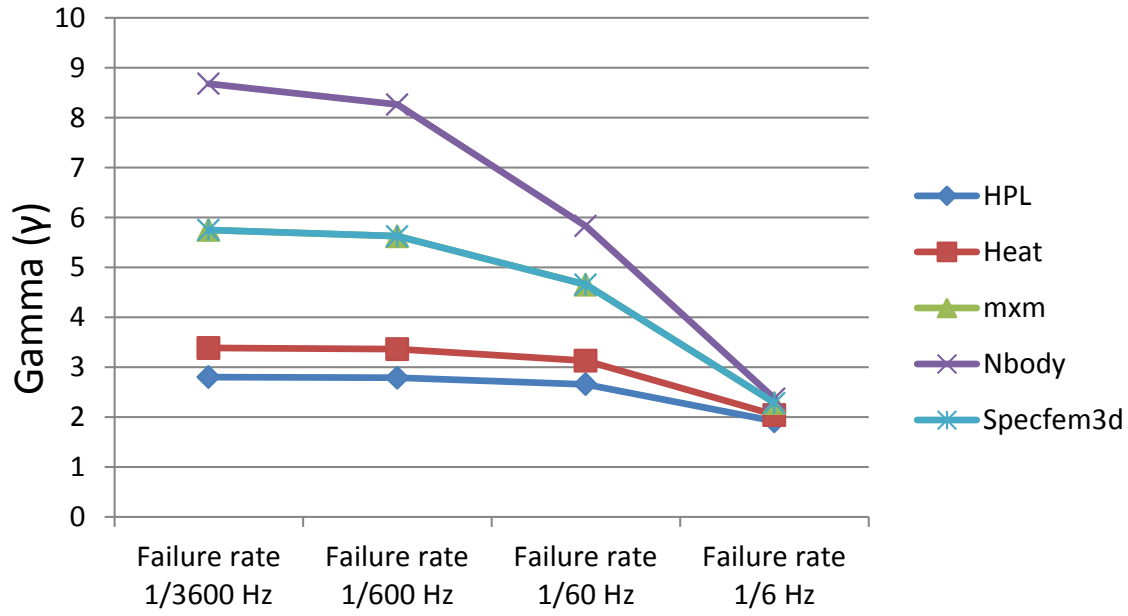


Figure 5.8: γ factor values of our benchmarks

Table 5.5: Checkpoint/Restart time of Tasks vs FTI

	Task restart time (sec)	FTI restart time (sec)
HPL	0.016	32.13
Heat	0.086	9.48
mxm	0.34	11.25
Nbody	0.08	2.28
Specfem3d	0.02	15

Table 5.6: NanoCheckpoints' peak checkpoint size

	Peak checkpoint size (MB)
HPL	34
Heat	128.5
mxm	386
Nbody	0.6
Specfem3d	38

parameters such as system failure rate and checkpoint time. Furthermore, results show that the unified model can decrease the overheads of system-wide checkpointing up to 539%. In the next chapter, we will broaden our scope to task-parallel message-passing applications.

6

Unified Fault-tolerance Framework for Hybrid Task-parallel Message-passing Applications

6.1 Introduction

In order to execute on large scale, task-based parallel models are usually combined with distributed memory PMs, such as the message passing PM. The hybrid MPI+OmpSs PM is an example of such integration. However, to the best of our knowledge, no resiliency solution has been proposed yet that is designed specifically for such hybrid models. In this chapter, we introduce such a solution. In particular, we extend our checkpoint/restart framework NanoCheckpoints (Chapter 4) for pure OmpSs applications with a message logging protocol adjusted to match the task-parallel execution model. The combined scheme is used to tolerate transient faults in the system and limits the consequences of such faults to the task that experienced them. It allows fast and asynchronous recovery that is more efficient than the con-

ventional full application rollback-restart. Our protocol has a negligible fault-free execution overhead and is highly scalable. NanoCheckpoints is implemented in the OmpSs runtime, and our message-logging protocol in the PMPI profiling layer. However, we point out that the combined protocol is applicable to any hybrid task-parallel message passing programming model that has a dataflow execution model.

In addition, we develop a mathematical model that unifies our fault-tolerant protocol with system-wide checkpointing. There is no previous work that studies and proposes task-level checkpointing with message logging on top of system-wide checkpointing. That is, there is no previous work that demonstrates how to set the checkpointing period of the unified scheme and whether or how much performance improvement will be gained if the unified scheme is adopted.

Moreover, other than being motivated by the lack of previous research, this unification has significant benefits and implications in terms of fault-tolerance. First, since our fault-tolerant protocol mitigates a fraction of failures, the total number of expensive system-wide checkpoints is reduced. Therefore, overall the checkpointing overheads are decreased. Second, failure recovery is mostly faster with the unified scheme thanks to the in-memory task checkpoints and message logs. Third, the unified model has a better failure containment, that is, the scope of failures becomes a single task rather than the complete application. Consequently, with the unified scheme the amount of lost computation is less than a system-wide only scheme. This is because in the unified scheme the amount of lost computation is the computation since the beginning of a task. This is typically much less than the amount of lost computation in a system-wide only scheme which is the computation since the beginning of the checkpointing period. With the unified model these benefits can be gained without sacrificing the complete failure coverage of system-wide checkpointing.

To the best of our knowledge, our model is the first that unifies task-level checkpointing and message logging with system-wide checkpointing. Moreover, we derive closed formulae for the optimal checkpointing interval and the performance score of the unified model. Results indicate that the performance gain (score) can be as high as 98% over system-wide-only checkpointing when the unified model is adopted. Our main contributions in this chapter are as follows:

- A scalable fault tolerance protocol for task-parallel message-passing applications for mitigating transient faults. The protocol has a 1.9% fault-free performance overhead on average and transparently handles MPI calls inside tasks in recovery. To the best of our

knowledge, this is the first resiliency solution for such a hybrid model.

- A mathematical model that unifies our fault-tolerant protocol with system-wide checkpointing.
- Closed formulae for the optimal checkpointing interval and the performance score of the unified model.
- An extended evaluation of fault-free execution, execution with faults, model validation and the performance score of the benchmarks.

The rest of this chapter is organized as follows. In Section 6.2, we discuss our deterministic model and application requirements for the proposed protocol. In Section 6.3, we move onto explaining the fault coverage. In Section 6.4 we describe the design of the proposed protocol. In Section 6.5 we present the unified model formalization. In Section 6.6, we evaluate our protocol in the fault-free execution and execution with faults. In addition, we validate our unifying model and evaluate the performance score of our model. In Section 6.7 we summarize the work in this chapter.

6.2 Deterministic model and application requirements

MPI+OmpSs hybrid model is a multithreaded dataflow-based execution model where any thread can execute communication calls at any time. Such an execution model implies a certain amount of nondeterminism as seen from the process's point of view.

Traditional models order events occurring during the execution using Lamport's *happened-before* relation [99]. This relation does not suit our execution model well, because the thread concurrency does not allow assumptions to be made about the order of events with respect to the process. For example, according to the MPI v3.0 standard [100], if two send operations are executed by two distinct threads concurrently, no assumption can be made about the relative order of completion of the two operations.

However, the distinctive property of the task-based execution model is that it is *dataflow driven*. This means that we can partially order events using the knowledge about task dependencies. We use the notion of the *always-happens-before* relation defined in [43] to do so. Briefly, always-happens-before relation orders the events not with respect to the program but with respect to different executions of the same program: if two events e_1 and e_2 are present

in all executions of a program and e_1 *happened-before* e_2 in all executions, then we say that e_1 *always-happens-before* e_2, or $e_1 \overset{A}{\rightarrow} e_2$.

Given two tasks A and B, let e_A and e_B denote any event in task A and task B, respectively. If task B is reachable from task A in the dependency graph (i.e., a path in the graph connects A with B), these tasks will always be executed in succession and, therefore, $e_A \overset{A}{\rightarrow} e_B$. If task B is unreachable from task A, however, no assumption can be made about the order of events because these tasks can be executed concurrently. For example, in the dependency graph in Figure 6.1 task 5 can run concurrently with task 2 since there are no dependencies between them; but task 4 depends on tasks 2 and 3, so in any execution any event of tasks 2 and 3 will always precede any event of task 4.

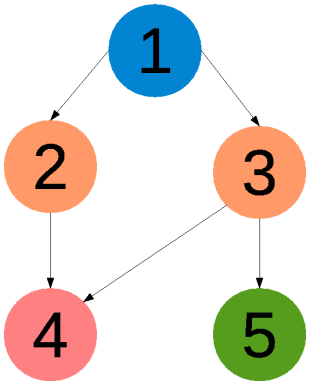


Figure 6.1: Example of a task dependency graph

The always-happens-before relation is typical for *channel deterministic* [43] applications. Channel determinism implies that the order of sends and receives in the application may change from execution to execution but, as long as the same set of messages is exchanged in any execution and the order of sends per channel is always the same, the application is channel deterministic. We cannot directly impose channel determinism on MPI+OmpSs hybrid applications because, as explained earlier, the thread concurrency introduces nondeterminism that may violate the requirement about the order of sends per channel.

However, because in this work we treat a task as the unit that may fail and be recovered, we will consider the deterministic model not from the point of view of the process but from the point of view of the task.

Therefore, as part of the proposed fault tolerance solution, we design a message logging protocol that can be applied to any MPI+OmpSs application in which all the tasks as stand-

alone units are channel deterministic.

6.3 Fault coverage

The extended protocol for hybrid MPI+OmpSs applications proposed in this work is used to handle transient faults that can cause detected uncorrected errors (DUE), such as, for example, multiple bit-flips that cannot be corrected by ECC. Such errors usually cause the system to raise an exception with consequent application crash (or abort) and restart from the latest checkpoint, in case checkpointing is provided. However, we assume that the runtime can catch such an exception and, instead of aborting the application, it tries to take recovery measures.

If the fault happened inside a task, the runtime can restart it from the checkpointed state of that task and use message logging to recover MPI communication state. We note, however, that our protocol can protect only tasks, therefore, any transient fault occurring during the execution of a nonprotected code, that is, nontaskified portion of code, will force the runtime to abort the execution or trigger the conventional application rollback-restart. Therefore, to maximize the benefit from the task recovery and avoid the whole application restart, it is better to taskify the application code as much as possible in order to reduce the ratio of protected/unprotected code.

6.4 Message logging protocol for MPI+OmpSs applications

To allow recovering a task with an MPI call inside we use a message-logging protocol. We opted for receiver-base logging because we can benefit from the fact that with transient faults in tasks we lose only the task-local data, such as results of its computations, but the process state data, such as message logs, persists because the process itself keeps running.

Algorithm 1 presents a pseudo-code of our protocol. Incoming messages are logged in the main memory during the execution of a task and are garbage collected after the task has successfully completed. Logs are kept in the context of a task in order to prevent other tasks from matching a message from the log by mistake.

We differentiate messages by their msg_id defined as a tuple $\{rank, tag, comm\}$. Here, $rank$ is the id of the process with whom this process communicates, tag is the MPI message tag, and $comm$ is the communicator. For every incoming and outgoing msg_id we keep a sequence number $seqnum$ which is incremented every time a communication involving this

Algorithm 1: Fault tolerance protocol for task A

```
1:  Upon starting task
2:      Checkpoint input parameters
3:      in_seqlist = out_seqlist = anonlist = NULL
4:  Upon finishing task
5:      Delete all logs and seqnum lists.
6:  Upon recovering task
7:      Restore input parameters
8:      for all seqnums in in_seqlist, out_seqlist do
9:          seqnum.current = 0 /* Reset current values for all seqnums */
10: Upon receiving
11:     if Anonymous recv & Recovering anonymous communications then
12:         /*Get next msg to match in recovery*/
13:         msg_id ← next_item(anonlist)
        /* Get msg_id's seqnum. If not found, create new and add to in_seqlist */
14:     seqnum ← find(in_seqlist, msg_id)
15:     if seqnum.current < seqnum.commited then
16:         seqnum.current ← seqnum.current + 1
17:         msg ← extract_log(msg_id, seqnum.current)
18:         if Anonymous recv & Reached the last item in anonlist then
19:             Finished recovery of anonymous communications
20:     else
21:         seqnum.current ← seqnum.current + 1
22:         seqnum.commited ← seqnum.current
23:         log(payload, msg_id, seqnum.current)
24:         if Anonymous recv then
25:             /*Log the order of anonymous receives*/
26:             add_item(anonlist, msg_id)
27: Upon sending
28:     seqnum ← find(out_seqlist, msg_id)
29:     if seqnum.current < seqnum.commited then
30:         seqnum.current ← seqnum.current + 1
31:         Skip sending the message
32:     else
33:         seqnum.current ← seqnum.current + 1
34:         seqnum.commited ← seqnum.current
```

msg_id takes place. The pair $\{msg\_id, seqnum\}$ can uniquely identify every received message and can be used to restore the order of receives for each channel. We keep two values: $seqnum.current$ for the current value and $seqnum.commited$ for the latest value of $seqnum$, both are incremented simultaneously in fault-free execution. If a task restarts, $seqnum.current$ of all sequence numbers is reset to zero (Line 9).

For anonymous receive calls, i.e. calls with `MPI_ANY_SRC` or `MPI_ANY_TAG`, we additionally store a list $anonlist$ of msg_id-s of messages that were received during the fault-free execution, so that in recovery we could receive them from log in the same order (Line 24).

In recovery, that is when $seqnum.current < seqnum.commited$ for current msg_id, in case of a send call, the process simply increments $seqnum.current$ and skips sending the message as it had been already sent before the fault (Line 29–31). In case of a receive call, it copies the message with the corresponding $seqnum.current$ value from the log to the receive

buffer (Line 15–17). If an anonymous receive call is executed in recovery, the protocol matches the next msg_id from $anonlist$ (Line 11). Once it matches the last item of the list, the recovery of anonymous communication is considered finished.

There is a trade-off in using the receiver-based logging that usually has a higher failure-free overhead than does the sender-based logging [101]. On the other hand, receiver-based logging allows us to perform better in recovery because the task will receive the message directly from the log and will not need to wait for it. Also, it simplifies garbage collection: we can delete logs immediately after the task finishes. Sender-based message logging would require the sender to keep the log until the corresponding task on the receiver has completed and, hence, may cause large memory usage during runtime.

6.5 Unified Model for Task-level Fault Tolerance and System-wide Checkpointing

We develop a mathematical model to combine task-level fault-tolerance (i.e., checkpointing and message logging) with a system-wide checkpointing scheme. This model handles distributed applications which the performance model in Chapter 5 cannot be applied to. If an error cannot be handled by task-level fault-tolerance, the whole application will have to roll back to the last system-wide checkpoint. Our model targets DUEs i.e. fail-stop errors.

Table 6.1: Model Parameters

Parameter	**Description**
τ_s	Checkpoint interval of system-wide checkpoints
c_s	Time to take a system-wide checkpoint
r_s	Time to restart with system-wide scheme
T_{ff}	Failure-free computation time without fault-tolerance
μ_s	Expected rate for fail-stop errors
$T_{overall}$	Total execution time including fault-tolerance
$CovTL$	Failure coverage of task-level fault-tolerance from fail-stop errors
T_{TL}	Total amount of time used for task-level fault-tolerance
T_{sys}	Total amount of time used for system-wide fault-tolerance
T_{wasted}	Total amount of time used for task-level and system-wide scheme
W_{TL}	Total waste per unit of time of task-level fault-tolerance
W_{sys}	Total waste per unit of time of system-wide scheme

Table 6.1 shows the model parameters. We first introduce the total time equations:

$$\begin{aligned} T_{overall} &= T_{ff} + T_{wasted} \\ &= T_{ff} + T_{TL} + T_{sys} && (6.1) \\ &= T_{ff} + (W_{TL} + W_{sys})T_{overall} && (6.2) \end{aligned}$$

The first equation shows the overall execution time which includes the wasted time due to task-level fault-tolerance and the wasted time due to system-wide checkpointing. The second equation shows the breakdown of the overall execution time with respect to the total waste per unit of time which we will minimize. We note that the *failure coverage* $CovTL$ refers to the fraction of time that a task-level scheme is able to recover from DUEs. It is the percentage of time task-level fault-tolerance successfully recovers from failures. $CovTL$ is experimentally obtained for an application. This is a significant difference from our first model in Chapter 5. Now failure coverage can be measured directly by experiments, which might reflect the application behaviour relatively more realistic.

In the next section, we detail the waste per unit of time of system-wide checkpointing, i.e. W_{sys}, and then the waste per unit of time of task-level fault-tolerance, i.e. W_{TL}. After that, we analytically find the optimal checkpoint intervals of the system-wide scheme and of the unified scheme. Finally, we conclude the model formalization with defining the performance score for an application using the unified model.

6.5.1 The wasted time of a system-wide scheme

The waste per unit of time of a system-wide scheme, i.e. W_{sys}, is the sum of the checkpointing, rework and restart overhead per unit of time. Checkpointing time is the overhead of taking checkpoints. The rework time is the lost computation from the last checkpoint to the moment of the failure. The restart time refers to the time to restore the latest checkpoint. We include down time, if any, in restart time since it does not require any special treatment. Now we formalize checkpointing, rework and restart overhead per unit of time or equivalently rates.

The checkpoint overhead per unit of time of a system-wide scheme, denoted by W_{syschk}, is the product of the time to checkpoint c_s and the number of checkpoints which is $\frac{1}{\tau_s}$. Hence the

checkpoint overhead per unit of time is

$$W_{syschk} = \frac{c_s}{\tau_s} \tag{6.3}$$

When a failure occurs, on average, half of the computation is lost since the last checkpoint. Thus, the expected value of rework overhead per unit of time of a system-wide scheme, denoted by W_{sysrew}, is

$$W_{sysrew} = \frac{\mu_s \tau_s}{2} \tag{6.4}$$

Restart overhead per unit of time, denoted by W_{sysres}, is the product of the failure rate and the time to restore the checkpoint. Thus,

$$W_{sysres} = \mu_s r_s \tag{6.5}$$

Therefore, the waste per unit of time of a system-wide scheme without task-level fault-tolerance is

$$W_{sys} = \frac{c_s}{\tau_s} + \frac{\mu_s \tau_s}{2} + \mu_s r_s \tag{6.6}$$

6.5.2 The wasted time of a task-level scheme

The waste per unit of time of a task-level scheme, W_{TL}, is constituted by the checkpoint overhead of all tasks, the rework and restart overheads of the failed tasks, and the message logging overheads of receiver tasks during the program execution. We note that our model assumes, like our protocol, a receiver-based message logging protocol. Let FTs be the set of the failed tasks during the program execution due to DUEs. Let RTs and STs be the set of the receiver and sender tasks of the program execution respectively (Note that a task can be both a sender and a receiver task). In addition, let α be a factor showing the fraction of the total wasted time of task-level fault-tolerance and recovery that is reflected in wall-clock time of the application execution. So for instance if $\alpha = 0$, then the application perfectly overlaps the computation with the task-level fault-tolerance and recovery and there is no overhead reflected in the wall-time of the application. On the other hand if $\alpha = 1$, then it means that task-level fault-tolerance is sequential and fully reflected in the wall-time of the application. Note that $0 <= \alpha <= 1$.

α captures the parallelism in application executions and scales down the total overhead to the reflected overhead in the wall-clock time.

The checkpoint overhead per unit of time, W_{TL}^{chk}, is the overhead of taking task-local checkpoints. Hence the checkpoint overhead per unit of time of task-level fault-tolerance is

$$W_{TL}^{chk} = \sum_{\forall i, Tk_i} Tk_i^{chk} \tag{6.7}$$

where Tk_i^{chk} is the checkpoint overhead per unit of time of task Tk_i.

The rework time for a single task is the computation lost since the beginning of the task. Thus the rework overhead per unit of time, W_{TL}^{rew}, of task-level fault-tolerance is

$$W_{TL}^{rew} = \mu_s\left(\sum_{Tk_i \in FTs} Tk_i^{rew} - \sum_{Tk_i \in STs \cap FTs} Tk_i^{noSend}\right) \tag{6.8}$$

where Tk_i^{rew} is the lost computation overhead per unit of time of the failed task Tk_i since the beginning of its computation. Tk_i^{noSend} is the overhead per unit of time saved because the recovering task does not need to re-send messages to non-recovering tasks.

The restart overhead per unit of time, W_{TL}^{res}, is the overhead per unit of time of restoring the task-local checkpoints. Thus the restart overhead per unit of time of task-level fault-tolerance is

$$W_{TL}^{res} = \sum_{Tk_i \in FTs} \mu_s \times Tk_i^{res} \tag{6.9}$$

where Tk_i^{res} is the restart overhead per unit of time of task Tk_i.

The message logging overhead per unit of time, W_{TL}^{log}, is the sum of the message logging overhead per unit of time of all receiver tasks. Hence this overhead per unit of time is

$$W_{TL}^{log} = \sum_{Tk_i \in RTs} Tk_i^{log} \tag{6.10}$$

where Tk_i^{log} is the message logging overhead per unit of time of task Tk_i.

Finally the waste of a task-level fault-tolerance scheme is

$$W_{TL} = \alpha(W_{TL}^{chk} + W_{TL}^{rew} + W_{TL}^{res} + W_{TL}^{log}) \tag{6.11}$$

In the next section we combine the two models.

6.5.3 The wasted time and the optimal checkpoint interval of the unified model

If task-level fault-tolerance has the failure coverage $CovTL$, then for system-wide checkpointing μ_s is decreased as

$$\mu_s' = (1 - CovTL) \times \mu_s \tag{6.12}$$

Let us denote the waste per unit of time of system-wide checkpointing under failure coverage $CovTL$ with $W_{sys}(CovTL)$. As a result, note that $W_{sys}(0)$ denotes the waste per unit of time of system-wide checkpointing without task-level fault-tolerance. The waste per unit of time of system-wide checkpointing under failure coverage $CovTL$ is

$$W_{sys}(CovTL) = \frac{c_s}{\tau_s} + \frac{\mu_s' \tau_s}{2} + \mu_s' r_s \tag{6.13}$$

Then the total waste per unit time of the unified model is

$$W = W_{TL} + W_{sys}(CovTL) \tag{6.14}$$

We now compute the optimal checkpoint intervals of both the system-wide only checkpointing scheme and the unified scheme. To find the optimal interval of the system-wide only scheme, we take the derivative of Equation 6.6 with respect to τ_s to and equate it to zero to find the global minimum. We compute it as

$$\tau_s^* = \sqrt{\frac{2c_s}{\mu_s}} \tag{6.15}$$

which is similar to the Young's formula [52].

The waste per unit of time of the task-level scheme W_{TL} is independent from τ_s of the system-wide scheme. That is, the task-level overheads and wasted time is independent from the checkpoint interval of system-wide scheme. W_{TL} then can be treated as a constant - with respect to τ_s - such that the total wasted time of the unified scheme in Equation 6.14 is still

a convex function and thus has a global minimum. The first derivative of Equation 6.14 with respect to τ_s is given by

$$\frac{dW}{d\tau_s} = \frac{\mu'_s}{2} - \frac{c_s}{\tau_s^2} \tag{6.16}$$

where the derivative of W_{TL} with respect to τ_s is zero. Setting the derivative to zero, the solution is

$$\tau^*_{unified} = \sqrt{\frac{2c_s}{\mu'_s}} \tag{6.17}$$

Hence, if the task-level scheme has coverage $CovTL$, then τ^*_s is increased by a factor of $\sqrt{\frac{1}{1-CovTL}}$, according to Equation 6.17. That is, the optimal solution of the unified model is

$$\tau^*_{unified} = \sqrt{\frac{1}{1-CovTL}}\sqrt{\frac{2c_s}{\mu_s}} \tag{6.18}$$

6.5.4 Performance Score of the Unified Model

The performance score of the unified model can be intuitively understood as whether the unified model provides any performance gain over system-wide-only checkpointing. If the score is positive, then there is some performance gain, otherwise some additional performance overhead is incurred. The latter can happen when the system-wide checkpointing overhead is very low and the task-level fault-tolerance overhead is high. However most often in the unified model, even though task-level fault-tolerance incurs overhead, the unified model increases the optimal checkpoint interval and thereby decreases the number of system-wide checkpoints. This effectively reduces the checkpoint overhead of the system-wide-only checkpointing scheme which is much higher than that of task-level fault-tolerance. This way the overhead of the unified scheme is reduced.

Formally, the performance score of the unified model can be computed by calculating the difference between the overhead of the system-wide only scheme and the overhead of the uni-

fied scheme. The *performance score* S of the unified model is

$$S = W_{sys}(0) - [W_{sys}(CovTL) + W_{TL}] \tag{6.19}$$

S can be further simplified as

$$S = (1 - \sqrt{1 - CovTL})\sqrt{2c_s\mu_s} + (CovTL)\mu_s r_s - W_{TL} \tag{6.20}$$

An important observation is that the performance score is increasing (linearly) with the failure rate. This is especially critical considering the expected increase in failure rates in the exascale era.

6.6 Evaluation

In this section, we first present the experimental setup and our benchmark set. We next discuss the scalability, performance, and memory overheads of the proposed protocol in fault-free execution and in the presence of faults. Finally we provide the experimental evaluation of the proposed unified model.

6.6.1 Experimental setup

We conducted experiments with three hybrid MPI+OmpSs benchmarks[1]: Himeno, which performs the incompressible fluid analysis; Nbody, which simulates a dynamical system of particles; and a matrix multiplication benchmark. The details of the input parameter sizes are given in Table 6.2.

Table 6.2: Benchmark parameters

Matrix multiplication (mxm)	Matrix size 32,768
Nbody	524,288 particles
Himeno	1408 x 704 x 704

The experiments were carried out on the MareNostrum III Supercomputer [90].

All the tests were run with 64 MPI processes, one process per node and 16 threads per process. All the overheads are computed against the pure execution time without our protocol.

[1] Application repository: https://pm.bsc.es/projects/bar/wiki/Applications

6.6.2 Fault-free execution

Table 6.3: Memory overhead

	Checkpoint Size (MB)	Total Message Log (MB)	Peak Message Log (MB)
mxm	512	8064	128
Nbody	0.59	0.75	0.25
Himeno	1202	448	1.20

Table 6.4: Task statistics with corresponding fault rates (faults/sec)

Himeno	Total 600 tasks				Nbody	Total 383 Tasks				mxm	Total 126 Tasks			
Fault probability	10	25	50	75	Fault probability	10	25	50	75	Fault probability	10	25	50	75
Avg. recovered tasks	60	151	298	449	Avg. recovered tasks	43	104	193	287	Avg. recovered tasks	12	32	63	93
Fault rate	1.59	2.82	4.24	5.44	Fault rate	0.30	0.64	0.99	1.27	Fault rate	0.16	0.31	0.53	0.76

First, Figures 6.2 and 6.3 show the impact of our protocol on the original strong and weak scalability efficiency of the applications. We measured the scalability efficiency in the following manner. If t_1 is the time to execute a unit of work on one process and t_N is the time to execute the same amount of work on N processes, then the strong-scalability efficiency is computed as $S_{strong} = \frac{t_1}{N t_N} * 100\%$.

Respectively, if t_1 is the time to execute a unit of work on one process and t_N is the time to execute N units of work on N processes, then the weak scalability efficiency is computed as $S_{weak} = \frac{t_1}{t_N} * 100\%$.

The tests show that checkpointing and message logging may decrease the scalability to different extents depending on the application. The impact for the matrix multiplication benchmark was the biggest in both cases. Its strong scalability with 64 processes dropped by 24% because at larger scale the input data size per process was so small that the checkpointing and message logging overheads became more noticeable and difficult to hide by task overlapping. This was not the case in the weak scalability test, so the scalability dropped only by 2.7% at a respective scale. The scalability of Nbody and Himeno did not change significantly.

Next, the runtime overhead of using our protocol measured as 4.45% for the matrix multiplication application, 0.31% for Nbody and 0.89% for Himeno (Note that these overheads

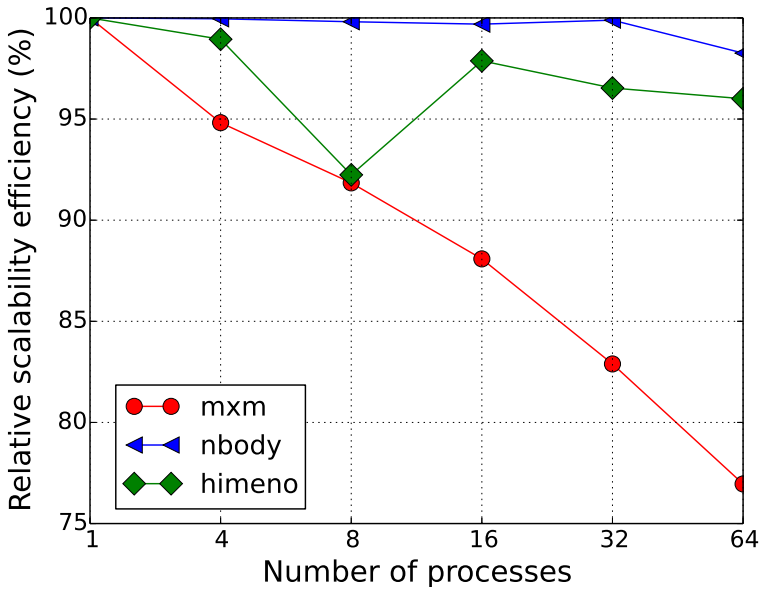


Figure 6.2: Relative strong scalability of the fault tolerance execution compared with nonresilient execution

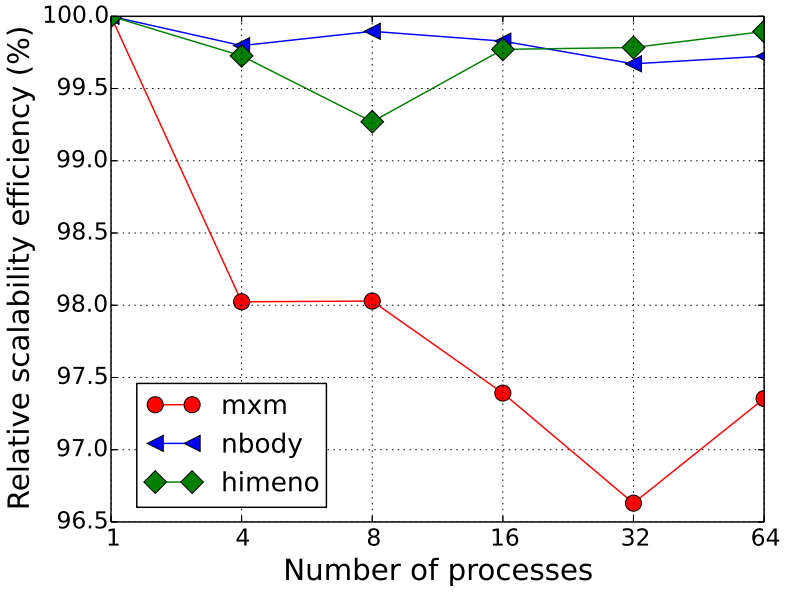


Figure 6.3: Relative weak scalability of the fault tolerance execution compared with nonresilient execution

correspond to W_{TL} in Equation 6.11 in our model.). To understand these results, one needs to look at the memory overhead of the protocol presented in Table 6.3.

Column one of Table 6.3 shows the total memory size that was occupied by NanoCheckpoints: 512 MB for the matrix multiplication, around 1 GB for Himeno, and less than 1 MB for Nbody. The size of a checkpoint depends on how the program was taskified. For instance, Nbody has a small checkpoint size because every process works only on the portion of particles that it is responsible for. In Himeno, on the other hand, additional arrays hold the data, and whole arrays are passed to the tasks; therefore, more data is checkpointed.

The total amount of logged messages is presented in the column two: 448 MB for Himeno, 7.8 GB for the matrix multiplication benchmark and less than 1 MB for Nbody. We note that Nbody is a computation-intensive application in which communication occurs only when the processes exchange particle data at the end of a step; therefore, the message log is small for this application. We tried to increase the input number of particles for Nbody application to see whether the checkpointing and message logging overheads would noticeably rise, but they did not grow significantly compared with the greatly increased compute time. Therefore, we decided to leave the current input parameter size.

Analyzing the numbers, we concluded that message logging was the main contributor to the execution overhead. In particular, the results for the matrix multiplication application show a lot of communication, which is clear from the total size of all logged messages.

Finally, we point out that the memory occupied by the protocol at any given moment is much smaller because message logs are kept only until the corresponding task is finished. Column three in Table 6.3 shows the peak message log size per process observed during the whole execution in all tests. As one can see, the peak log size is much smaller than the total message log size.

6.6.3 Execution with faults

To evaluate how much time it would take for an application to finish when transient faults are present, we simulate faults in the following manner. We first set a parameter for probability of a task failure. At the end of a task, a random number between 0 and 100 is generated. If the number is less than the probability of a failure, we consider that a fault has occurred, restore the task input parameters, and re-execute it. For example, if the probability is set to 50%, we expect approximately half the tasks to re-execute.

Because we simulate faults in the end of the task, we imagine the worst-case scenario and measure the upper bound for the execution time with failures.

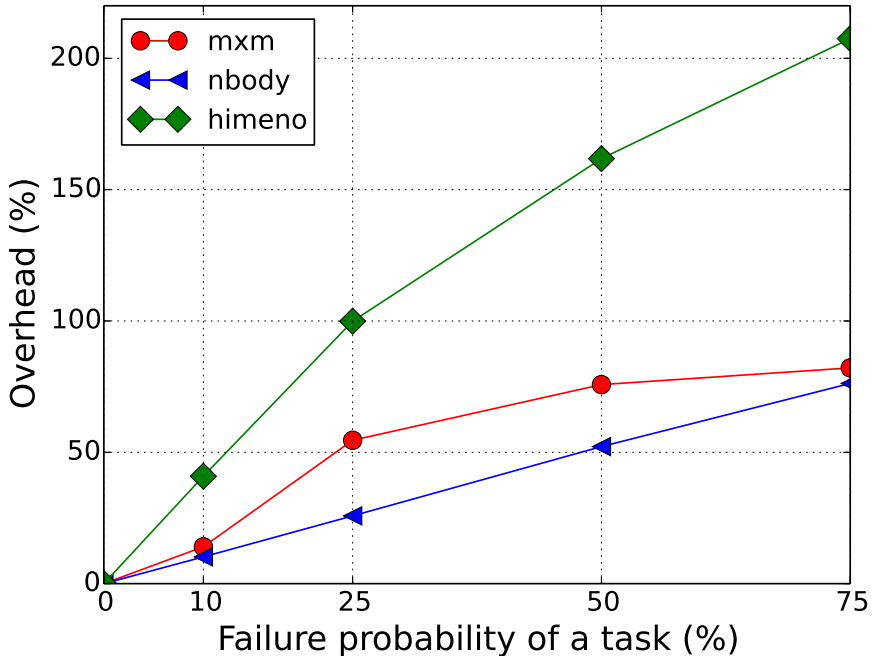


Figure 6.4: Runtime overhead in the presence of faults

Figure 6.4 shows the benchmark execution overhead for different failure probabilities. Additionally, Table 6.4 gives a general idea about the total number of tasks executed by each process and the relation between the probability of a task failure and the actual number of failed tasks and corresponding fault rates.

Nbody has a linear dependency between the number of task re-executions and the program complete time. The reason for such a correlation is that Nbody is tightly coupled: In each step, processes must exchange particle information; therefore, if one task from the step has to re-execute, all the other processes that have already finished computing for this step will have to wait for the delayed process.

In the matrix multiplication benchmark, recovering a quarter of the total number of tasks delays the execution by 50%. Studying the source code of the program suggested that this benchmark does not have fine task granularity. Coarse task granularity along with tight data

dependency results in worse parallelization, and hence less overlapping with task recovery is possible because dependent tasks can not be executed until the current one recovers and finishes. On the other hand, when there are more independent tasks, they can be executed while the failed task is recovering to fill in the gaps and hide the time lost on task recovery.

Similar effects of the coarse granularity and task dependency were observed in the tests for Himeno: recovering about half of the tasks delayed the execution by 1,5 times. After examining the source code we noticed that whole arrays of data are passed in virtually all taskifying pragmas in Himeno. Hence, all tasks become tightly dependent on each other; and if one task has to re-execute, no other tasks will be able to make other threads busy. Everything will have to wait for the recovery of this one task.

6.6.4 Unified Model Experiments

In this section, we first present experimental results regarding the validation of our model. Then we study the failure coverage of task-level fault-tolerance which is needed for the calculation of the performance score of the unified scheme. Finally, we present the performance score experiments of our model.

Model Validation:

We validate our model through Monte Carlo simulations. To perform simulations, we implement a simulator that generates failure sequences where failure arrivals follow Weilbull distribution. In our experiments even though we study different Weilbull distributions, we report realistic scenarios where the shape parameter is 1 (exponential distribution) and the scale parameter is the number of tasks to have reasonable failure distributions in the simulations. We feed the model and the simulator with the same parameter values. We set parameter values to be on par with the predictions of studies [3] and [97]. We compare the total execution time predicted by our model to the total execution time of simulations in the presence of failures. We perform 10000 simulations for each experiment.

Figure 6.5 shows the model compared with the simulations under different parameters. On the left, the x-axis shows the average absolute difference in total execution time as the system-wide checkpointing time (y-axis) changes. As checkpointing time increases, the divergence between the model and the simulation increases. However as seen, the deviation is very low and less than 0.7%. The relation between checkpointing time and the deviation can be due

to the tacit assumption that no failures are expected to occur during the checkpointing itself however, in simulations this assumption may - though rarely - not hold.

On the right, again the x-axis shows the average absolute difference in total execution time and y-axis shows the task-level failure coverage. As seen, as the task-level coverage increases, the deviation between the model and the simulations decreases. This is because the task-level coverage reduces the failure rate of the system which is positively correlated with the deviation between the model and the simulation. This positive correlation stems from the fact that the approximations in the model, such as Equation 6.4, become less accurate as the failure rate increases. From the figure, we can see that the deviation is always less than 0.35%. We omit the effects of other parameters - which are similar - for brevity.

Overall, validation experiments demonstrate that our model accurately reflects the real-world behaviour of the unified checkpointing system.

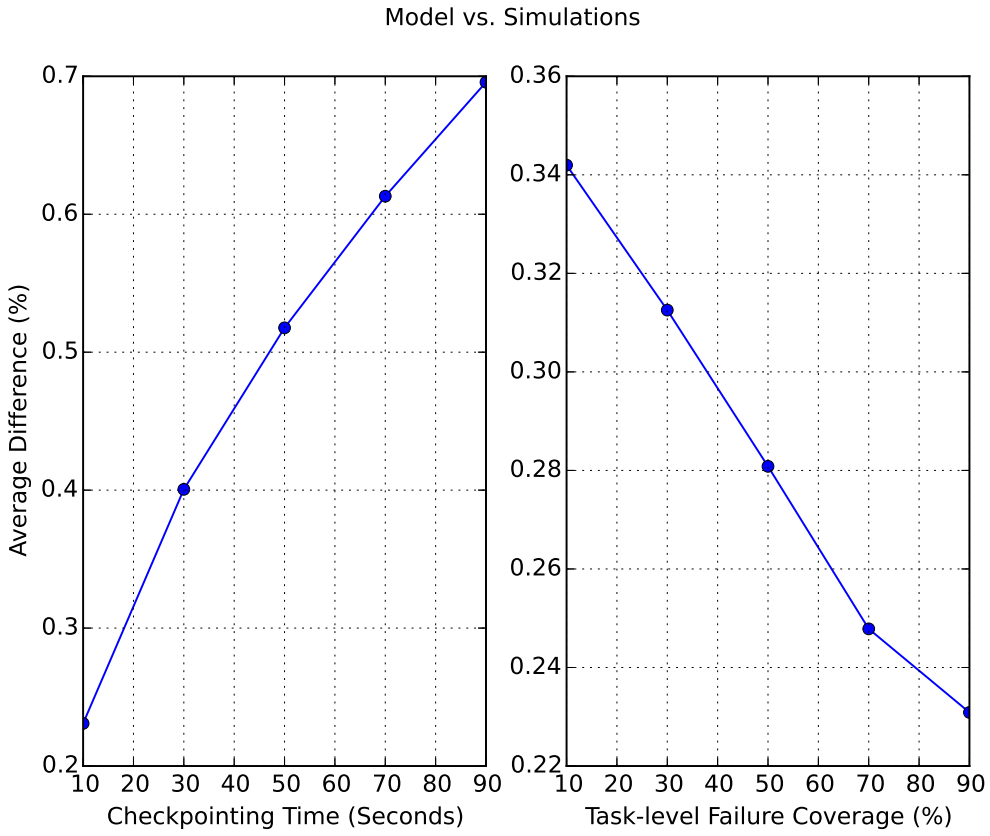


Figure 6.5: Model validation results

Failure Coverage of Task-level Fault-tolerance:

To see the failure coverage of task-level fault-tolerance we have performed fault injection experiments. In these experiments each workload was executed 10000 times with task checkpoints enabled. During each workload execution only one fault is injected into the dataset of the application. The fault injection occurs randomly both in time and in the space. The moment when a fault is injected is a random event with exponential distribution. The memory where a fault is injected is random event with uniform distribution. Depending on how the fault is injected a workload may crash (segmentation fault), complete with a wrong result, or complete with a correct result. In these experiments we compare how many times the executions of a workload complete successfully. The more successfully completed executions the better coverage. Table 6.5 shows the failure coverage of task-level fault-tolerance for each benchmark, that is, the percentage of time task-level fault-tolerance successfully recovered from the injected faults. As seen, the failure coverage of task-level fault-tolerance is high for all benchmarks showing its effectiveness.

Table 6.5: Task-level failure coverages of benchmarks

	Task-level failure coverage (%)
mxm	98%
Nbody	73%
Himeno	86%

Performance Score Experiments:

In order to assess the performance score of the unified model for the benchmarks under different failure rates, the system-wide checkpointing times are needed by Equation 6.20. We use FTI [38] checkpointing library as the system-wide checkpointing scheme to measure checkpointing times. We perform experiments to obtain the FTI checkpointing time of benchmarks. Table 6.6 shows the system-wide checkpointing times of benchmarks per process when running with a total of 64 processes. Figure 6.6 shows the performance score of the unified model (over system-wide-only checkpointing) for the benchmarks under different failure rates. We see that Nbody and Himeno almost always has performance gain with the unified model and matrix multiplication has performance again after the failure rate is 10^{-3} Hz (1 failure every 1000

Table 6.6: FTI Checkpointing time of benchmarks

	System-wide checkpointing time (seconds)
mxm	1.92
Nbody	40.44
Himeno	45.79

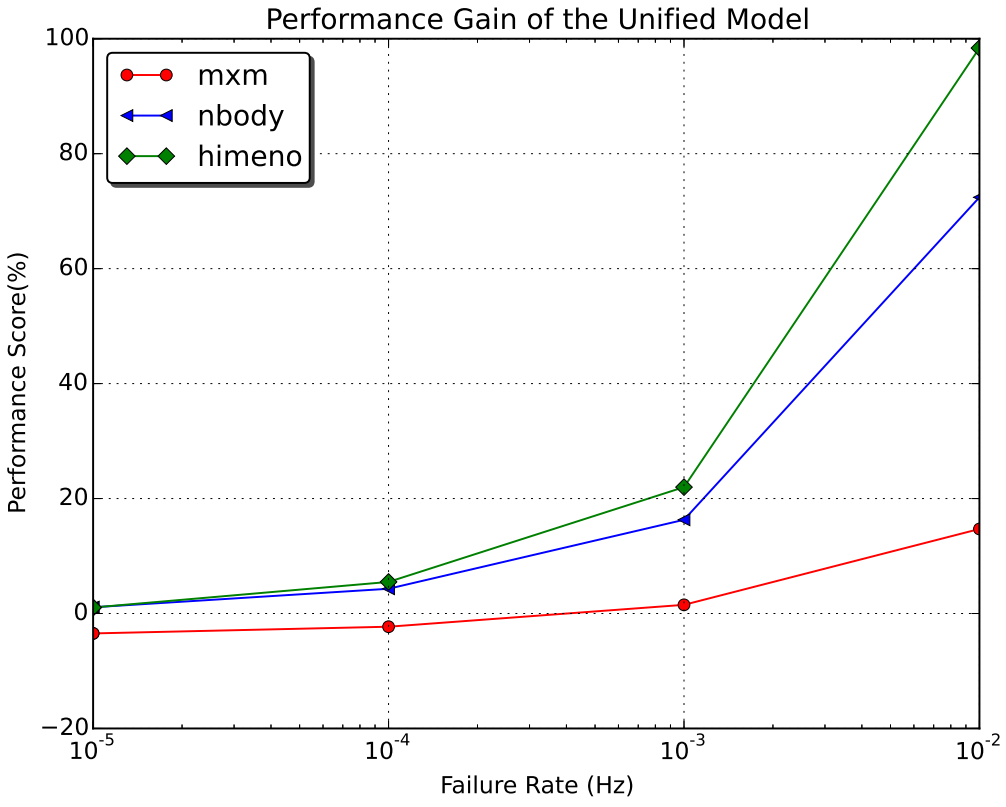


Figure 6.6: Performance score results

seconds). As seen, only matrix multiplication has a relatively low system-wide checkpointing time. As discussed before in matrix multiplication, the per process input data size is very small which leads to low a system-wide checkpointing time.

Another important observation is that as the failure rate increases, so does the performance score of the unified model. This is especially important in the exascale era since the failure rate are expected to increase dramatically.

6.7 Conclusion

In this chapter we proposed a unified framework that provides fault-tolerance for hybrid task-parallel message-passing HPC applications [102]. Our framework can be adopted for any dataflow task-based message-passing programming model. With our framework this study had two main contributions.

We first introduced a message logging protocol that, combined with task checkpointing, provides a resiliency solution for mitigating fail-stop errors (DUEs) in task-parallel message-passing applications. The protocol allows to avoid the costly application rollback-recovery by asynchronously restarting only tasks in which a fault has occurred and transparently resolves recovery of tasks that have MPI calls inside, thanks to the message logging.

Evaluation of the execution in the presence of faults showed that task granularity and coupling play an important role in hiding task recovery. The more the number of tasks that can be executed independently while some other task is recovering from a fault, the less the impact faults will have on the total execution time. If the program was not taskified well, however, recovery of even one task may slow the program significantly.

Second, we introduced a unified model that combines task-level fault-tolerance with the system-wide checkpointing. We analytically derived the optimal checkpointing interval (Equation 6.18) and the performance score (Equation 6.20) of the unified model. Moreover, our results show that with the unified scheme, the performance gain can be as high as 98% over system-wide-only checkpointing.

Starting with the next chapter, we will consider silent data corruptions (SDCs) or silent errors in addition to fail-stop errors, e.g. DUEs. To handle SDCs, we will propose selective task replication mechanisms.

7

Replication for Task-parallel 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 [2] 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 (Chapter 4) 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 chapter we combine redundant computation - in our case replication of application tasks - and

checkpoint/restart to address SDCs and fail-stop errors of task-parallel HPC applications while increasing reliability.

The straightforward way to address SDCs and fail-stop errors is to replicate all application tasks, that is, complete task replication[1]. However complete task replication may be prohibitive due to the extremely high resource cost and in fact not needed due to the uneven susceptibility of the different program parts to SDCs [65]. Therefore effective and efficient techniques are needed to selectively replicate tasks. In this chapter, we provide four different runtime-based techniques to selectively replicate tasks. We start with two relatively less formal techniques.

7.1 Automatic Risk-based and Programmer-directed Partial Redundancy for Resilient HPC

7.1.1 Introduction

In this section, we introduce two partial task redundancy (replication and checkpointing) schemes to reduce resource costs while providing sufficient level of fault-tolerance.

First, we develop an effective and efficient runtime heuristic that selectively replicates the most reliability critical tasks for a reasonable trade-off between performance and error coverage. The heuristic selects tasks based on their risks solely utilizing existing knowledge at runtime. Therefore no additional bookkeeping is added to the runtime. In Section 7.1.2 we discuss how the risk metric is established through *component analysis*. Our heuristic is automatic and completely transparent to user/programmer, application and OS. Moreover, it is very lightweight having negligible performance overhead. Our motivation is to design a heuristic that is as effective, efficient and low-cost as possible while not requiring to keep any extensive information at runtime or any expensive offline application profiling.

Second, we implement a programmer-specified partial redundancy mechanism where the programmer knowledge is leveraged. We therefore extend the programming model minimally with a `protect` clause to enable the programmer to provide task criticality hints to the runtime. For the programmer-directed partial mechanism we explain how we analyzed our benchmarks to select the more reliability critical tasks. The results indicate that with selective task protection, we increase the application reliability efficiently. In addition, to our surprise, we

[1] We use replication to refer to replication and checkpoint/restart together.

find that our automated runtime heuristic performs as well as the programmer-directed mechanism.

Finally, we compare both our heuristic and programmer insights to the baseline where tasks are selected for replication randomly. Results indicate that both are significantly more effective and efficient than random task selection.

Our main contribution in this section is the development of a partial task redundancy framework for task-parallel dataflow HPC applications. To the best of our knowledge, this is the first work that explores the risk-based automatic and programmer-directed selective task redundancy for task-parallel data-driven programming. Our main contributions in this section are:

- A runtime heuristic that automatically selects the most reliability critical tasks for partial protection providing around 70% detected and corrected errors with only 5% overhead on average. Moreover our heuristic itself has negligible performance overhead.
- For the runtime heuristic, component analysis for each benchmark to identify the most reliability-critical tasks or task components.
- A programmer-directed mechanism enabling programmer hints to be propagated to the runtime for efficient partial task protection providing around 65% detected and corrected errors with only 4% overhead on average.

7.1.2 Design of Our Framework

This section first presents our fault injection mechanism. It then details the baseline and the selective task replication and checkpointing designs. We use *task redundancy* to refer to task replication and checkpointing together.

Fault Injection Mechanism

We implement our fault injection mechanism to evaluate task redundancy and to have different injection capabilities to corrupt the task state at runtime to assess our partial task protection schemes. We inject faults in the state of the tasks by flipping bits of the outputs. We support two different injection settings where fault injection i) is based on fixed fault rates per task or ii) is based on the application execution time and input size. Injection is performed at the end of the execution of a task based on the chosen setting. In a chosen setting, if a fault is to be

injected to the task then we flip as many bits as specified beforehand. Each flip is a random bit of a random element of the task output. In our experiments fault arrivals follow an exponential distribution for the second setting - in the first fixed fault rate setting, it is not possible to enforce a specific distribution.

Complete 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. As stated earlier, the performance of OmpSs+Nanos is on par with the highly optimized commercial and open source implementation of OpenMP [23], [25].

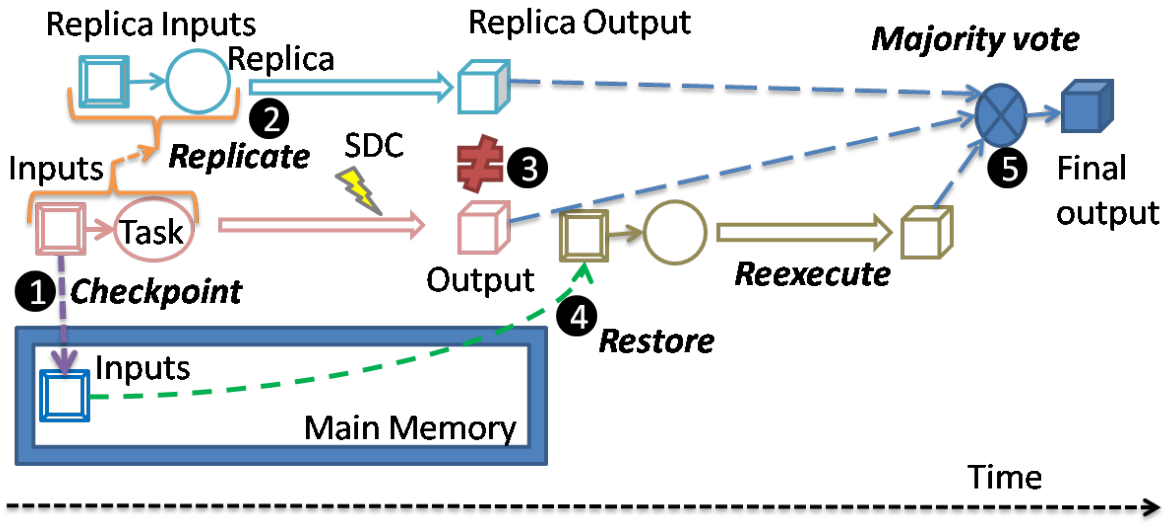


Figure 7.1: Task replication design: SDC mitigation

Baseline Task Replication Design: Figure 7.1 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 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.

Risk-based and Programmer-directed Task Redundancy Design

The baseline task redundancy design protects, *i.e. duplicates and checkpoints*, all tasks in a program. However in the cases where the protection of all tasks is not needed, partial task protection can provide both required level of fault-tolerance and reduce costs such as power, energy, memory usage. Thus we design and implement two different mechanisms for selective task protection. In the automated mechanism, we implement a runtime heuristic that automatically selects the tasks to be protected according to our empirically determined criticality measure. It is transparent to the user/programmer since no annotations or modifications to applications are required. In the programmer-directed mechanism, the programmer can give hints to the runtime in terms of which tasks to protect.

Runtime-directed Selective Redundancy

We have developed a runtime heuristic that automatically protects tasks in terms of their criticality defined by a risk function. The risk of a task t is defined to be:

$$risk(t) = (w_i \times ISize(t) + w_o \times OSize(t)) \times w_s \times Succ(t) \tag{7.1}$$

where $ISize(t)$ is the size of the inputs of t, $OSize(t)$ is the size of the outputs of t, $Succ(t)$ is the number of the immediate successors of t in the task dependency graph. The coefficients or weights w_i, w_o and w_s can be determined by *component analysis* which we will elaborate below.

The rationale behind the risk definition is to consider memory usage and task dependency relation. The size of inputs and outputs is a good hint for the memory space used by tasks.

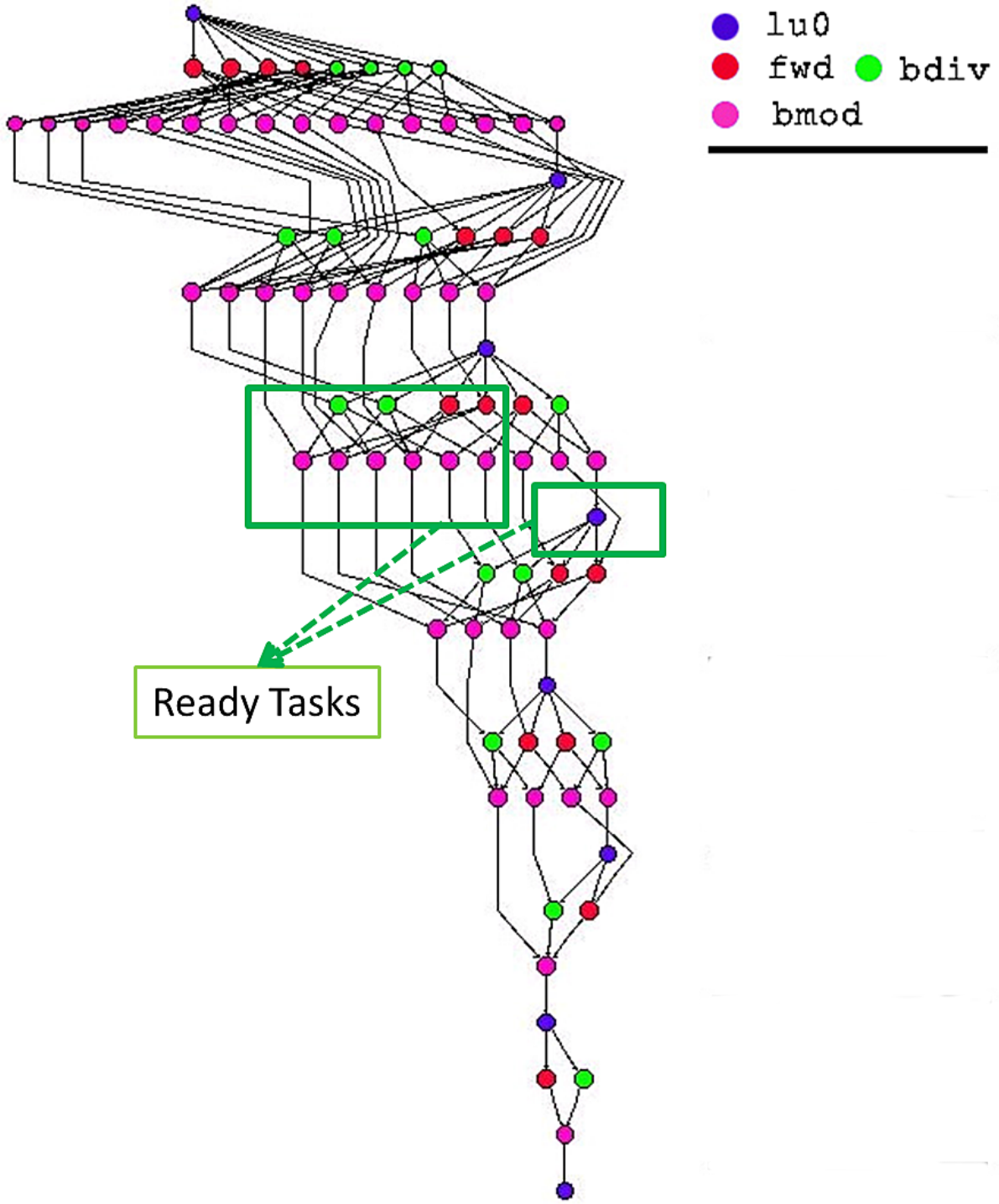


Figure 7.2: Sparse LU task dependency graph

Moreover, for parallel applications it has been established that the probability of a fault occurring in a task is associated with the memory space it uses [65]. For this reason, we use task input and output sizes in the risk function. In addition, the risk of a task is positively correlated with the number of its immediate successors as a fault occurring in a task might affect a large number of tasks if it has a large number of immediate successors. We compute the risk of a task without the need for a profiling run since all the necessary information above is available at runtime when a task is ready for execution.

To be able to automatically and dynamically select a task, our mechanism maintains a *global task risk*. Each time a decision is made about whether to protect a task, we update the global task risk as 70 percent of the current global risk value plus 30 percent of the risk of that task. These percentages were determined empirically. We decide updating the global task risk this way to provide some damping when a task with a much higher risk is encountered. Our heuristic computes the task's risk and compares it against the global tasks risk. If the risk of the task is smaller than the global risk then the task is not protected. Otherwise the task is protected. Note that our heuristic only maintains a global variable and performs a simple risk calculation for each task. As a result, it has no significant overhead.

As our running example, Algorithm 2 depicts Spare LU factorization which is one of our benchmarks which are included in Table 5.2. It has four phases - *lu0*, *fwd*, *bdiv* and *bmod* - and ported to OmpSs simply by annotating the inputs and outputs of the functions. A is the input matrix and NB is the number of the blocks determining the granularity of tasks. Figure 7.2 shows the complete color-coded task dependency graph of a Sparse LU computation. The rectangles on the graph show the set of ready tasks to be executed at some particular time. We want to compute the risks of different types of task instances. Let us take four tasks that are instances of *lu0* having 4 immediate successors, *fwd* having 3 immediate successors, *bdiv* having 3 immediate successors, and *bmod* having 1 immediate successor respectively among the ready tasks. Let the block size be BS. Then each task argument's size is BS^2. Let $X = BS^2$. The *lu0* has an input, *fwd* and *bdiv* have an input and an output, and *bmod* has two inputs and an output. In addition as we will see shortly, the weights w_i and w_o are roughly twice as w_s. For simplicity, let $w_i = 2$, $w_o = 2$ and $w_s = 1$. Then according to Equation 7.1, the risk of the *lu0*, *fwd*, *bdiv* and *bmod* task is $8X$, $12X$, $12X$ and $6X$ respectively. This shows that the instances of *fwd* are *bdiv* more risky/critical than the those of *lu0* and *bmod* at that particular time.

Component Analysis:

To analyze the effect of different task components such as the size of inputs or outputs, the

Algorithm 2: Sparse LU Algorithm

```
   Input: A: Input matrix to be factorized into LU.
   Input: NB: The number of blocks tiling the input matrix.
 1 for (kk = 0; kk < NB; kk++) do
 2     #pragma omp task inout(A[kk][kk]) if (kk < NB/2) then protect
 3     lu0(A[kk][kk]);
 4     for (jj = kk + 1; jj < NB; jj++) do
 5         if (A[kk][jj] != NULL) then
 6             #pragma omp task input(A[kk][kk]) inout(A[kk] [jj]) if (jj < (NB + kk)/2) then protect
 7             fwd(A[kk][kk], A[kk][jj]);
 8         end
 9     end
10     for (ii = kk + 1; ii < NB; ii++) do
11         if (A[ii][kk] != NULL) then
12             #pragma omp task input(A[kk][kk]) inout(A[ii][kk]) if (ii < (NB + kk)/2) then protect
13             bdiv(A[kk][kk], A[ii][kk]);
14         end
15     end
16     for (ii = kk + 1; ii < NB; ii++) do
17         for (jj = kk + 1; jj < NB; jj++) do
18             if (A[kk][jj] != NULL) then
19                 if (A[ii][jj] == NULL) then
20                     A[ii][jj] = allocate_clean_block();
21                 end
22                 #pragma omp task input(A[ii][kk], A[kk][jj]) inout(A[ii] [jj])
23                 if (ii < (NB + kk)/2&jj < (NB + kk)/2) then protect
24                 bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
25             end
26         end
27     end
28 end
29 #pragma omp taskwait
```

number of inputs, outputs, immediate successors or predecessors of a task, we use a method that we call *component analysis*. We use this method to determine the weights in the risk definition in a systematic manner. We set the risk of a task to be one of these components. We run fault-injected experiments with our fault injection mechanism in which we measure the time overhead and the number of corrupted entries in the final program output for each component. We then define *relative effectiveness (RE)* to evaluate which component is effective in terms of overhead and corruption in the program output as follows:

$$RE(c_1, c_2) = \frac{overhead(c_2)}{overhead(c_1)} \times \frac{errors(c_2)}{errors(c_1)} \tag{7.2}$$

where $overhead(c*)$ is the time overhead of the redundant execution and $errors(c*)$ is the number of corrupted elements in the final output when the component $c*$ is set to be the risk of the task and our heuristic is enabled. If $RE(c_1, c_2) > 1$, then c_1 is more effective than c_2. Larger

value implies c_1 has a clearer advantage in terms of providing good tradeoff between performance and coverage. With this, we proceed as follows: We calculate the relative effectiveness for every pair of components for each benchmark. We take the average for each pair across the benchmarks. Using the order of averages, we sort the components in terms of effectiveness. Then, we normalize averages and calculate weights in the risk formula. Our experiments show that components such as the number of inputs and outputs are almost equally effective as the size of inputs and outputs. Hence, we dismiss them in the risk formula above. For our risk definition, the weights are found to be $w_i = 2.03$, $w_o = 2.71$ and $w_s = 1.32$. In this work, *we only use this method to tune the risk definition*. As future work, we consider to look at the possibility of providing a semi-automated feedback mechanism for our heuristic by utilizing this method and the profiling of applications.

Our motivation, in this study, is to leverage only what is available at runtime, and avoid logging/utilizing high-cost meta-data (for instance task execution times, memory access patterns) which can be only obtained by extensive offline profiling. Not only this, but this meta-data then has to be maintained and identified with the corresponding tasks dynamically at runtime which can be clearly prohibitive due to additional performance, memory and power costs.

Lessons learned from Component Analysis:

We now summarize our findings from the component analysis (see also Table 7.1):

1.) The size of arguments and the number of arguments are nearly equally effective across all benchmarks. We suspect that this due to the fact that the program input is evenly distributed among the tasks.

2.) In Stream, other components lead to the protection of all tasks since they are the same for all tasks.

3.) In Sparse LU, the size of arguments (or equally number of arguments) are more effective than other components (by about 2×). In addition, the number of immediate successors is more effective than the number of immediate predecessors (by about 3×).

4.) In Cholesky, the number of immediate successors is the most effective component.

5.) For FFT and Perlin, certain tasks (early and late tasks respectively) are more effective in preventing the data corruption without regard to different components. In FFT and Perlin, there is only one unique static task definition. Thus, individual task components, such as the size of inputs or the number of immediate successors, become indistinguishable in terms of being more effective than the other components. Instead, individual task instances that are executed earlier or later during the benchmark execution become more distinctive and effective than the

Benchmark	Most Effective Components or Tasks
Sparse LU	Size or number of arguments
Cholesky	Number of immediate successors
FFT	Early tasks
Perlin Noise	Late tasks
Stream	No outstanding component or task

Table 7.1: Lessons learned from Component Analysis

complementary tasks.

Programmer-directed Selective Redundancy

Algorithm 3: Cholesky Factorization Algorithm

```
   Input: A: Input matrix.
   Input: NB: The number of blocks.
1  for (j = 0; j < NB; j++) do
2      #pragma omp task(A[j][j]) protect
3      spotrf(A[j][j]);
4      for (i = j + 1; i < NB; i++) do
5          #pragma omp task input(A[j][j]) inout(A[i][j]) protect
6          strsm(A[j][j], A[i][j]);
7      end
8      for (k = 0; k < j; k++) do
9          for (i = j + 1; i < NB; i++) do
10             #pragma omp task input(A[i][k], A[j][k]) inout(A[i][j])
11             sgemm(A[i][k], A[j][k], A[i][j]);
12         end
13     end
14     for (i = 0; i < j; i++) do
15         #pragma omp task input(A[j][i]) inout(A[j][j]) protect
16         ssyrk(A[j][i], A[j][j]);
17     end
18 end
```

In some applications, programmer insight can be helpful in order to provide a sufficient level of reliability. Thus, enabling the programmer to express its insight is crucial to get the necessary reliability while matching resource constraints. We provide an programmer annotation mechanism to mark certain tasks to be critical tasks by `protect` keyword and the runtime will replicate those tasks. This way, the runtime will partially protect the application as well as reduce resource consumption for fault-tolerance.

We now present the programmer insight for each benchmark. For Sparse LU, the programmer can specify the early tasks during the execution of the algorithm. We see that this selection is wise by also assessing the selection of the complementary tasks. In Algorithm 2, Lines 2,

6, 12 and 23 show how the programmer can specify these tasks for instance, by only adding annotations like if $(kk < NB/2)$ then protect. For FFT, similar to Sparse LU, the crucial tasks are the early tasks. As being a iterative refinement algorithm, the early stages of computation are more likely to affect the final output.

For Cholesky, we specify those tasks which process the diagonal entries. In every iteration of the algorithm, the first/diagonal phase is over the diagonal blocks. The second phase calculates the column blocks from the output columns of diagonal phase. The outputs of these phases are used in the third and fourth phases for calculating the rest of the blocks. The programmer intuition is that the diagonal phase outputs data that is used relatively more by the rest of the calculations. This means if errors occur in the output of the diagonal phase, it will be propagated relatively more. Algorithm 3 depicts the Cholesky Factorization algorithm which is implemented with `spotrf`, `strsm`, `sgemm` and `ssyrk` linear algebra subroutines. As we can see, the existing task directionality annotations in Lines 2, 5 and 15 already help to spot the tasks processing the diagonal entries. Therefore, the programmer just adds the `protect` keyword to these lines. For Stream, the programmer has the knowledge of the code such that one of the arrays in the application becomes destination 100 times more than the other two arrays. This array becomes the main source of error propagation to the final output since the error propagation is only possible through task arguments that are both inputs and outputs. Thus, the programmer specifies those tasks having that array as an output by only adding the `protect` keyword, similar to Cholesky.

Finally, in Perlin Noise, the later stages of the algorithm are proved to be decisive. The later iterations influence the final noise generation the most. Thus, the programmer chooses to protect the tasks constituting the later half part of the iterations, thereby detecting and correcting all errors that are visible in the program output. These annotations are similar to the Sparse LU or FFT annotations with straightforward `if` conditions.

7.1.3 Evaluation

We run our experiments in Marenostrum Supercomputer [90]. Tables 7.2 and Table 7.3 show our benchmarks.

We evaluate our selective redundancy mechanisms in terms of effectiveness. For these mechanisms, we take the averages of percentages of detected and corrected errors and of the final output corruptions as the final results. We measure the final output corruption by compar-

Sparse LU	LU decomposition Matrix size 6400x6400, block size 100x100
Cholesky	Cholesky factorization Matrix size 16384x16384 and block size 512x512
FFT	Fast Fourier Transform Array size 16384x16384, 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 32678

Table 7.2: Details of task-parallel shared-memory benchmarks

Nbody	Interaction between N bodies Array size 65536 bodies, block size depends on #nodes
Matmul	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 ported to OmpSs Matrix size 131072 doubles, block size 256, 8x8 grid

Table 7.3: Details of distributed benchmarks

ing the corresponding elements in the correct outputs of the fault-free executions to those in the outputs of the fault-injected executions. The injection overheads are included in the reported overheads. We now present the complete and selective task redundancy results.

Complete Task Redundancy Results

In this section, we evaluate the scalability and overheads of complete task redundancy. Our motivation is that if experiments show that complete task redundancy scales and has low overhead, then it trivially follows that partial redundancy also scales and has low overheads. In addition, the scalability experiments will indicate whether task redundancy have any negative effect on the scalability of the applications.

Figure 7.3 illustrates the scalability of the complete task redundancy for shared memory benchmarks. In the figure, the x-axis shows the number of cores and y-axis shows the speedups. They show the baseline/original speedups without any fault injection and any redundancy, speedups without any fault injection but with complete redundancy (0% fault rate), and

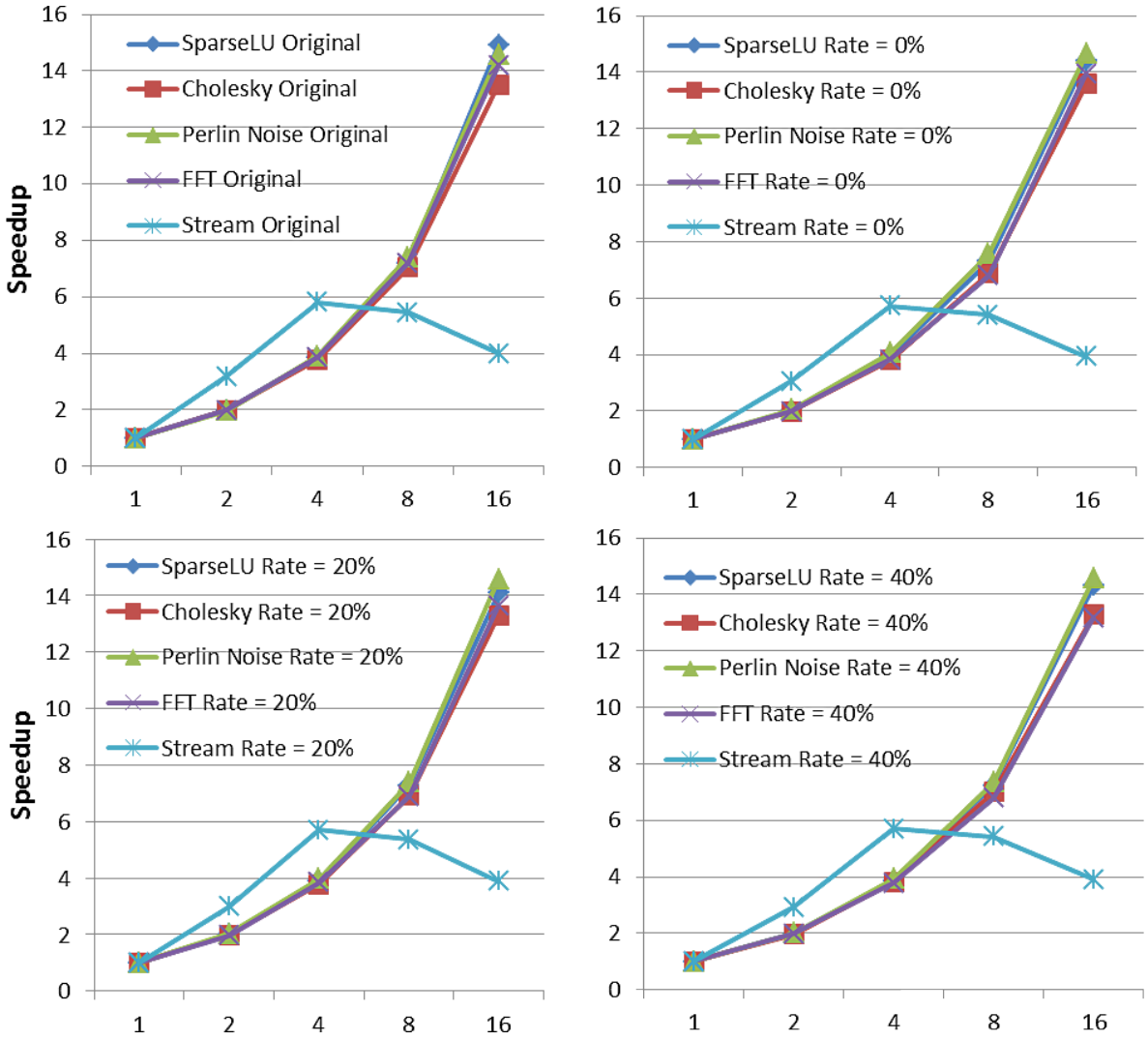


Figure 7.3: Scalability for shared-memory benchmarks

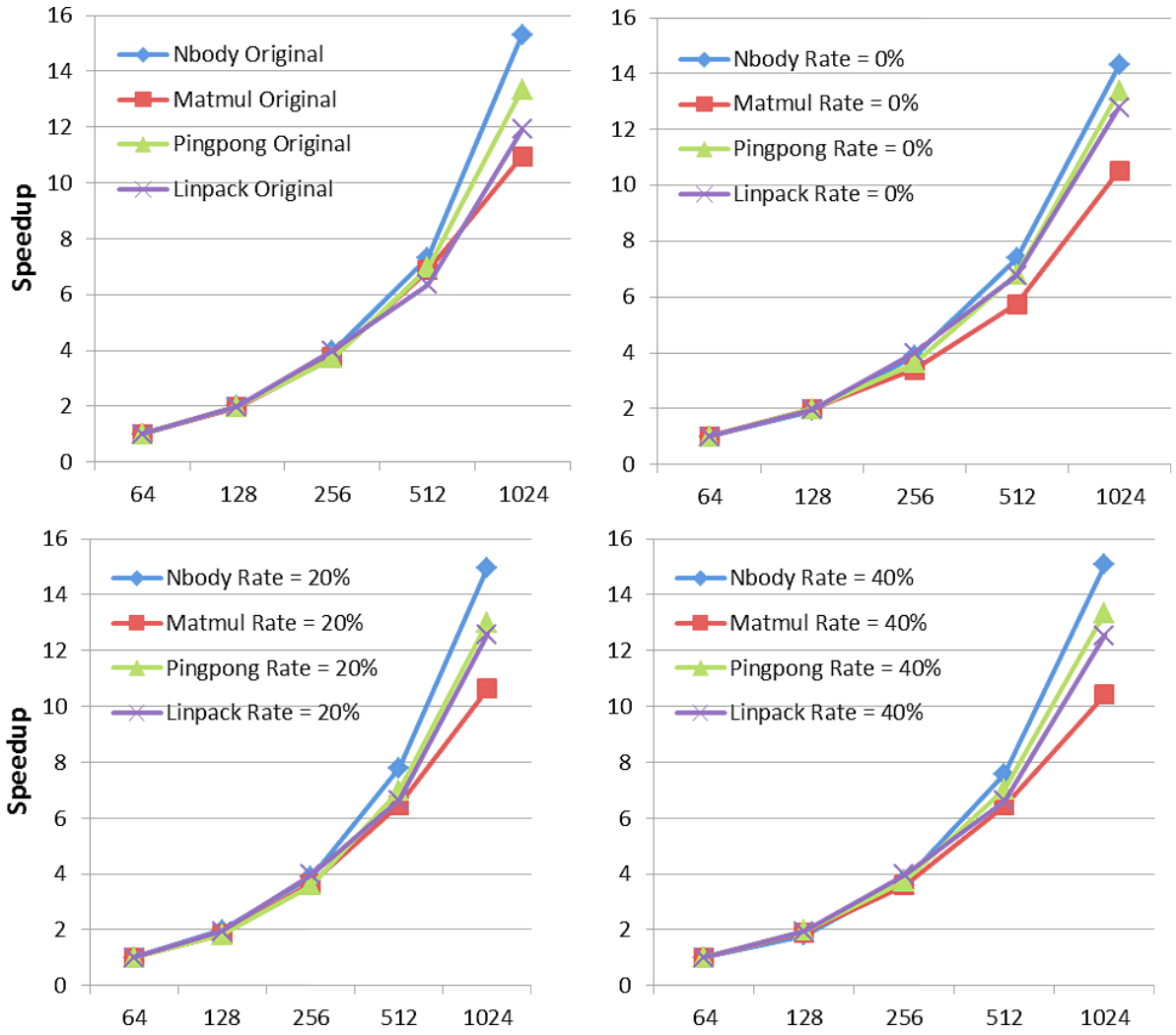


Figure 7.4: Scalability for distributed benchmarks

with 20% and 40% fault rate per task fault injections with complete redundancy. As seen, complete task redundancy scales very well, which means it does not affect scalability negatively. In Stream, we see that it scales well up to 4 cores (the super-linear speedup with 4 cores is due to the fact that task arguments fit in 20 MB caches of each core). It does not scale with 8 and 16 cores even when task redundancy is disabled. Stream mostly consists of memory operations and does not have much parallelism. We purposely chose it to analyze the behavior of an artificial memory-intensive benchmark in terms of performance and scalability since it is a good candidate to stress task redundancy.

We note that the fault rates used for injections are high considering the fault frequencies in HPC systems since the rates that we experiment with are for a *single* task. This in effect simulates higher fault rates for the overall system. Our motivation is that if we achieve good performance for the upper bound case of high error rates, it would confirm our expectation that our implementation is what we call “failure scalable".

Figure 7.5(a) shows the complete task redundancy performance overheads with respect to the fault-free execution times of the benchmarks. These overheads include task input checkpointing and output comparison overheads when benchmarks are executed with 16 cores. However for other core counts they are similar. As seen, they are very low for Sparse LU, Cholesky, Perlin and modest for FFT and Stream. In FFT with the given benchmark block size, the task granularity is very coarse; there are only 256 application tasks. This prevents effective overlapping of checkpointing with task computations. In Stream, memory system is further stressed with the additional I/O due to checkpointing. Note that duplicate tasks are executed on spare cores.

Distributed Applications Results: We include and evaluate these hybrid MPI and task parallel applications in order to experiment and show i) complete task redundancy is well-suited for such hybrid MPI programs and ii) it incurs low overhead and is highly scalable for high core counts at large scale. Figure 7.4 shows the scalability of the complete task redundancy for distributed benchmarks i.e. speedups over 64 cores. We see that complete task redundancy is also scaling for these benchmarks. Moreover, Figure 7.5(b) shows the performance overheads with respect to the fault-free execution times when 1024 cores are used. As we can see, the overheads are low.

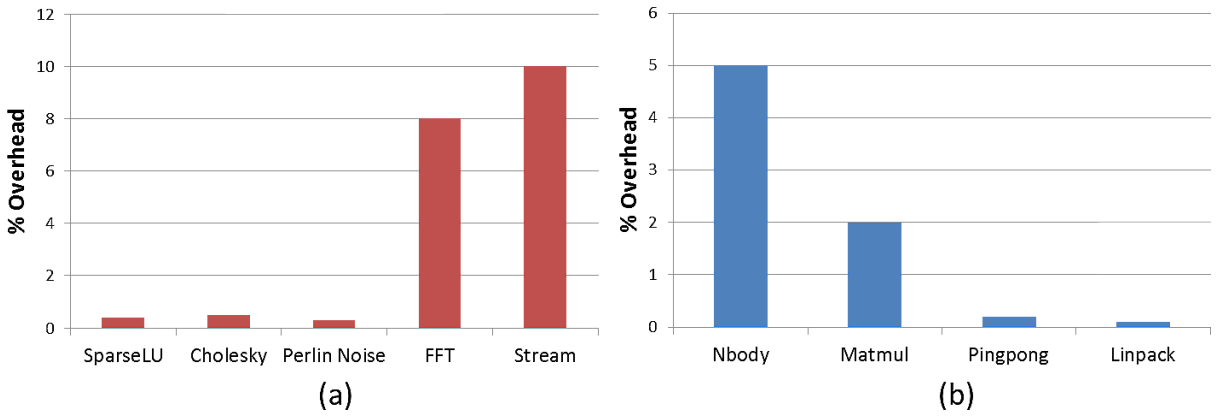


Figure 7.5: Task redundancy overheads

Selective Task Redundancy Results

To assess the selective redundancy framework thoroughly, we use two fault injection settings. First with the fixed fault-rate per task injection setting, we aim to analyze the performance of the framework in a task-centric manner. Second with a fault injection setting based on the application execution time and input size, we aim to assess it in an application-centric manner, that is to consider the memory space and the time in the processor pipeline when the data is alive. Including the task execution time in the injection model enables us to mimic the impact of combinational logic errors in the processor pipeline. We have two different scenarios for the second injection setting: Time 1 is the one where we inject 1 fault per 1 second of the fault-free execution time per 32 MBs of the benchmark input. For the case of Perlin Noise, we inject 1 fault per 1 second per 2048 pixels. In Time 2, we inject at a rate of around 1/5 of Time 1. We use these two time-based settings to analyze our framework in different fault occurrence frequencies.

Tables 7.4 A) and B) show the automated task protection results of our heuristic by showing the percentage of the performance overhead including re-execution of tasks to the fault-free execution time and the percentage of detected and corrected errors. The results for FFT and Stream show that our heuristic is very effective. For Cholesky and Sparse LU, it is moderately effective. We note that in the time-based scenario we see that FFT has relatively more overhead (more than 20%) than other benchmarks. The reason is that around half of the tasks are selected to be protected and this overhead comes from the re-execution overhead of these selected long-

running tasks. For Perlin Noise, the results for fixed-rate injections are similar to the results of Cholesky and Sparse LU. In case of time-based injections, since the number of injections is significantly lower than those of fixed-rate injections, we almost always detect and correct errors.

Tables 7.4 C) and D) show the results for programmer-directed selective redundancy. Consistently, our heuristic is very effective for FFT and Stream, and it is moderately effective for Cholesky and Sparse LU. The point we made above for FFT in time-based injection setting is also valid for the programmer-directed mechanism. For Perlin Noise the results show that the programmer hint is very effective, as we explained before, the tasks in the later phases influence the final output the most. Overall our automated and transparent heuristic is as good as the programmer-directed mechanism.

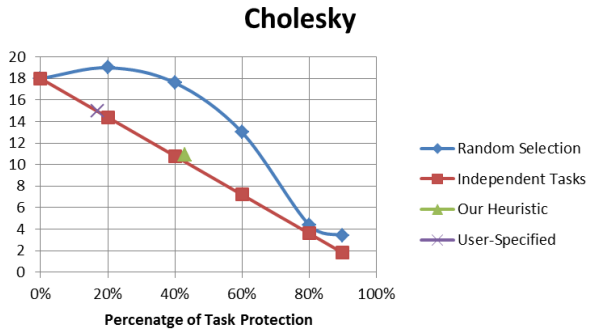


Figure 7.6: Comparison to random task selection: Cholesky

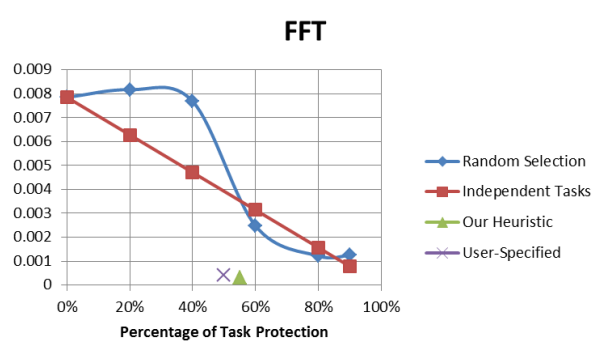


Figure 7.7: Comparison to random task selection: FFT

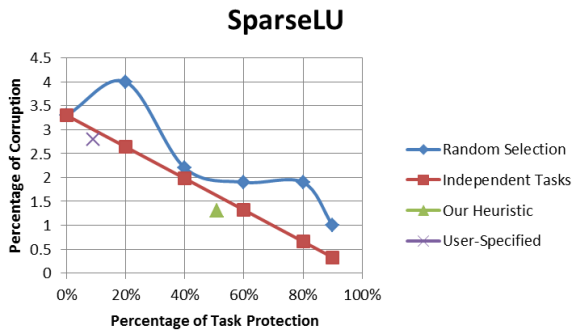


Figure 7.8: Comparison to random task selection: Sparse LU

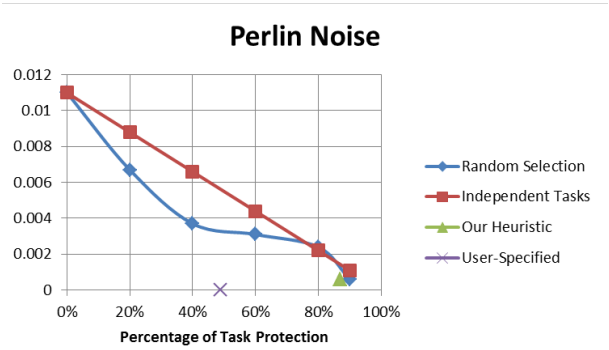


Figure 7.9: Comparison to random task selection: Perlin

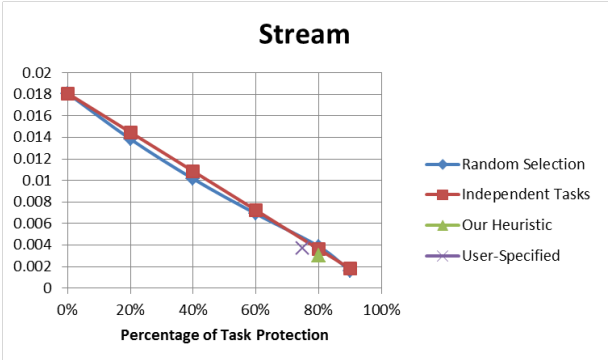


Figure 7.10: Comparison to random task selection: Stream

A) Automated Task Selection: Fixed Rate Injection				
	Fault Rate 3%		Fault Rate 20%	
Benchmarks	**Overhead**	**% DetectCorrect.**	**Overhead**	**% DetectCorrect.**
SparseLU	1.8%	38%	3.5%	18%
Cholesky	0.6%	57%	4.7%	23%
FFT	2.9%	99%	8%	99%
Perlin	1.4%	75%	10%	38%
Stream	3.1%	77%	3.6%	80%
B) Automated Task Selection: Time-based Injection				
	Time 1		Time 2	
Benchmarks	**Overhead**	**% DetectCorrect.**	**Overhead**	**% DetectCorrect.**
SparseLU	1.8%	39%	0.9%	89%
Cholesky	1.7%	49%	1.2%	46%
FFT	27.7%	99%	21.9%	99%
Perlin	0.07%	100%	0.02%	100%
Stream	3.6%	84%	2.4%	99%
C) Programmer-directed Task Selection: Fixed Rate Injection				
	Fault Rate 3%		Fault Rate 20%	
Benchmarks	**Overhead**	**% DetectCorrect.**	**Overhead**	**% DetectCorrect.**
SparseLU	0.8%	27%	1.9%	24%
Cholesky	0.7%	19%	1.4%	16%
FFT	2.2%	99%	8.1%	99%
Perlin	1.4%	100%	7.4%	100%
Stream	1.2%	76%	2.4%	76%
D) Programmer-directed Task Selection: Time-based Injection				
	Time 1		Time 2	
Benchmarks	**Overhead**	**% DetectCorrect.**	**Overhead**	**% DetectCorrect.**
SparseLU	0.8%	68%	0.8%	25%
Cholesky	0.9%	15%	0.6%	19%
FFT	28.7%	99%	22.1%	99%
Perlin	0.48%	100%	0.2%	100%
Stream	0.5%	46%	0.2%	99%

Table 7.4: Partial task selection results

We additionally compare our heuristic and programmer-directed task selection to random task selection to discuss their effectiveness. Figures 7.6, 7.7, 7.8 7.9 and 7.10 show how effective our heuristic and programmer hints are. In all benchmarks, both our heuristic and programmer hints are more effective than the random selection. We experimentally obtain ran-

dom task selection curve by running experiments with different random task selection rate and 3% per task fault injection rate. We then construct the linear curve (by extrapolation), which represents the independent tasks, by calculating successive points assuming that the percentage of corruption is linearly correlated with the percentage of task protection. We start from the first point where fault injection is performed with no task protection. This conceptual curve is drawn to see the case as if the tasks were completely independent.

We now analyze the effect of the task dependencies in our applications on the error propagation to the final output. Obviously, in an application when tasks are independent of each other, then the final output corruption is a linear function of the percentage of task being protected (represented by the linear curve). However, because of the dataflow execution model, tasks are dependent on each other. Thus, the final output corruption is a nontrivial function of the percentage of task protection and this function differs for each particular application. For Sparse LU and Cholesky, we see that this function is similar and different than the linear case. For Stream, even though there are task dependencies, the application behavior is almost same as if the tasks were independent. For FFT and Perlin Noise, the application behavior shows that there are some tasks that are clearly crucial than others in terms of final output corruption.

7.1.4 Conclusions

In this section, we present our two lightweight mechanisms - an automatic risk-based heuristic and a programmer-directed partial task redundancy - to mitigate silent and fail-stop errors for task-parallel HPC applications. We show that our designs are highly efficient and scalable. Our schemes are shown to be effective by our results for the cases where partial fault-tolerance is considered adequate. In the next section, we will provide a more systematic way to select tasks for replication.

7.2 Modelling Task-Parallel Dataflow Applications

In this section, we provide a systematic and formal way to select tasks for replication. It is known that the optimal selective replication is NP-hard which can be formalized as a bounded knapsack problem [103]. Therefore, practical selective replication solutions must employ heuristics. In our main contribution in this section, we propose two runtime-based, fully automatic and completely transparent heuristics, called App_FIT and Target_Rep, 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) 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 [66]. With Target_Rep, users can set the maximum resource utilization for replication and our heuristic maximizes the application reliability by transparently and automatically replicating tasks but without exceeding the resource utilization. Target_Rep is useful in cases when the system is not fully utilized and the idle/spare resources can be used for replication. For example, 10% of the nodes might be idle, and we could utilize this spare capacity by replicating 10% of the tasks. This is especially attractive since our replication has low performance overhead and the Target_Rep heuristic chooses the tasks which would improve the program reliability the most.

In order to understand how much we would improve the overall system reliability by replicating a task, we develop a theoretical model based on Markov chains. The reliability model provides a quantitative way to estimate the reliability of task-parallel dataflow HPC applications. It is basically a mathematical framework to characterize the reliability of task-parallel dataflow programs with or without task replication. *To the best of our knowledge, this is the first mathematical model which describes the reliability of task-based parallel HPC programs. In particular, our model is the first that distinguishes the failures/crashes and SDCs. The proposed reliability model is a powerful tool which enables us to develop efficient selective replication techniques (heuristics). Moreover these techniques are the first to solve the problems regarding the reliability and resource budgets of task-parallel HPC applications.*

The results suggest that our task replication design and heuristics have very low overheads. The fault-free performance overhead of complete task replication is 2.5% on average. For App_FIT it is negligible while for Target_Rep it is 1.2% on average. Our model predicts the application FITs very accurately with only 0.26% deviation from the actual FITs obtained by fault-injection experiments.

Utilizing our model, we find that complete task replication over-allocates resources. In fact, we show that by using App_FIT heuristic, we can tolerate pessimistic exascale error rates with only 53% of task replication. Moreover, our Target_Rep heuristic stays within 5% of the optimum solutions with 50% target task replication percentage.

We highlight three 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, some application tasks will be relatively more reliability-critical in the exascale time-frame. Third, our evaluation confirms our intuition that fault-tolerance based on task-parallel dataflow programming is efficient, scalable and low-overhead. Briefly, our contributions in this section are:

- Development, validation and implementation of a reliability model for task-parallel dataflow programs based on Markov Chains.
- Two automatic and efficient heuristics to selectively replicate tasks while reducing costs significantly.
- Design, implementation and evaluation of low overhead and highly scalable selective task replication.

The rest of this section is organized as follows: Subsection 7.2.1 provides the design of selective task replication. Subsection 7.2.2 presents our reliability model. Subsection 7.2.3 discusses our heuristics. Subsection 7.2.4 presents the experimental evaluation. Finally, subsection 7.2.7 summarizes this section.

7.2.1 Selective Task Replication Design:

We have discussed the complete task replication in Section 7.1.2. In this section we discuss our formal Markov model based selective replication design.

In this design, selection for task replication is based on a criteria from the Markov model. This criteria for task selection is the rate that the replicated task will improve the overall application reliability. To estimate the rate of the reliability improvement we develop a rigorous model that we describe in Section 7.2.2. At runtime the reliability model is invoked before the execution of a task and the decision for the selection of the task for replication is taken by our heuristics.

The basic design/behaviour of the selective task replication is as follows: If a task is not replicated and experiences some errors that lead to a crash or SDCs, it crashes - in which the runtime aborts - or it produces a wrong output. If it is replicated and one of replicas crashes, the runtime ignores the crashed one and continues with the computation of other replica. If

replicas experience some SDCs, SDCs are detected by the comparison of the outputs and the task is recovered by the checkpoint. However they will not detect those SDCs and will produce wrong result if the SDCs occur in exactly the same bit positions across the replica outputs. If both replicas crash, the runtime aborts. Note that there are options that can be easily included in our design such as trying to recover a crashed replica from the checkpoint.

7.2.2 Reliability Model

We now introduce and elaborate on our theoretical model for estimating the reliability of task parallel programs. We start by introducing the concepts which take part in our model. Then we present the reliability model when tasks are not replicated. After that, we extend the model using Markov Chains to account for the case when tasks can be replicated. Finally we discuss the calculation of the failure rates of tasks and benchmarks, and the orthogonality of our framework to the failure rate estimation studies.

Concepts of the Reliability Model

In our model we first define what we call *the intrinsic reliability* of a task to take into account the effects of the failures and silent data corruptions (SDCs) happening solely due to task-local causes. Examples of these are event upsets or bit flips occurring during the task computation in the resources (memory, processor structures) on which the task computation is performed. The intrinsic reliability is the probability that the task will not crash and not experience SDCs during a period of time from the point that the task started its computation. In essence, the intrinsic reliability excludes the errors stemming from other tasks such as predecessor tasks from the task dependency graph and child tasks in case of task nesting. As a consequence, *the intrinsic reliabilities* of the tasks are independent and hence multiplicative.

After defining individual task intrinsic reliabilities, we proceed with defining the reliability of a task-parallel program. For a task-parallel program to successfully finish its computation without experiencing any SDCs, all the program tasks should finish successfully their computations without experiencing any SDCs. Hence, we define the reliability of a task-parallel program to be the product of all task intrinsic reliabilities. We will define both *the overall* and *instantaneous* reliability of a program where the former captures the reliability from the beginning of the program computation till the very end of it and the latter captures the reliability up to some particular time during the computation of the program.

Our framework does not model the message passing library or messages over the network. For the protection of the library and messages, orthogonal SDC mitigation mechanisms (which can be seamlessly integrated with our framework), e.g., Fiala et al. [2], should be deployed.

Reliability Model for Task-parallel Programs without Replication

In this section we present our formal reliability model in which we define the intrinsic and complete reliabilities of tasks, and the overall and instantaneous reliability of task-parallel programs. This model sets the base for our model that incorporates task replication in the next section.

Let $ExecTime(T)$ be the amount of time that the task T takes to finish its computation. We define *the intrinsic reliability* of the task T during $0-t$ period, denoted by $R_{intr}(T,t)$, as the probability it will *not crash* and *not experience SDCs* from the beginning of its computation ($t=0$) until time t where $0 <= t <= ExecTime(T)$ due to task-local errors. We assume that the system on which the task-parallel programs are running is in its useful lifetime which means that the failure rate is constant and thus the intrinsic reliability has an exponential distribution [104]. Mathematically, $R_{intr}(T,t) = e^{-\lambda(T)t}$ where $\lambda(T)$ is the failure rate of task T. Note that the unit of $\lambda(T)$ is Failures in Time (FIT) showing the number of failures in one billion hours. The intrinsic reliability of the task when it finishes, i.e., $t = ExecTime(T)$, is denoted by $R_{intr}(T,\perp)$, and $R_{intr}(T,\perp) = e^{-\lambda(T)\times ExecTime(T)}$.

Note that it is straightforward to modify the reliability definition to account for different phases in the lifetime of a computing system such as for the aging phase of a system by Weibull distribution where $R_{intr}(T,t) = e^{-\lambda(T)(t/\alpha)^{\beta}}$. In Weibull distribution α is known as the scale parameter and β is the shape parameter. Briefly, $\beta < 1$ models the burn-in phase, $\beta = 1$ models the useful lifetime phase (equivalent to the exponential distribution we use) and $\beta > 1$ models the aging (wear-out) phase in Weibull distribution [104].

We now establish the (overall) complete reliability definition for a task step by step. Our aim is to account for *the extrinsic effects* of the predecessor tasks due to the dependencies and of the descendant (child) tasks due to nesting on the reliability of a task. Hence the complete reliability of a task T during $0-t$ period is defined as:

$$R(T,t) = R_{intr}(T,t) \times R_{pred}(T) \times R_{chd}(T,t). \tag{7.3}$$

where $R_{intr}(T,t), R_{pred}(T)$, and $R_{chd}(T,t)$ is its intrinsic, its predecessors' and its nested chil-

dren's extrinsic reliabilities respectively. The extrinsic reliabilities are defined below.

Effect of Predecessors Tasks: In task-parallel dataflow programs tasks constitute the task dependency graph under dataflow semantics. As a result of this, every task has a set of *predecessor tasks* due to the dependencies among them. Let $PR(T)$ be the set of predecessor tasks of task T. When T is about to be executed, all of its predecessors had finished their computations which is guaranteed by the data dependencies. Thus, *the extrinsic reliability of the predecessors* of task T during $0-t$ period is:

$$R_{pred}(T) = \prod_{i}^{|PR(T)|} R_{intr}(T_i, \perp) \times R_{pred}(T_i, \perp). \tag{7.4}$$

Effect of Nested Tasks: In task-parallel programs nested parallelism is commonly used and as a result some tasks are *nested*: They create new tasks during their computations. Let $CH(T,t)$ be the set of children of task T which are either finished or executing at time t. *The extrinsic reliability of the descendants* during $0-t$ period, denoted by $R_{chd}(T,t)$:

$$R_{chd}(T,t) = \prod_{i}^{|CH(T,t)|} R(T_i, t). \tag{7.5}$$

Note that $R(T_i,t)$ is the complete formula for the i^{th} child defined in Equation 7.3. If a child T_i is finished by the time t, $R_{chd}(T_i,\perp)$ is used to refer to its reliability when it finished instead of $R_{chd}(T_i,t)$.

The reliability definitions for task graphs: We now define the reliability for a task dependency graph. Let $G = (V,E)$ be the task dependency graph of the entire computation of a task-parallel program where V is the set of non-child tasks (including predecessors) in the graph and E is the set of dependencies. We define the *the overall reliability of* G, denoted by $R(G)$, and *the instantaneous reliability of* G, denoted by $R(G,t)$, as follows:

$$R(G) = \prod_{i}^{|V|} R_{intr}(T_i, \perp) \times R_{chd}(T_i, \perp) \tag{7.6}$$

$$R(G,t) = \prod_{i}^{|L(t)|} R_{intr}(T_i, t) \times R_{chd}(T_i, t) \tag{7.7}$$

where $L(t)$ is the set of leaf tasks (equivalently the set of executing) at time t.

Finally we define *the overall* and *instantaneous* reliability of a task-parallel program respectively as follows:

$$R(App) = \prod_{i}^{|\aleph|} R_{intr}(T_i, \perp) \tag{7.8}$$

$$R(App, t) = \prod_{i}^{|\varepsilon(t)|} R_{intr}(T_i, t) \tag{7.9}$$

where $\aleph$ is the set of all application tasks and $\varepsilon(t)$ is the set of executing tasks at time t. Basically the reliability of a task-parallel program is the product of intrinsic reliabilities of all tasks in the program.

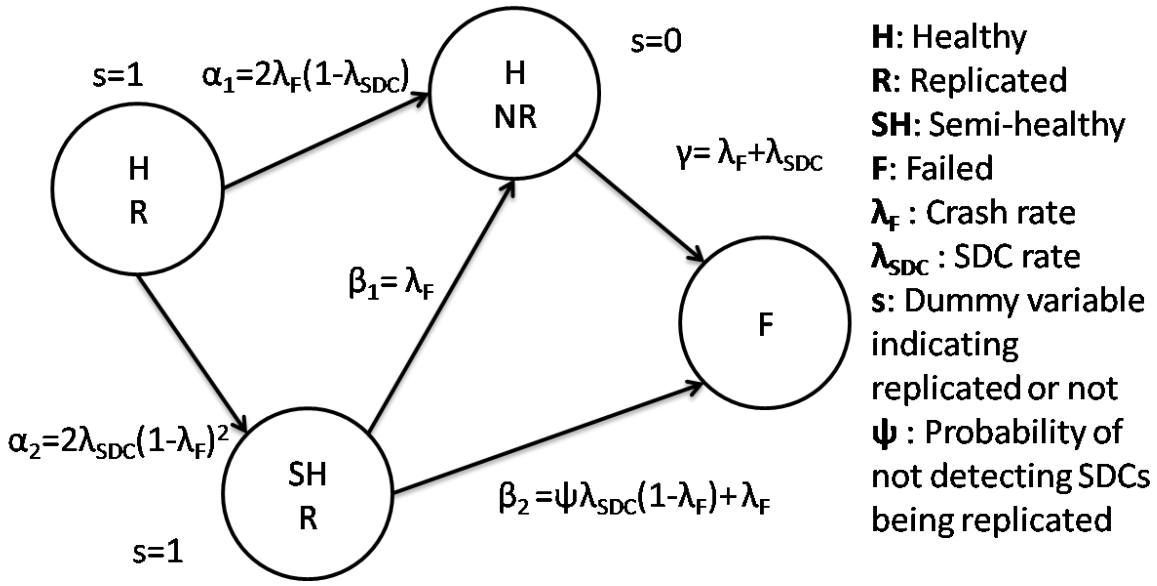


Figure 7.11: The Markov model for a single task

In this section we described the reliability model for programs where tasks are not replicated. Now in the next section we continue by extending this model for programs in which tasks are replicated.

Reliability Model for Task-parallel Programs with Replication

When task replication is available, it is not possible (or may not be the best fit) to use only combinatorial techniques to model the intrinsic reliability for the tasks. This is because in case of being replicated, a task has different states during its computation and moves among them with some error rates. Thus we establish a Markov model based on Markov chains. The model is constructed for *a single task* and we use it to remodel the *intrinsic reliabilities* of the tasks. The reliability definitions (Equations 7.8 and 7.9) for a task-parallel program remain the same.

A Markov model is a stochastic model assuming the *memoryless* property in which the future states are determined solely by the present state and are independent of past states. A Markov chain is a mathematical system where the system (in our case an individual task) goes through transitions from one state to another in the state diagram. We establish the state diagram where a task goes through the possible set of states with certain error rates. Figure 7.11 shows our model for a single task. State HR represents the case where the task is *healthy* and replicated. A task is *healthy* if it has not experienced any SDCs and has not failed (crashed). State SHR represents the case where the task is replicated and is *semi-healthy*. A task is *semi-healthy* if none of the replicas has crashed but one of them has experienced some SDCs. State HNR refers to the case where exactly one of the replicas has crashed but the remaining one has no SDCs, i.e., it is *healthy*. Finally, state F indicates that the task is *in failure*. That is, both replicas have crashed or they have experienced SDCs that have not been detected.

We now elaborate on the transitions between the states. Transitions encapsulate the rate and conditions for them to occur between states. $\lambda_{SDC}(T)$ refers to the rate of experiencing a SDC for task T. Similarly $\lambda_F(T)$ is the rate for failures or crashes due to DUEs for task T. The transition from state HR to state HNR shows that only one of the replica crashes and the other does not experience any SDCs for which there are two possible combinations. We denote it by $\alpha_1 = 2\lambda_F(T)(1-\lambda_{SDC}(T))$. The transition from state HR to state SHR refers the case where none of the replicas crashes and exactly one of them experiences some SDCs for which there are two possible combinations. We denote it by $\alpha_2 = 2\lambda_{SDC}(T)(1-\lambda_F(T))^2$. The transition from state SHR to state HNR indicates that the replica which experienced some SDCs actually crashes and the remaining replica has no SDCs i.e., healthy. We denote it by $\beta_1 = \lambda_F(T)$. The transition from state SHR to state F represents the case where either the healthy replica crashes or the replicas experience the same number of SDCs in the same memory locations whose probability is represented by ψ. The latter causes SDCs to go undetected. If the size of

the memory usage of a task is N bits in total and k errors occur in N bits, then $\psi = 1/C(N,k)$ ($C(N,k)$ denotes N *choose* k). We denote this transition by $\beta_2 = \psi\lambda_{SDC}(T)(1-\lambda_F(T)) + \lambda_F(T)$. Finally, the transition from state HNR to state F shows that the remaining replica crashes or experiences some SDCs. We denote it by $\gamma = \lambda_F(T) + \lambda_{SDC}(T)$.

We now establish the Markov equations from the state diagram from which we will derive *the intrinsic reliability* of a single task. Let $P_{HR}(t)$, $P_{SHR}(t)$, $P_{HNR}(t)$ and $P_F(t)$ be the probability of a task to be in the state HR, SHR, HNR and F respectively. We use the dummy s variable to indicate whether a task is replicated before its computation ($s = 1$) or not ($s = 0$). The s variable is set when a task is chosen to be selected by the runtime heuristic. Then the series of differential equations describing the state diagram for a single task is

$$\frac{dP_{HR}(t)}{dt} = -(\alpha_1 + \alpha_2)P_{HR}(t) \tag{7.10}$$

$$\frac{dP_{SHR}(t)}{dt} = \alpha_2 P_{HR}(t) - (\beta_1 + \beta_2)P_{SHR}(t) \tag{7.11}$$

$$\frac{dP_{HNR}(t)}{dt} = \alpha_1 P_{HR}(t) + \beta_1 P_{SHR}(t) - \gamma P_{HNR}(t) \tag{7.12}$$

$$\frac{dP_F(t)}{dt} = \beta_2 P_{SHR}(t) + \gamma P_{HNR}(t) \tag{7.13}$$

with the initial conditions:
$P_{HR}(0) = s, P_{SHR}(0) = 0, P_{HNR}(0) = 1 - s$ and $P_F(0) = 0$.

The solution to this series of differential equations of Markov model:

$$P_{HR}(t) = se^{-(\alpha_1+\alpha_2)t} \tag{7.14}$$

$$P_{SHR}(t) = C_1 \times (e^{-(\alpha_1+\alpha_2)t} - e^{-(\beta_1+\beta_2)t}) \tag{7.15}$$

$$\begin{aligned} P_{HNR}(t) = {} & \frac{s\alpha_2}{\gamma - (\alpha_1+\alpha_2)} \times e^{-(\alpha_1+\alpha_2)t} \\ & + C_2 \times e^{-(\alpha_1+\alpha_2)t} - C_3 \times e^{-(\beta_1+\beta_2)t} \\ & + (1 + s\frac{\gamma(\alpha_2+\beta_1+\beta_2) - \gamma^2 - \alpha_1\beta_2}{(\gamma-(\alpha_1+\alpha_2))(\gamma-(\beta_1+\beta_2))}) \times e^{-\gamma t} \end{aligned} \tag{7.16}$$

$$P_F(t) = 1 - P_{HR}(t) - P_{SHR}(t) - P_{HNR}(t) \tag{7.17}$$

where

$$C_1 = \frac{s\alpha_2}{\beta_1 + \beta_2 - (\alpha_1 + \alpha_2)} \tag{7.18}$$

$$C_2 = \frac{s\beta_1\alpha_2}{(\gamma - (\alpha_1 + \alpha_2))(\beta_1 + \beta_2 - (\alpha_1 + \alpha_2))} \tag{7.19}$$

$$C_3 = \frac{s\beta_1\alpha_2}{(\gamma - (\beta_1 + \beta_2))(\beta_1 + \beta_2 - (\alpha_1 + \alpha_2))} \tag{7.20}$$

Hence the intrinsic reliability of a task T when replication is available is defined as

$$R_{intr}^{Rep}(T,t) = P_{HR}(t) + P_{SHR}(t) + P_{HNR}(t). \tag{7.21}$$

From the formula above MTTF (Mean Time To Failure) [104] of a task T is calculated as:

$$MTTF(T) = \int_0^\infty R_{intr}^{Rep}(T,t)\,dt. \tag{7.22}$$

After calculating MTTF, we calculate FIT in billion hours of task T as follows:

$$FIT(T) = \frac{1}{MTTF(T)} \times 10^9. \tag{7.23}$$

For brevity we omit the FIT formula, but it directly follows from applying Equation 7.22 to the solutions.

An interesting ramification of our Markov model is that the selective replication will be more significant and the difference among the impacts of the tasks on the overall application reliability will be even more visible: The intrinsic reliability of T can be written if it is replicated as follows:

$$R_{intr}^{Rep}(T,t) = c_1 \times e^{-(\alpha_1+\alpha_2)t} + c_2 \times e^{-(\beta_1+\beta_2)t} + c_3 \times e^{-\gamma t} \tag{7.24}$$

for some constants c_1, c_2 and c_3. In addition, if T is not replicated, its intrinsic reliability (it is

in HNR state at the beginning of its computation) is

$$R_{intr}(T,t) = c_3' \times e^{-\gamma t} \tag{7.25}$$

for some constant c_3'. Thus *the reliability impact (improvement) factor* of T is

$$\frac{R_{intr}^{Rep}(T,t)}{R_{intr}(T,t)} = \frac{c_1}{c_3'} \times e^{(\gamma-(\alpha_1+\alpha_2))t} + \frac{c_2}{c_3'} \times e^{(\gamma-(\beta_1+\beta_2))t} + \frac{c_3}{c_3'}, \tag{7.26}$$

which is an exponential function.

Takeaway-1: *Hence, given exascale task execution times and failure rates, some tasks will be exponentially more significant than others in terms of their impact on the overall reliability of the application.*

The Failure Rates of Tasks and Benchmarks

In this section we first present how we calculate failure rates in FITs for each application task and for the whole application/benchmark. We take the FIT rates for crashes (DUEs) and SDCs from Michalak et al. [105] for the Roadrunner supercomputer. Michalak et al. obtained these rates via accelerated neutron-beam test. We take the FITs of a Roadrunner TriBlade and adjust them for each individual task and each benchmark proportional to their memory footprint and execution times. We use the task failure rates i.e., $\lambda_F(T)$ and $\lambda_{SDC}(T)$, to feed our mathematical model. They are also used for the validation of the model. We use the benchmark FIT rates to calculate and specify the target reliability thresholds which are to be achieved by the App_FIT heuristic.

We now elaborate on how our framework, i.e., the reliability model and heuristics, 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 [105]. 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 reliability model and heuristics. 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 [106] 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 model and heuristics, without any further modification, will simply make use of this refined task failure rate instead of the previous rate.

We estimate the reliability of the overall application by using the described model in this section, and by considering whether the individual tasks are replicated or not. The two task selection heuristics described in the next section use this model to make a decision to replicate a task.

7.2.3 Task Selection Heuristics

When selecting tasks dynamically at runtime, our goal is to avoid requiring the knowledge of the entire execution (can be obtained by offline profiling) and to avoid keeping extensive information as the execution continues since both are expensive. Therefore we propose heuristics that make use of only already existing information at runtime to achieve efficient, lightweight and near-optimal selective task replication. By using the free information from dataflow, we are able to design and implement two automatic heuristics which do not require any profiling. We detail them in the next two subsections.

App_FIT Heuristic

This heuristic is for the case 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 to decide whether to replicate the task T:

$$current\_fit + (\lambda_F(T) + \lambda_{SDC}(T)) > (threshold/N) \times (i+1) \tag{7.27}$$

where $current\_fit$ is the current FIT of the computation at the time which is maintained by App_FIT, $\lambda_F(T)+\lambda_{SDC}(T)$ is the FIT that the task will incur if it is not replicated, $threshold$ 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 that the Markov model dictates. If the condition does not hold, App_FIT does not replicate the task.

Note that failure rates $\lambda_F(T)$ and $\lambda_{SDC}(T)$ depend on the memory usage and execution time of a task T. They are proportional to each task instance's computation time (duration) and memory usage. This means the failure rates are different across the tasks. However since our heuristics will not know the memory usage before executing the task, instead they use the size of all arguments (inputs + outputs) to estimate them which is available at runtime before the execution of the task thanks to the dataflow annotations. We will show that the size of all arguments is indeed close to the task memory usages in Section 5.5. The proportionality to the computation time is already captured by the fact that the unit of failure rates is in FITs in one billion hours. At the end of the execution of a task, the failure rates are adjusted with respect to the length of the execution.

Target_Rep Heuristic

Target_Rep is to replicate some x% of the application tasks according to the spare resources in the system on which the application runs. In essence, it is a greedy approach that tries to get the global optimal solution by approximating and aggregating the local optimal solutions using the set of ready tasks as the computation progresses. Algorithm 4 is the pseudo-code for Target_Rep. We omit details such as the synchronization among the threads for brevity. Since Target_Rep will not have the entire global information, we design it as follows to get a near-optimal solution: Target_Rep inspects the ready task queue as needed, marks and sorts the tasks in the queue according to their FITs (Lines 8-11). It selects the first x% of tasks from the set of marked tasks (Line 12). It accumulates the application FIT as the execution continues according to whether a task has been selected for replication (Lines 19-25).

7.2.4 Evaluation

In this section we provide the evaluation and analysis of our techniques. We implement our ideas in OmpSs [18] and Nanos [21]. We perform our experiments on Marenostrum supercomputer [90]. Up to 64 nodes and 16 cores per node are used in the experiments. Table 7.5 summarizes our benchmarks [98]. In addition to shared-memory benchmarks, we have distributed

Algorithm 4: Target_Rep Heuristic

```
   Input: x: Target % of replication to be performed.
   Output: appFIT is the final FIT of the application
   /* RdQ is the ready queue.  N is the total of the tasks in the application.     */
1  Algorithm begin
2      appFIT = 0; /* Global atomic variable                                      */
3      numSelected = 0; /* Global atomic variable                                 */
4      target = x × N; locCounter = 0;
5      For each thread in Threadpool:
6      repeat
7          createTasksAndAddtoQueue(RdQ);
8          if (locCounter == 0) then
9              if (numSelected < target) then
10                 SortedQueue queue;
11                 queue = sortandMarkOriginals(RdQ);
12                 selectTasks(queue);
13             end
14         end
15         Task t = RdQ.pollMarkedTasks();
16         if t == NULL then continue;
17         executeTask(t);
18         if t.isTwin() then continue;
19         if t.isSelected() then
20             t.waitTwin(); locCounter − −;
21             updateAppFITwithReplica(t, appFIT);
22         end
23         else
24             updateAppFITwithNoReplica(t, appFIT)
25         end
26     until (App is finished)
27 end
28 Function selectTasks(SortedQueue queue)
29     while (locCounter < x × queue.size()) do
30         Task temp = queue.peek();
31         temp.selected = true;
32         RdQ.push(temp.duplicate());
33         numSelected + +; locCounter + +;
34         if (numSelected == target) then return;
35     end
36 end
```

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. To evaluate our work we implement a fault injection mechanism within Nanos. We first describe our injection mechanism in Section 7.2.5. We then present the results of our experiments in Section 7.2.6.

First, we validate the estimations of our model against the results obtained in Monte Carlo simulations. Then, we evaluate App_FIT to see replication costs and performance overheads. After that, we evaluate the efficacy of the Target_Rep Heuristic by comparing it to the optimal

Table 7.5: 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

solutions. In addition, we evaluate its efficiency by evaluating its performance overheads. Finally, we evaluate the performance overheads of selective task replication and its effect on the scalability of the benchmarks. Moreover, by using fault injection we evaluate the scalability under very high fault rates to see also the effect of error recovery on the scalability.

7.2.5 Fault Injection

We use our fault injection mechanism implemented in the runtime to (i) evaluate and stress scalability and overheads of task replication under very high fault rates and (ii) validate the reliability model by accelerated fault injection.

First, we inject faults with fixed rates (probabilities) per single task to stress and evaluate the scalability and overheads of our task replication scheme. This will in effect correspond to higher overall fault rates for the entire system than the current rates. That is, task replication is scalable even when task recoveries and re-executions are performed due to failures. To show

failure scalability, we inject faults in the outputs of the tasks by flipping bits of the outputs. Each flip is a random bit of a random element of the task output. We inject multi-bit flips. Faults are injected at the end of the computation of a task based on the fixed rate. The experiments are repeated 10× and the averages are reported.

Second, our experiments for the validation are done by accelerated fault injection. We do accelerated injection since it is not practical to use current failure rates to validate our model which would require experiments over long periods of time (months). We inject faults at the accelerated rate which is based on the execution time and the memory usage of a benchmark for both failures (crashes) and SDCs. We accelerate fault injection with a rate of 3.6×10^{11} per bit. The rationale behind the acceleration rate is to allow faults occur in a realistic way by taking into account how long a benchmark is executed with the memory usage they require. The fault rate is amplified such that realistic occurrence of both crashes and SDC is achieved in all benchmark executions. We inject faults at the runtime to memory spaces of the tasks by flipping random bits with fault arrivals following the exponential distribution. While the benchmark executes, we then collect the number of crashes and SDCs. If a task is replicated, a crash means both replicas have crashed (crash of a single replica will not lead to a failure as the task computation can continue with the surviving replica). SDCs are measured by comparing the golden run outputs to the produced outputs. The golden run outputs are obtained by running benchmarks without any injections. We use the total number of crashes and SDCs to directly calculate the FIT in billion hours for each benchmark together with its execution time. Finally we map obtained FITs back to the actual FITs by factoring out the acceleration rate.

7.2.6 Experimental Results

We first discuss the model validation results. Then we present our results regarding App_FIT and Target_Rep. Finally, we present the experimental results for task replication.

Model Validation Results

The reliability model from Section 7.2.2 is utilized at runtime for selective task replication to estimate the possible improvement in application's reliability if a task is replicated. To see whether our model is accurate we provide an extensive validation using Monte Carlo simulations. For the Monte Carlo simulations, we execute each benchmark 100× and during its

Table 7.6: Model validation results

Benchmark	Model FIT	Experiments FIT	% Difference	STD
Cholesky	3.31×10^{0}	3.57×10^{0}	7.28%	2×10^{-4}
FFT	1.0×10^{0}	9.2×10^{-1}	8%	3×10^{-4}
Linpack	1.7677×10^{2}	1.7633×10^{2}	5.33%	7.1
Matmul	3.48×10^{-2}	3.28×10^{-2}	5.74%	6×10^{-6}
Nbody	2.5×10^{-4}	2.4×10^{-4}	4%	3.6×10^{-7}
Perlin	2.35×10^{-1}	2.31×10^{-1}	0.002%	3×10^{-4}
Pingpong	1.14×10^{-4}	1.16×10^{-4}	1.75%	1.8×10^{-6}
SparseLU	3.6×10^{-3}	3.9×10^{-3}	7.69%	1.33×10^{-4}
Stream	2.2×10^{-2}	2.1×10^{-2}	4.54%	1.32×10^{-4}

Table 7.7: App_FIT threshold results

Benchmark	Current FIT	Target FIT	Achieved FIT
Cholesky	7260	726	725.68
FFT	3520	352	347
Linpack	29280	2928	2921
Matmul	80	8	7.64
Nbody	20	2	1.96
Perlin	630	63	62.99
Pingpong	20	2	1.96
SparseLU	3230	323	322.99
Stream	33470	3347	3346.95

executions we inject faults. For a successful validation we would expect that the results from the Monte Carlo simulations closely match those that are obtained from our reliability model.

Table 7.6 shows the results for the validation of our reliability model. We perform validation for complete replication of tasks which is the most general version of our model. The table shows the average FIT estimated by our model and by our injection based experiments. It also shows the difference between the estimated FIT and the measured FIT for each benchmark. On average, the difference between them is 4.9%. The model underestimates FIT for Cholesky, Pingpong and SparseLU, and for others it overestimates. Note that FITs reported in the table are in billion hours. We also report the standard deviations of our results for each benchmark in the last column.

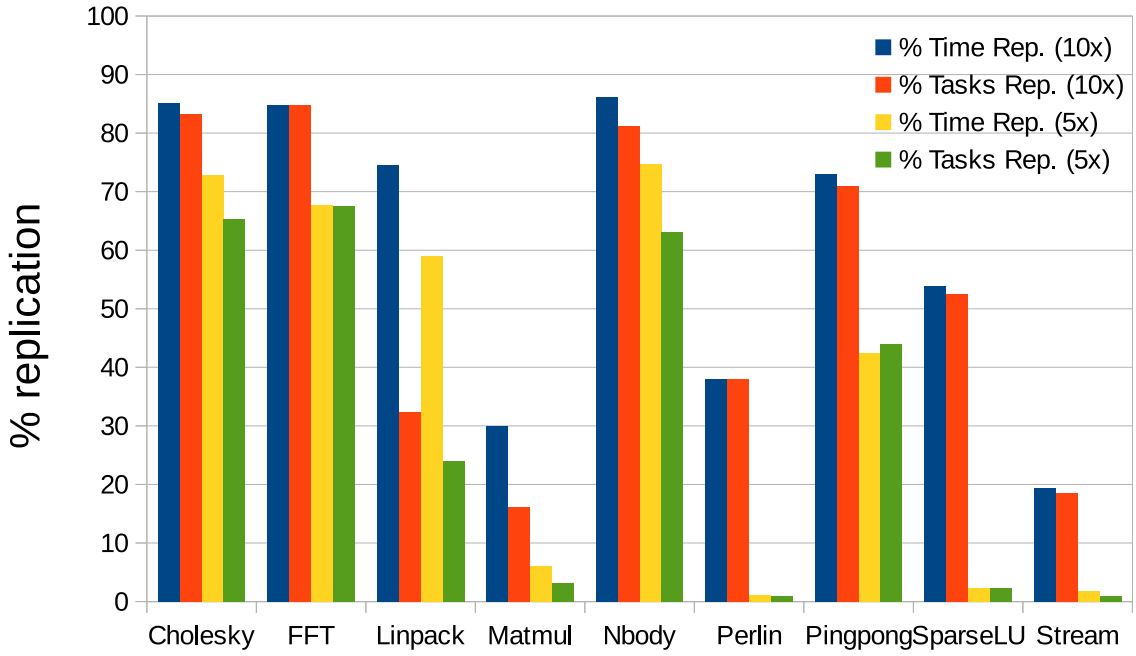


Figure 7.12: App_FIT results

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 [107]. To handle this increase, we will try to decrease the current FITs of our benchmarks by 10× so that the overall application FITs, thus their reliabilities, stay the same. In Section 7.2.2, we explain how we calculate the FITs of the benchmarks used in these experiments.

Figure 7.12 shows the results of the App_FIT experiments. 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× and 5× higher error rates. On average App_FIT replicates only 53% and 30% of the tasks, and 60% and 36% of the computation times to obey 10× and 5× rates respectively. Furthermore, in our experiments App_FIT achieves FITs that are lower and close to the specified FITs. Table 7.7 shows the current FITs of benchmarks, the specified thresholds

and the thresholds that App_FIT achieves.

Takeaway-2: *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.*

In general the task replication percentages and the computation time replication percentages 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 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 that for Perlin and SparseLU there is a significant difference between the resulting replication percentages when 10× and 5× 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 the 5× rates.

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

Evaluation of Target_Rep Heuristic

Target_Rep tries to maximize the reliability of an application while obeying the target percentage of task replication. We assess Target_Rep by comparing its output to the optimum solution to evaluate its efficacy. To get the optimum solution for a specified percentage, say x%, of task replication, we first profile each benchmark and then we sort all the tasks according to their FITs (from smallest to the largest) calculated during profiling and we sort and store these FITs. We choose the first x% of the sorted tasks. The chosen set of tasks is the optimum solution for x% of tasks. Figure 7.13 shows how close our solution is to the optimal one. The x-axis is the percentage of tasks being selected and the y-axis is the difference (in %) between the FITs that would be achieved with the tasks selected for replication by the optimum solutions and the FITs

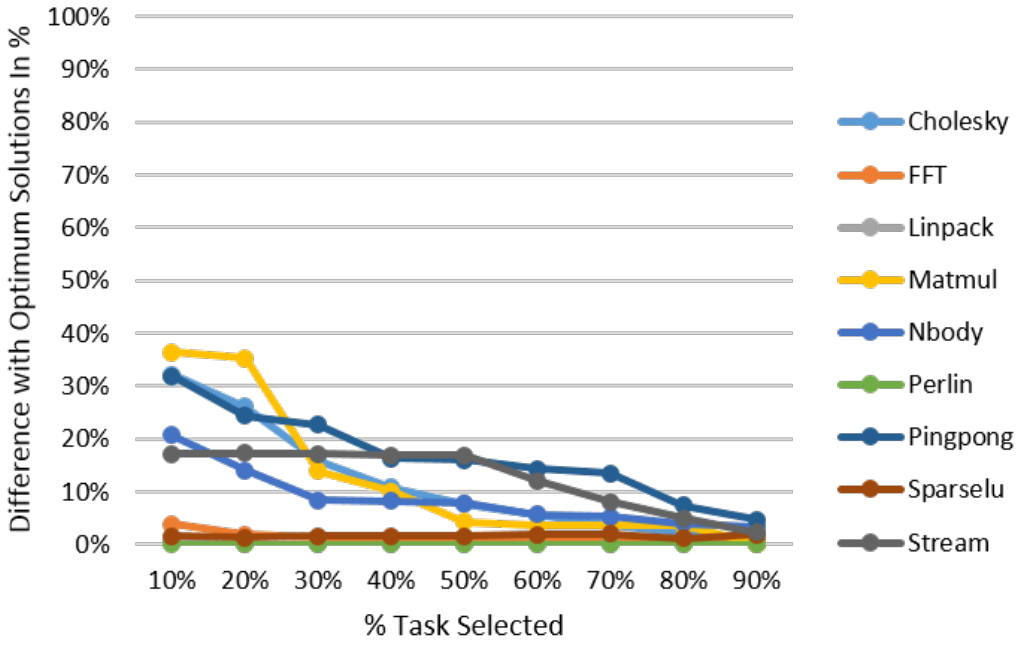


Figure 7.13: Target_Rep results

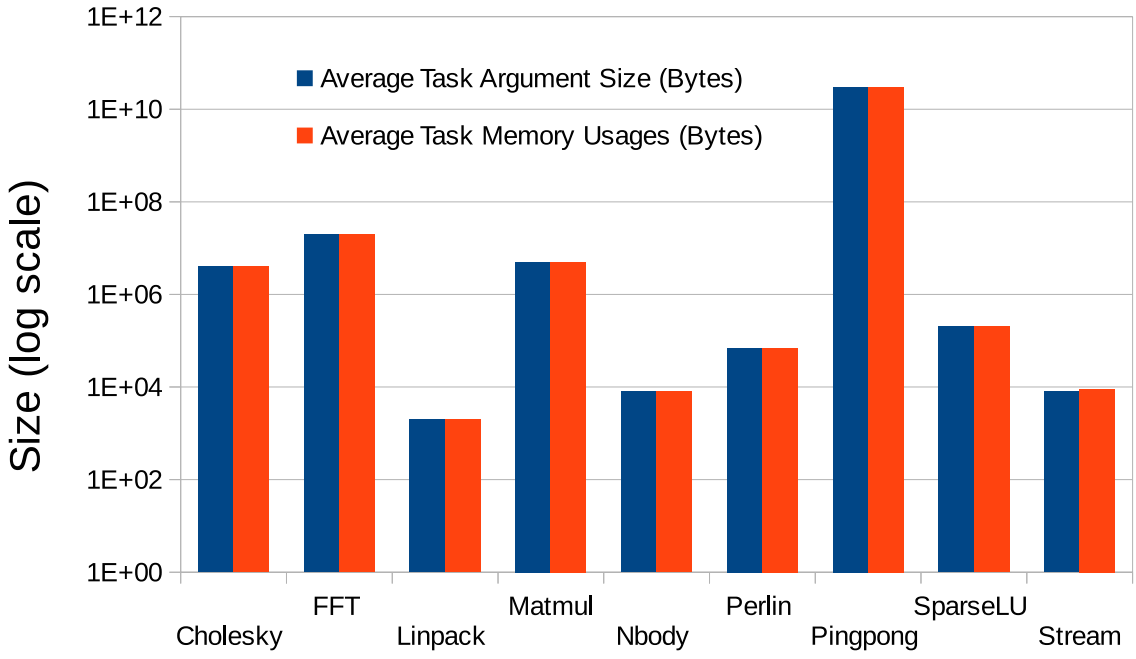


Figure 7.14: Task argument sizes vs. Task memory usages

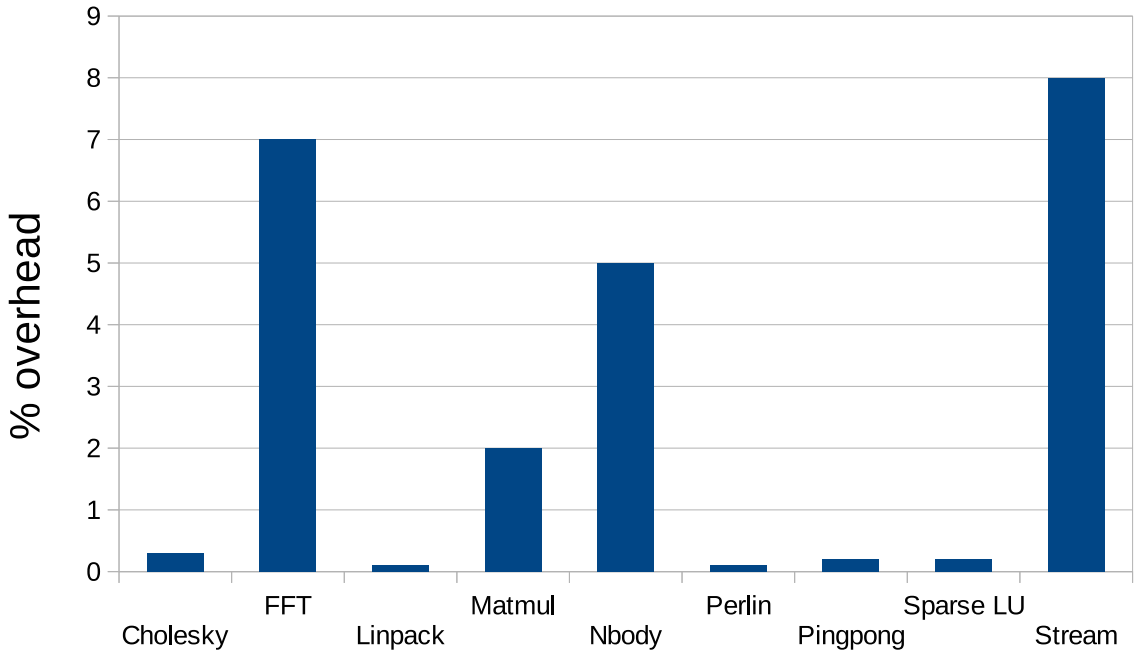


Figure 7.15: Target_Rep overheads

that Target_Rep achieves for a given percentage of task replication. As expected, as the replication percentage increases, the difference between the FITs achieved by Target_Rep and the optimal solution decreases. Overall, Target_Rep achieves close to the optimal solution. When half and more than half of the tasks are replicated (50% replication and more), the difference is only 5% and less than 5% on average respectively.

We now provide additional evaluation to explain in depth why Target_Rep is effective in selecting tasks for replication. Figure 7.14 (log scale) shows the average total arguments sizes and the memory usages for a single task in each benchmark. They are very close; as a result Target_Rep is effective in selecting tasks by using task argument sizes as proxy for the actual memory usages. Note that when a task is starting to execute (i.e., at the point of deciding whether to replicate it) the runtime has the argument sizes. Thus our approach has the advantage of not requiring a separate profile run to collect task memory usages.

Finally, we show the efficiency of Target_Rep by evaluating its overheads with respect to fault-free execution (wall-clock) time. Figure 7.15 shows its overheads. As seen, the overheads are low and the average overhead is 1.2%.

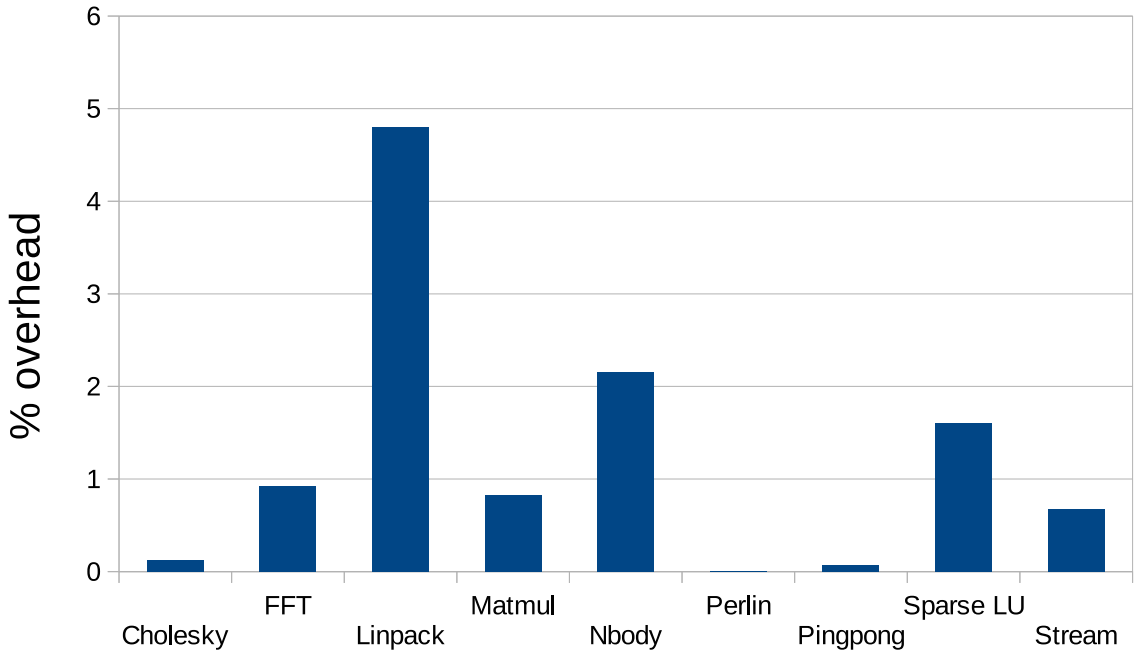


Figure 7.16: Task replication overheads

Evaluation of Task Replication (Overheads, Scalability)

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 [92] to stress-test our task replication design in terms of overheads and scalability. Task replicas are executed on spare cores. Note that the energy overheads are 100% for complete replication. However, the energy overheads decrease considerably when selective replications is used.

Figure 7.16 shows the performance overheads of checkpointing and task output comparison 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

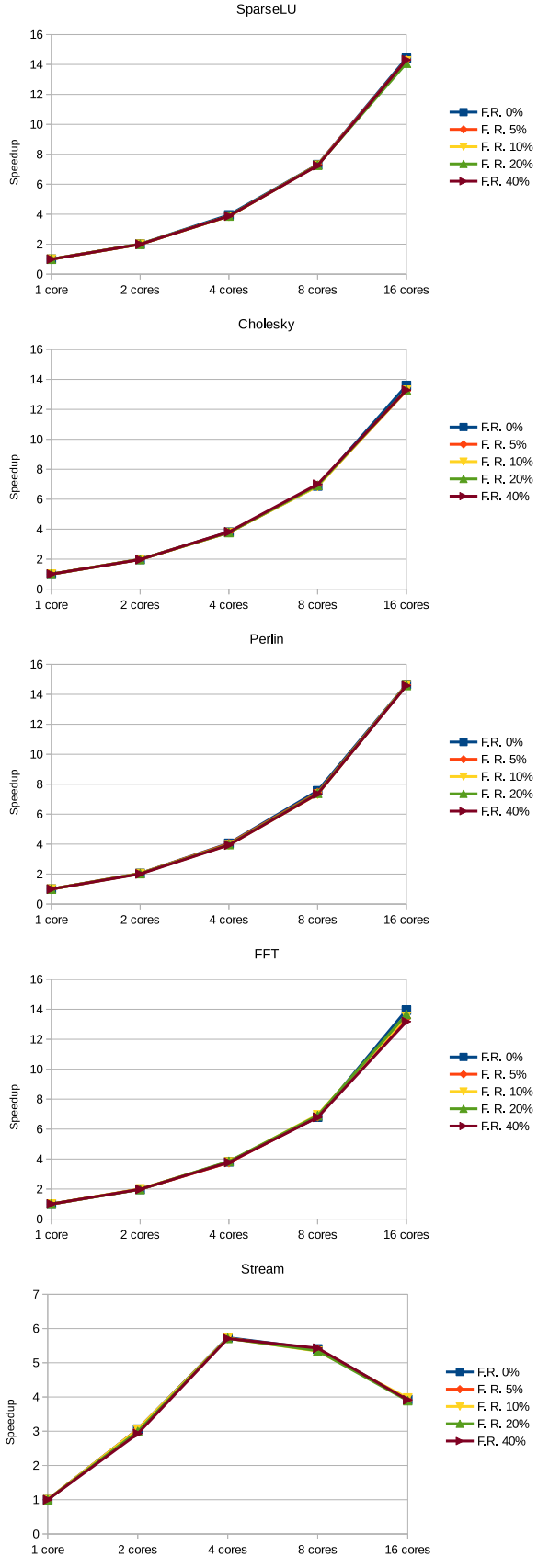


Figure 7.17: Complete replication scalability (shared memory)

assess the effect of fault recovery on the scalability with per task fixed fault rates. Figure 7.17 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 7.18 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.

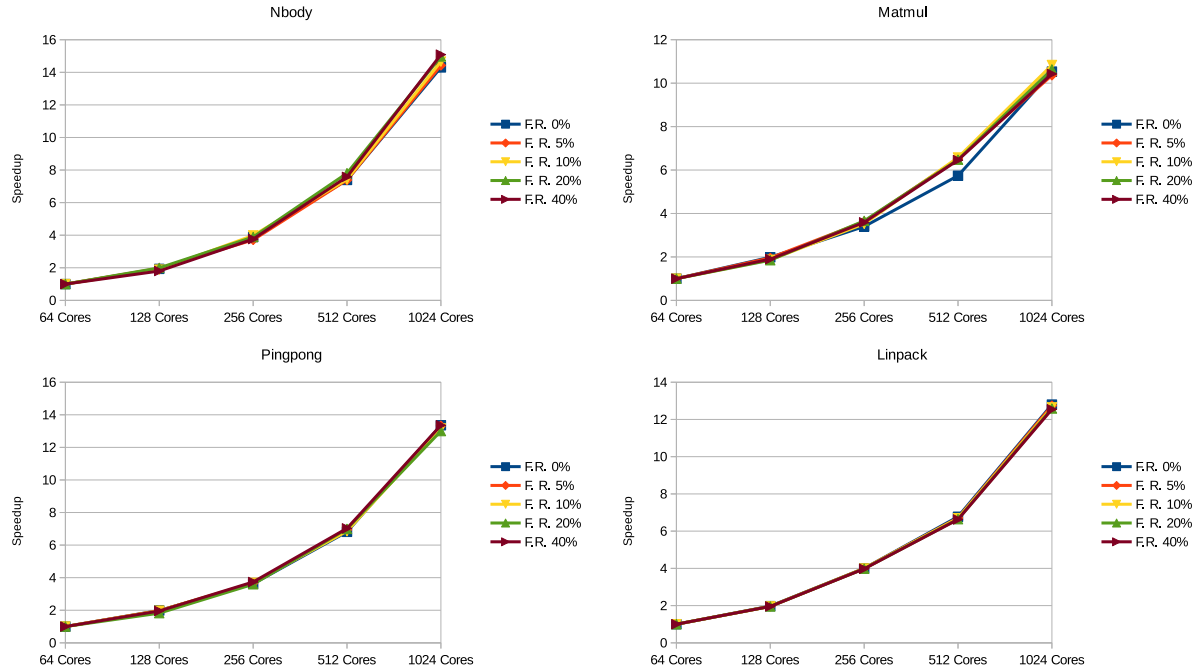


Figure 7.18: Complete replication scalability (distributed)

Takeaway-3: *Overhead and scalability results indicate that task-parallel dataflow programming makes fault-tolerance affordable while enabling design of replication heuristics.*

7.2.7 Conclusion

In this section we propose low-overhead and scalable selective task replication for task-parallel programs to mitigate SDCs and fail-stop errors (DUEs). To achieve selective task replication, we develop and validate a reliability model for task-parallel programs. Based on the model we present two automatic heuristics to select the tasks to replicate for two usages: i) keeping FIT of a program under a given threshold ii) using spare resources for redundancy. Results show that they have low overhead and select tasks in a near-optimum way.

In this research, our key findings were: First, the 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, the relative differences among the impacts of individual application tasks on the application reliability will increase exponentially. That is, certain application tasks will contribute much more than other tasks in increasing the reliability of the entire application if replicated. Third, task-parallelism and dataflow offer key properties making fault-tolerance support for the future HPC systems affordable.

7.3 Conclusion

In this chapter, we presented our task replication heuristics and reliability model for task-parallel dataflow HPC applications. We combined checkpoint/restart and task replication to address both SDCs and fail-stop errors. We showed that our techniques were efficient, low-cost and scalable. We leave the comparison among the formal heuristics and the risk-based and programmer-guided heuristics as future work. In the next chapter, we will focus on the memory reliability of task-parallel dataflow HPC applications.

8

CRC-based Memory Reliability for Task-parallel HPC Applications

8.1 Introduction

In the previous chapters, such as Chapters 4, 6 and 7, we proposed mechanisms to protect the computations of task-parallel HPC applications. In this chapter, we focus on the memory reliability of these applications. This is because it has been established that memory will be one of the most vulnerable system components. In fact, currently memory errors contribute to more than 40% of system failures [82] and memory reliability will be even more crucial due to massive increase in memory capacity in future HPC and exascale systems [32]. Additionally, issues with future transistor scaling will decrease memory lifetimes and make main memory more vulnerable to multiple bit flips [108].

Current HPC memory systems are mostly protected by hardware Error Correcting Codes (ECCs), such as Chipkill [29] which provides recovery for DRAM chip failures. However, we posit that hardware ECCs might not be adequate to cope with memory errors for the future

HPC systems [30, 82]. In particular, Sridharan et al. [30] report that the Chipkill's uncorrected error rate will increase 3.6-70× and consequently it will not be sufficient for the Exascale era. As a result stronger protection for memory errors will be needed. Therefore software-based solutions are needed to cooperate with hardware. These software-based solutions should have low performance overheads, be scalable and should provide multi-bit error detection/recovery. Thus, in this chapter, we introduce a Cyclic Redundancy Check (CRC) based software fault-tolerance mechanism that protects both main memory and cache, and is oblivious to both. Our mechanism is orthogonal to hardware ECCs for main memory and caches. It can be used to piggyback hardware ECCs to reduce the projected Exascale memory error rates to today's acceptable levels.

As stated earlier, task-based parallelism and data-driven execution are becoming widely used to implement HPC applications for achieving higher performance. However, we find that application memory is particularly vulnerable to faults in task-based dataflow applications (Section 8.3) especially when it is idle. This happens since data stays idle for a long time in memory between when it is created by a producer task and computed upon by a consumer task. In comparison, active data (i.e. data active in task computation) is less vulnerable. Therefore, in this chapter we provide fault-tolerance support for memory errors of task-parallel HPC applications where input-only and idle task arguments are of primary focus.

We design and implement a mechanism that, we term CRC, uses software-based Cyclic Redundancy Check [34] codes for strong error correction capability (32-bit consecutive and 5-bit arbitrary errors). *To the best of our knowledge, this is the first time that CRC codes are applied for memory reliability.* Our choice of CRC codes is directed by the fact that they offer strong error detection capability required for the exascale era and there is already existing hardware that accelerates the CRC computation in most of the state-of-the-art HPC systems. Our scheme computes the CRC of tasks outputs for error detection and takes one snapshot for error recovery (Section 8.2). Our runtime-based mechanism does not require any application annotations, recompilation, or OS modifications while protecting application memory.

To compare to our CRC mechanism, we additionally implement two different mechanisms. The second mechanism, that we term SNAP, takes two snapshots of task output data structures and uses majority voting to detect and recover from memory errors. Third mechanism, that we term CHKS, uses 64-bit checksum instead of CRC. With these three schemes we additionally provide a design space analysis. All three mechanisms are highly scalable and have low performance overheads: 9% for CRC, 3% for SNAP and 5% for CHKS. We then show how software

and hardware collaboration helps in terms of overheads, we accelerate CRC calculation leveraging an existing processor instruction in X86 processors (which also exists in IBM and ARM processors) and find that this reduces the overhead from 9% to 1.7% (Section 8.3).

Our CRC mechanism is orthogonal to hardware ECCs such as Chipkill. Chipkill protects system memory and our scheme protects reliability-critical application memory. In the task-parallel dataflow programming paradigm, the programmer typically declares the task input and output data structures. In this paradigm inputs and outputs most often comprise sufficient state to recover from errors. Thus the application memory that certainly needs to be protected is known at runtime. This way we provide efficient and low-overhead memory fault-tolerance while reducing the Chipkill uncorrected error rate significantly and sufficiently. Experiments demonstrate that our CRC scheme decreases the memory vulnerability of benchmarks by 87% on average while providing up to 32-bit burst (consecutive) and 5-bit arbitrary error correction capability.

Our main contribution is the design and implementation of a CRC-based memory fault-tolerance mechanism for task-parallel HPC applications. *To the best of our knowledge, our framework is the first work that complements hardware ECC mechanisms at software for task-parallel dataflow HPC applications.*
Our main contributions in this chapter are:

- Low-cost and highly scalable (fault-detection-only variant) CRC mechanism to detect and recover from memory errors with 9% (7%) performance overhead on average.
- Hardware acceleration that reduces CRC overhead from 9% to 1.7%.
- Mathematical quantification of reliability and detailed comparison of three mechanisms.

Our key research findings in this chapter are:

- Software-based memory reliability is viable for the projected Exascale error rates provided that minimal and critical application data is known, which is the case for the task-parallel data-driven programming paradigm.
- Our CRC technique is orthogonal to hardware ECCs and can complement them. It decreases the uncorrected error rates significantly and sufficiently for the Exascale era.

- Since hardware acceleration is widely available, the performance overhead of CRC scheme can be significantly reduced for most of the HPC systems, which is vital considering the Exascale era and the prohibitive cost of the stronger hardware ECCs.

- Our framework reduces the memory vulnerability in task-parallel data-driven programs significantly. In fact, to our surprise, much more memory is vulnerable outside task computations than within the computations as evident from both the benchmark analysis (Section 8.3.1) and the memory vulnerability analysis (Section 8.3.3).

8.2 Design and Implementation

We present our failure model in Section 8.2.1. We then discuss the design and implementation of SNAP, CHKS, and CRC in Sections 8.2.2, 8.2.3 and 8.2.4 respectively. In Section 8.2.5 we present the hardware acceleration of CRC. Our CRC implementation is open source and can be downloaded from [109].

8.2.1 Failure Model

Our failure model covers soft, intermittent and hard errors that can occur in application data residing in memory structures such as main memory and caches. In particular, our failure model targets memory errors that occur in task arguments not affected by on-going task computations: We provide fault detection and fault recovery support for i) memory errors occurring in task arguments when they are not modified by the task computations, i.e. input only arguments, and ii) when they wait idle (not accessed) in the memory. Therefore we protect all memory including the active memory that is not modified by the computation. The only exceptions are the task output memory (output arguments) and temporary memory. However protecting these two exceptions implies the protection of computation. As we will see from the benchmark analysis (Section 8.3.1) and the memory vulnerability analysis (Section 8.3.3), these two exceptions indeed constitute small fraction of memory that is vulnerable in task-parallel dataflow programs.

Our failure model also includes *burst errors*: They are consecutive bit errors in task arguments residing in memory. Our scheme is oblivious to how these consecutive errors are spread across DRAM memory or caches. They may spread across devices and not just multiple words.

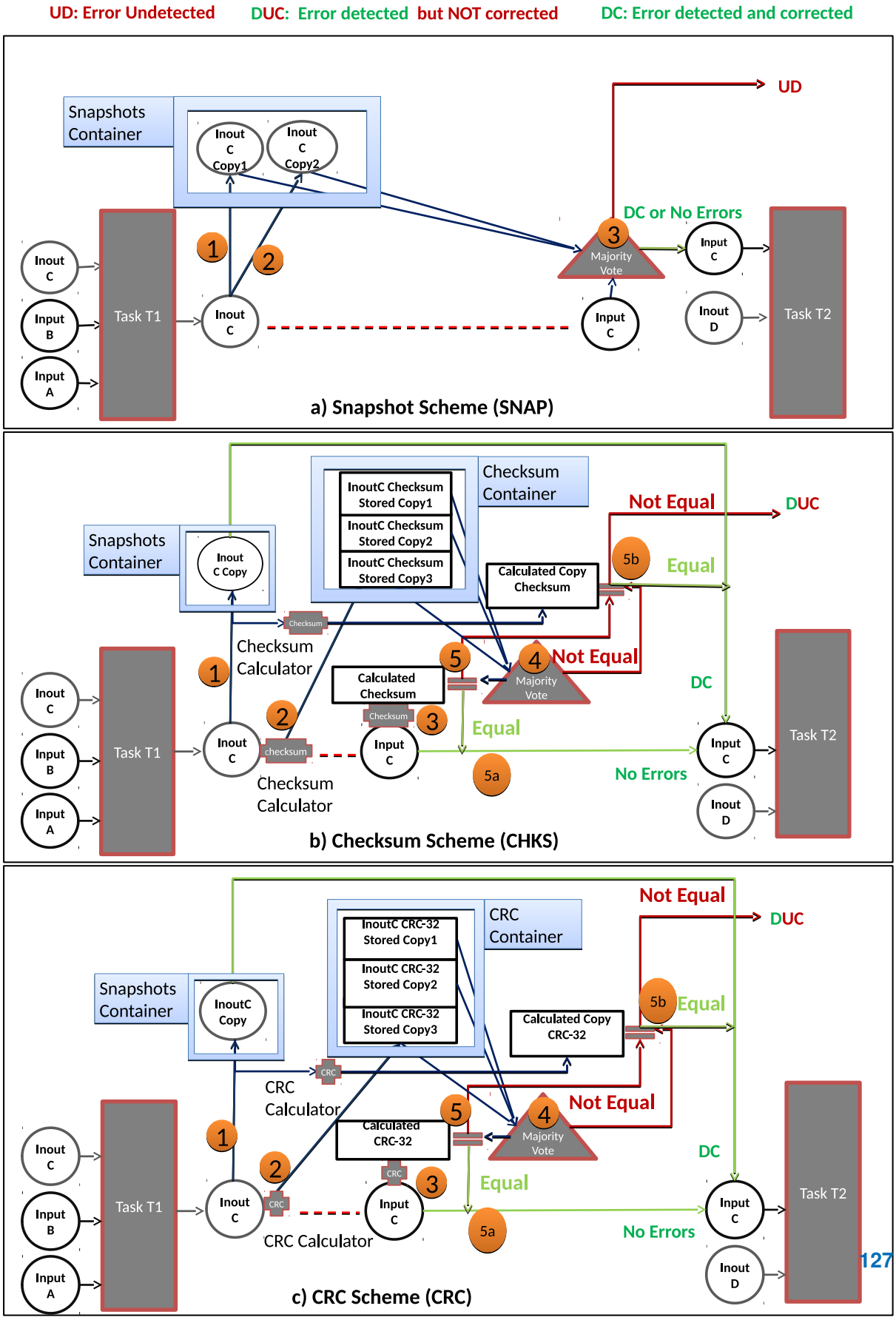


Figure 8.1: The overall flow of the three memory reliability mechanisms.

However, it is crucial to protect application-level or algorithmic data regardless of their storage in devices or cache lines. Our mechanism is designed to achieve this goal. We protect *algorithmic data (objects)* rather than architecture-level data or memory space.

8.2.2 Snapshots (SNAP) Design

Figure 8.1(a) summarizes the color-coded snapshots scheme (SNAP). Solid background is used for the computation of tasks and voting process to indicate that they occur at runtime. Task inputs and outputs, their copies (snapshots), and the snapshot container, which is implemented via a concurrent hashtable, reside in memory. In SNAP, when a task produces an output after its computation finishes, we take two snapshots of the output to the main memory (① and ②). Then when another subsequent task is about to use this particular output as its input, a majority vote via bitwise comparison among the two stored snapshots and the original output is taken ③. Errors that occur in the same locations in at least two copies of the output will not be detected. Other errors will be detected and corrected.

Fault-detection-only SNAP Design: The difference between SNAP and fault-detection-only SNAP is that we take only one snapshot of a task output when the task finishes. Then when another subsequent task is about to use this particular output as its input, the original output and the stored snapshot is compared. If they agree, the computation continues as usual. If they do not agree, the application aborts. However, this can be easily adapted to provide different options such as raising an exception to be caught by an application specific handler.

8.2.3 Checksum-based Scheme (CHKS) Design

Figure 8.1(b) summarizes the color-coded scheme. Again, solid background is used for the computation of tasks and voting process to indicate that they occur at runtime. Similarly, solid background is used for Checksum computation as well as their comparisons to indicate that they occur at runtime. Task inputs and outputs, their copies (snapshots), Checksums, the snapshot and the Checksum container, which are implemented with concurrent hashtables, reside in main memory. In this design, when a task produces an output after its computation finishes, we take one snapshot of the output to the main memory ①. In addition to taking a snapshot, we compute 64-bit checksum of the output: We go over the output 8-byte (64 bits) by 8-byte and accumulate the sum of the 8-bytes as the 64-bit checksum. We then store three copies of

the checksum in three hashtables to ensure that it is not corrupted while staying in memory ②. Then when another subsequent task, whose input is this particular output, is ready to be executed, the checksum of the output is recomputed ③ and compared to the checksum obtained by the majority vote among its three stored copies (④ and ⑤). If they match, the computation of the subsequent task starts with the output as its input ⑤a, which means no errors are detected. If they do not match, then the stored snapshot is retrieved and its checksum is calculated. This checksum is compared to the checksum resulting from the voting ⑤b. If they match, the snapshot is taken as the input of the task, which means error(s) are detected and corrected (if error(s) occurred in the snapshot or the original output do not lead to the same checksum). If they do not match, the application detects the errors but cannot correct them, thus exits.

Fault-detection-only CHKS Design: The difference between CHKS and fault-detection-only CHKS is that when the task finishes, no snapshot of its output is taken, its checksum is computed and three copies of the checksum is stored. Then when another subsequent task is about to use this particular output as its input, the checksum of the original output is recomputed and compared to the checksum which obtained by the majority vote among its three stored copies. If they agree, the computation continues. Otherwise the application aborts.

8.2.4 CRC-based Scheme (CRC) Design

The design of CRC with one snapshot is the same as that of CHKS except that instead of 64-bit checksum of task outputs, CRC-32 is utilized. CRC-32 computation is implemented with pre-calculated CRCs for each possible byte and stored in an array. There are $2^8 = 256$ possible values for a byte. For each possible value, CRC is pre-calculated when the program starts according to the CRC generator polynomial. To calculate CRC for an output, a byte by byte iteration over the output is performed and the final CRC is accumulated. The final CRC is accumulated by XOR and shift operations to augment each next byte correctly. We propose two different CRC-32 versions where we use two different polynomials to provide flexibility and adaptation in terms of fault detection capabilities for different applications with different task argument sizes. For the applications that have shorter task inputs and outputs, we propose to use Koopman's polynomial [36], 0xBA0DC66B, which has a hamming distance of 6 for up to 16K bits for a better fault detection capability. For the applications that have longer task inputs and outputs, we propose to use Castagnoli's [35] polynomial, 0x11EDC6F41, which has a hamming distance of 4 for very long data words (up to 2^{31} bits). Figure 8.1(c) shows the

Algorithm 5: Hardware-based Parallel CRC Algorithm

```
   Input: int* task_argument: Task Argument Pointer.
   Output: final_crc: The Final CRC of the Task Argument.
   /* Assuming that the task argument size is 1024 (= 336 * 3 + 16) bits          */
   /* For larger sizes, the argument is divided and processed in 1024-bit chunks.
      */
 1 extern int mul_table_336[256] // Precalculated CRCs
 2 extern int mul_table_672[256] // Precalculated CRCs
 3 long crc1, crc2, final_crc, tmp;
 4 crc2 = final_crc = 0;
   // Process first 8 bytes here for better pipelining
 5 crc1 = _mm_crc32_u64(0, task_argument[0]);
 6 for (int i = 0; i < 42; i++) do
       // The following three instructions are executed in parallel
 7     crc2 = _mm_crc32_u64(crc2, task_argument[43 + i]);
 8     final_crc = _mm_crc32_u64(final_crc, task_argument[85 + i]);
 9     crc1 = _mm_crc32_u64(crc1, task_argument[1 + i]);
10 end
   // merge in crc2
11 tmp = task_argument[127];
12 tmp ^= mul_table1_336[crc2 & 0xFF];
13 tmp ^= ((long)mul_table_336[(crc2 >> 8) & 0xFF]) << 8;
14 tmp ^= ((long)mul_table_336[(crc2 >> 16) & 0xFF]) << 16;
15 tmp ^= ((long)mul_table_336[(crc2 >> 24) & 0xFF]) << 24;
   // merge in crc1
16 tmp ^= mul_table1_672[crc1 & 0xFF];
17 tmp ^= ((long)mul_table_672[(crc1 >> 8) & 0xFF]) << 8;
18 tmp ^= ((long)mul_table_672[(crc1 >> 16) & 0xFF]) << 16;
19 tmp ^= ((long)mul_table_672[(crc1 >> 24) & 0xFF]) << 24;
   // return final merged CRC
20 return (int) _mm_crc32_u64(final_crc, tmp);
```

scheme. The design of fault-detection-only CRC is the same as that of fault-detection-only CHKS except that instead of 64-bit checksum, CRC-32 of task outputs is used to detect errors.

8.2.5 Hardware-assisted Acceleration of CRC

Even though the software-based overheads are low, we propose and implement hardware optimization of CRC-32 by utilizing the existing Intel CRC-32 Instruction [110] ($\_mm\_crc32\_u64$) introduced in the Intel Core i7 Processor at runtime. We note that the CRC polynomial 0x11EDC6F41 we chose for our design is the same polynomial that the instruction implements which enabled us to implement and evaluate this optimization. IBM [111] and ARM [112] have already implemented this instruction staring with their Power8 and with ARMv8 64-bit architecture respectively. AMD implemented PCLMULQDQ instruction starting with Bulldozer processors [113] which can be used to implement the CRC calculation. Hence hardware acceleration can be implemented for the most of the HPC systems.

We implement two versions of this acceleration. The first one is solely using the instruction

during the CRC calculation instead of our software implementation. The second implementation leverages instruction-level parallelism (ILP). This method splits each 1024 bits into three separate segments (since the CRC instruction has a throughput of 1 cycle and latency of 3 cycles) and calculates the CRC of each segment. Since there are no data dependencies between segments, each can be processed in the execution pipeline, out-of-order, and in parallel leveraging ILP when necessary. Then it recombines these CRCs into a single CRC which is the CRC of all 1024 bits. The recombination is done through xor and shift operations. Algorithm 5 shows details.

8.3 Evaluation

We implement our techniques in Nanos runtime [21]. We run our experiments in MareNostrum III Supercomputer [90]. We run experiments on both task-parallel shared-memory and task-parallel distributed MPI benchmarks listed in Table 8.1 [98]. We include and evaluate hybrid, task-parallel distributed applications to show i) our mechanism is well-suited for such hybrid MPI programs and ii) it incurs low overhead and is highly scalable at large scale and for high core counts. We obtain all results by running each single case 10 times and take the averages. In our experimentation 16 cores are used for shared-memory and 1024 cores on 64 nodes are used for distributed benchmarks. In our evaluation, we will highlight the takeaway lessons that we conclude from our experimental results.

We provide three-fold evaluation. First we analyze the behavior of task scheduling at runtime and memory characteristics of our benchmarks in Section 8.3.1. This analysis will show how crucial it is to provide reliability for memory errors occurring in task arguments while staying idle in memory or being not modified by the task computations. Second, we present performance and memory overheads, and scalability results in Section 8.3.2. Third, in Section 8.3.3, we evaluate the reliability of three mechanisms. We conclude the evaluation section with the Section 8.3.4 where we compare and contrast the three schemes.

8.3.1 Benchmark Behavior Analysis

Table 8.2 shows the average number of tasks in the ready queue, the total number of inputs and outputs and the temporarily idle memory amount throughout executions of benchmarks. For distributed benchmarks, the statistics are per single node. We obtain these statistics by

Table 8.1: Details of Benchmarks

Shared Memory Benchmarks	
Sparse LU	LU decomposition Matrix size 12800x12800 doubles
CG	Conjugate Gradient Solver Matrix size 4096x4096 doubles, with 16 iterations
Cholesky	Cholesky factorization Matrix size 16384x16384 doubles
FFT	Fast Fourier Transform Array size 16384x16384 complex doubles
Perlin Noise	Noise generation to improve realism in motion pictures Array of pixels with size of 65536 (1500 iterations over it)
Knapsack	Parallel 0-1 Knapsack by Dynamic Programming Number of weights 100000000, Knapsack Capacity 1000
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
Ping-Pong	Computation and communication between pairs of processes Array size 65536 doubles
Linpack	High-Performance Linpack Matrix size 131072 doubles, 8x8 grid

Table 8.2: Tasks and Memory Statistics

Benchmarks	# Ready tasks	# Inputs	# Outputs	Temporally idle sizes(KB)
Sparse LU	136	20507	1055	87876
Perlin	113	33	33	57757
CG	31	619776	1	487
Knapsack	2581	9586	9794	486660
FFT	211	127	127	7082845
Cholesky	109	10643	1050	490010
Matmul	32	191	68	680333
Nbody	96	191	4	4026
Linpack	80	384	384	56304
Pingpong	16	181	65	2666

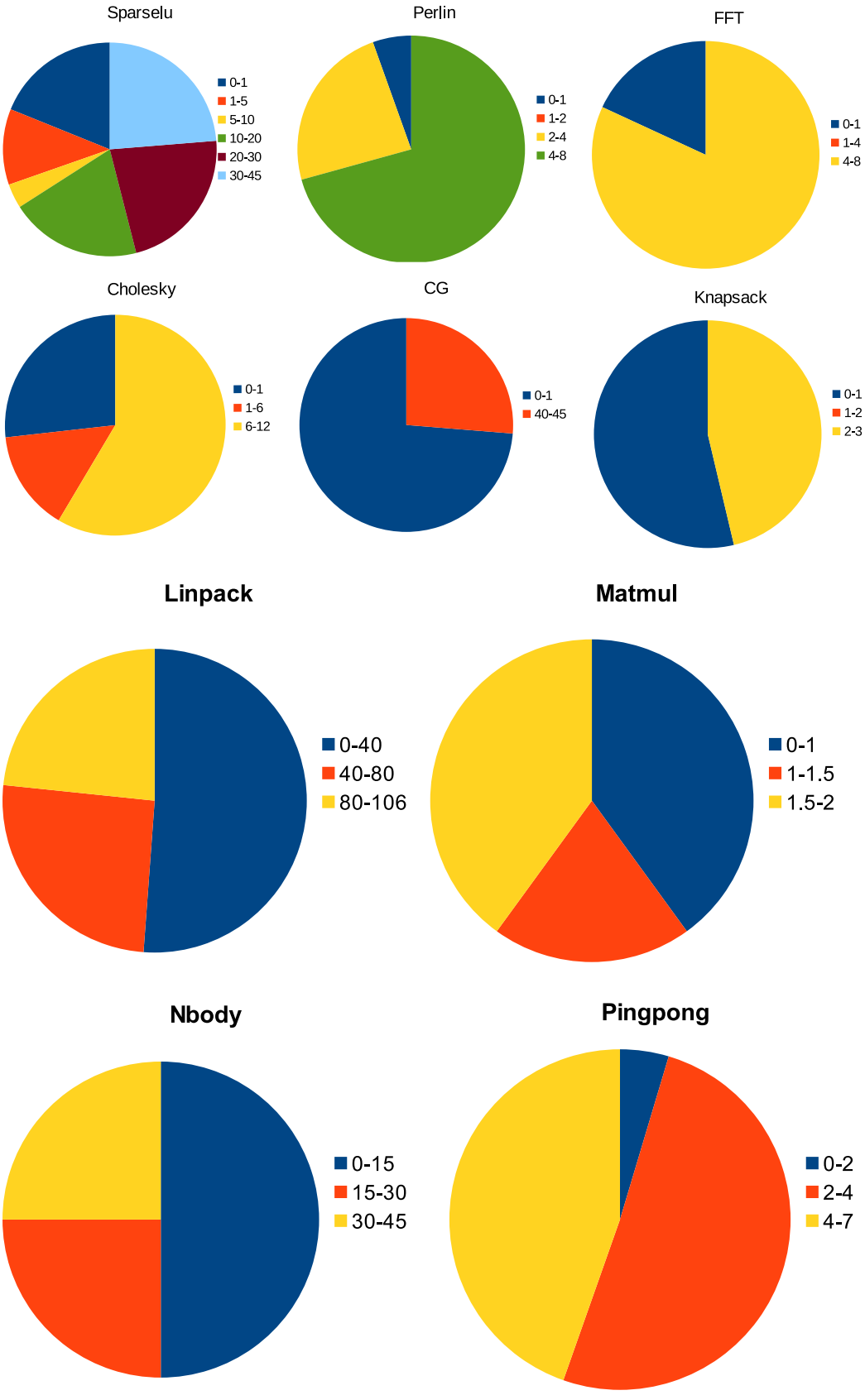


Figure 8.2: Arguments waiting idle or being used but not modified by task computations

sampling and profiling the executions. The temporarily idle memory amount shows the total size of the application memory not accessed by any computation due to the lack of sufficient computation resources on average i.e. waiting idle in main memory, which is due to what we call *look-ahead capabilities* of runtimes for task-parallel applications: Runtimes, in particular task-parallel ones, usually produce more work than computation resources could consume attempting to exploit as much as parallelism they can. We see that *look-ahead* causes huge amount of idle memory blocks and significant number of idle tasks while only about 16 tasks (1 thread per core) are executed at any time. This means that considerable amount of memory is left vulnerable. In terms of input argument numbers, we see that all benchmarks have a high number of inputs and large amount of memory that is not modified by their corresponding task computations. This is due to the fact that in task-parallel applications, most commonly there are task arguments which are only inputs. This shows providing fault-tolerance for memory, that is used but not modified by the computation, is significantly crucial.

In Figure 8.2 we show the time for each task argument in which it is used but not modified by any task computation (i.e. it is input-only) and/or is waiting idle in memory throughout the application computation, i.e. is vulnerable to errors. The x-axes show the percentage of task arguments within certain time intervals. The y-axes show the time intervals in seconds. Since the characteristics of each benchmark are different, we use different time intervals to provide better insight about the benchmarks in the figure. In Sparselu, we see that about 65% of task arguments are vulnerable more than 10 seconds in memory where the benchmark takes 40.4 seconds. In CG, about 25% of the arguments are vulnerable 40 to 45 seconds and CG takes about 45.3 seconds. In Perlin, about 70% of the arguments are vulnerable 4 to 8 seconds and Perlin itself takes about 7.3 seconds. In Cholesky, close to 60% of the arguments are vulnerable more than 1 seconds where Cholesky takes about 12 seconds. In FFT, more than 80% of the arguments are vulnerable 4 to 8 seconds and FFT takes about 9 seconds. Finally, in Knapsack, about 65% of arguments are vulnerable 2 to 3 seconds and Knapsack takes 3.1 seconds. In Linpack, we see that majority of task arguments is vulnerable 100 to 105 seconds where the benchmark takes 111.5 seconds. In Nbody, half of the arguments are vulnerable less than 15 seconds, the other half of the tasks are vulnerable than 15 seconds and Nbody takes about 45 seconds. In Pingpong, about half of the arguments are vulnerable 2 to 4 seconds and more than 40% are vulnerable 4 to 7 seconds. Pingpong takes about 7 seconds. In Matmul, there is symmetric and even distribution of arguments being vulnerable in memory. Matmul takes about 2 seconds. We therefore conclude that most task arguments in all benchmarks are left

vulnerable in memory for considerable amount of time.

8.3.2 Overhead and Scalability Results

Figure 8.3 shows the performance overheads of three mechanisms for shared-memory benchmarks with respect to fault free executions for 16 cores. As seen, SNAP incurs less overhead than CHKS and CHKS incurs less than CRC. Other than FFT and Cholesky, we see that the overheads are less than 9%. The issue with FFT and Cholesky is that their tasks are coarse-grained and as a result they do not overlap taking snapshots and/or calculating checksum/CRC well with the computation as much as other benchmarks do. However, increasing task granularity can resolve this issue. Figure 8.3 also shows the overheads for distributed applications for 1024 cores. We see that they are quite small, all of which under 4%. This is due to the fact that in task-based distributed programming models, MPI calls are taskified and the runtime effectively overlaps the task computations with MPI communication together with taking snapshots and/or calculating checksums/CRCs. Different from shared memory benchmarks, for Linpack and Pingpong, SNAP is more costly than CHKS and CRC. For Pingpong, the overheads are quite small which makes it is not relevant to use which mechanism other than the considerations for stronger fault detection capability. For Linpack, CHKS is the right decision when stronger error capabilities such as that of CRC-32 is not needed. We note that the overheads stay similar for different number of cores for all benchmarks. The figure also shows the overheads for fault-detection-only mechanisms.

Figure 8.4 shows the overheads of CRC hardware acceleration techniques. Both implementations significantly decrease overheads but especially the parallel version. On average they incur 4.8% and 1.7% overhead respectively.

Figure 8.5 shows the scalability of three mechanisms for all benchmarks. The x-axes show the number of cores. The y-axes show the speedups over 1-core cases when they are enabled. The baselines are different: 1-core execution in the respective mechanism. It also includes the speedups for the benchmarks when no mechanisms is enabled (indicated as “original”). We see that all benchmarks scale well under our benchmarks and they do not disturb the scalability of them when they are enabled. Figure 8.5 also shows the scalability for distributed benchmarks, i.e. speedups of 1024 cores over 64 cores. *We see that three mechanisms are highly scalable at large scale and with high core numbers.* Speedups of fault-detection-only variants are similar and omitted.

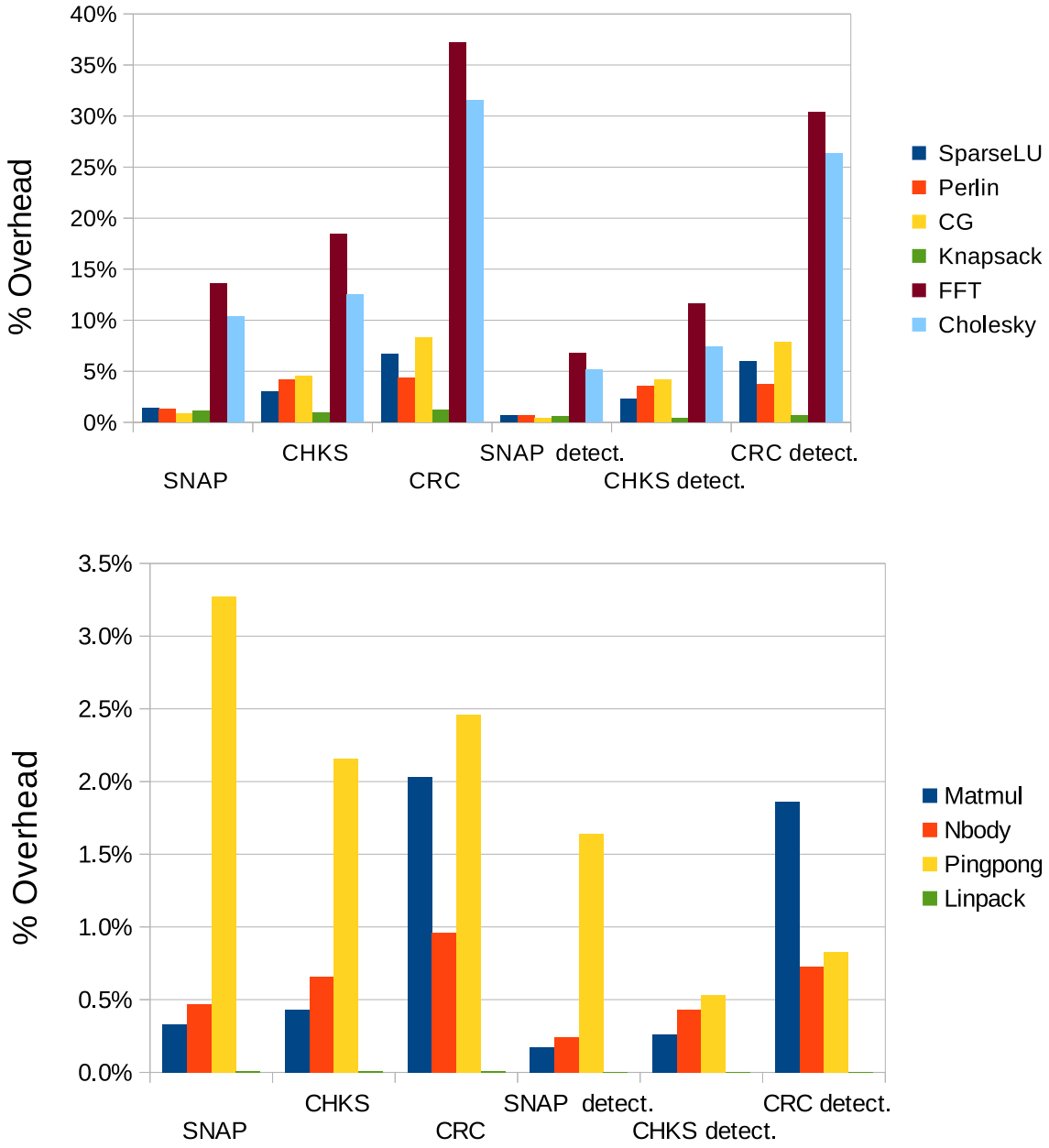


Figure 8.3: Performance overheads of our schemes

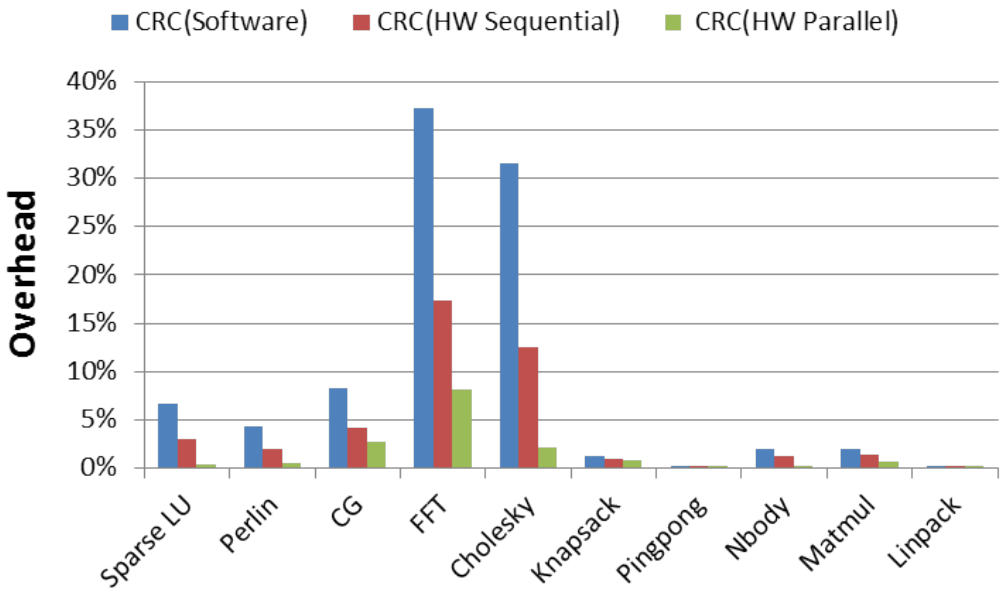


Figure 8.4: CRC Hardware-based overheads

The memory usage of three mechanisms, i.e., memory usage due to taking snapshots, or checksums/CRCs for task arguments ranges from 1.6MB to 8.5 GB depending on the input size and the benchmark for shared-memory applications. CHKS and CRC have similar memory usage and SNAP has memory usage roughly about twice as much as CHKS and CRC since it takes one more snapshot for each task argument. The memory usage of fault-detection only SNAP roughly gets halved and significantly drops for fault-detection only CHKS and CRC since no snapshots are taken. The memory usage for distributed applications ranges from 34KB to 330 MB per single node where applications run on 64 nodes. On average, SNAP incurs 52% and 26% memory overheads for fault detection and correction, and detection-only variants respectively. Note that due to the data awareness at runtime the memory overhead of SNAP is not 200% but 52%. We omit the detailed memory usage statistics for brevity.

Concurrent hashmaps are used to store snapshots and CRCs. This technique significantly reduces the number of memory transfers used for snapshots and CRCs. This is because if a snapshot of a memory location is taken before, another snapshot will not be taken. Coupled with the hardware-accelerated CRC computation, the energy and power consumption of our CRC scheme is minimized with optimized memory management.

Takeaway 1: *Performance and memory usage results show that software-based memory*

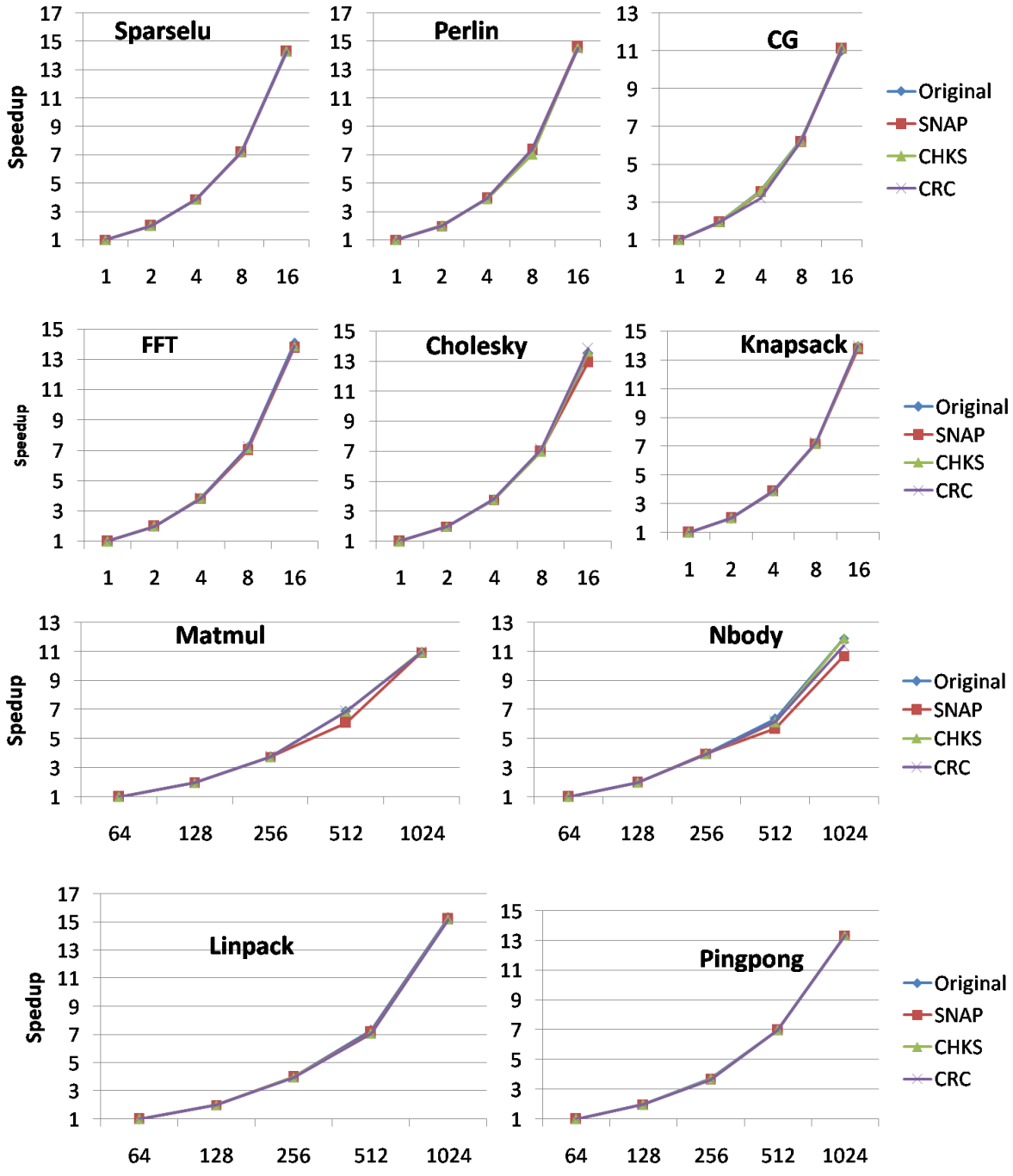


Figure 8.5: Scalability of the benchmarks

fault-tolerance is feasible in the Exascale era in terms of overheads given that data awareness is present such as in task-parallel dataflow paradigm.

Takeaway 2: *Hardware acceleration of CRC calculation drastically reduces the overheads and is widely available.*

8.3.3 Reliability of Three Mechanisms

In this section, we first present the reliability of the three mechanisms by quantifying them by how much they reduce Chipkill's undetected error rates in the Exascale era which are projected by Sridharan et al. [30]. Then we perform memory vulnerability analysis of the benchmarks.

Reduction Factors of Three Mechanisms

In this section we study the *reduction factors* of the three mechanisms. The *reduction factor* of a mechanism refers to the ratio of the Chipkill's error rate to the mechanism's error rate. To calculate the reduction factors, we need to estimate the single-bit error rate of Chipkill in the Exascale era, which we denote by p. For this purpose we use the failure rates from the work of Sridharan et al. [114], for the Cielo supercomputer which has Chipkill-correct for main memory. In particular, we use the error rate which is 0.044 FIT (Failures in Time in billion hours) per Mbits in Table 1 from their work. In addition, following Sridharan et al. [30], Chipkill will have 3.6-70× increase in its uncorrected error rate. Therefore Chipkill's undetected error rate will be 1.76 FIT/Mbits on average and thus $p = 1.68 \times 10^{-6}$. In our experiments, since task arguments vary in size across benchmarks and are in the order of KBs, we choose 1 and 8 KBs to investigate the impact of different data sizes.

CRC's reduction of Chipkill's undetected error rates: We calculate the CRC's reduction for arbitrary k-bit errors through *Hamming Weight ($HW(N,k)$)*: The number of k-bit errors that a coding scheme (in our case the CRC-32) cannot detect for N-bit data. Then the formula for calculating the reduction of CRC scheme in Chipkill's error rate:

$$\frac{\binom{N}{k} \times p^k \times (1-p)^{N-k}}{HW(N,k) \times p^k \times (1-p)^{N-k}} = \frac{\binom{N}{k}}{HW(N,k)} \tag{8.1}$$

where $\binom{N}{k}$ is the possible number of k-bit errors in N-bit data. Our CRC-32 scheme has an error detection capability having a HD (Hamming distance) of 6 unless Castagnoli's polynomial is

used for data with much larger sizes in which case the HD is 4 as discussed in Section 8.2.4. Thus it detects all errors of 5 bits or less. Moreover, CRC detects all odd number of errors since the CRC polynomials we choose have $x + 1$ factor. Hence the reduction factor for these errors is infinite i.e. these errors are completely eliminated.

Additionally, for 4 and 6-bit errors we consult Koopman et al. [115] where they experimentally work on determining the HWs for various CRC polynomials for different sizes of data. For the polynomials 0xBA0DC66B and 0x11EDC6F41, HW(1024,4) is reported zero. For 0xBA0DC66B (HD=6) that we propose for short data, HW(1024,6) is found to be 945102. For 0x11EDC6F41 (HD=4) that we propose for long data, HW(8192,4) is reported 46252. For this polynomial HW(8192,6) is yet to be found experimentally [115]. Figure 8.6 shows the CRC's reduction factors for 4 and 6-bit errors. Overall, the reduction factors are very high leaving very low probability for undetected errors.

CRC scheme can detect and correct all 32-bit and less than 32-bit burst errors, hence the reduction factor is infinite for these errors. For k-bit burst errors where $k > n$, the reduction factor is 2^n where n is the degree of the CRC polynomial which is 2^{32} in our case.

SNAP's reduction of Chipkill's undetected error rates: SNAP fails if there are at least two bit flips in the same positions in different snapshots for a task argument. Therefore SNAP's reduction factor for the undetected error rate of Chipkill by the following formula:

$$\frac{1-(1-p)^N}{1-[(1-p)^3+3p(1-p)^2]^N} \tag{8.2}$$

where N is the total number of bits in a task argument. Figure 8.6 shows the reduction factors of SNAP for Chipkill's undetected error rates with different increases in Chipkill's error rates. As seen, the larger the data size or the higher the increase in error rate, the lower the reduction is, as expected.

CHKS's reduction of Chipkill's undetected error rates: CHKS's reduction factor depends on the detection capability of 64-bit CHKS. This capability depends on the number and the positions of errors. Our checksum scheme is an addition checksum as opposed to the XOR checksum. It fails only if there are even number of *compensating bit errors* in the same positions in different checksum blocks. That is, even if errors occur in the same positions of different blocks, they will be detected if the original values are the same (all 0 or all 1) since the carry will be propagated (not true for XOR checksum). But if they have different values, then the CHKS will not detect. As a simplified example, assume that the checksum is just 4-bit,

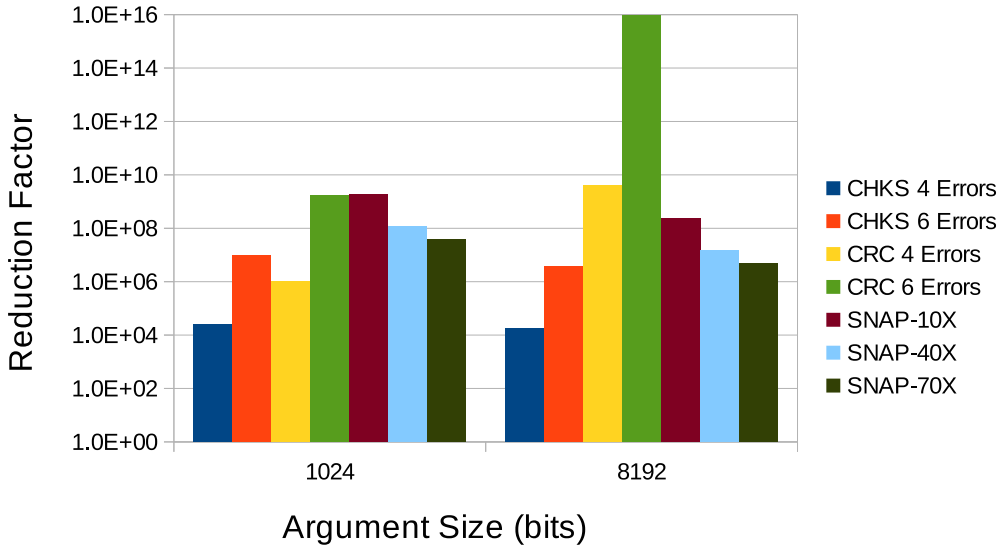


Figure 8.6: Reduction factors of our schemes

therefore the block size is 4. Further assume that we have the following two blocks of data: 0010 and 0100 whose checksum is 0110. Now, if two errors occur such that 0010 becomes 0011 and 0100 becomes 0101, then CHKS will detect the errors since the new checksum is 1000 even though they were both in the first position. However, if the errors are *compensating*, that is, assuming the two errors occur such that 0010 becomes 0110 and 0100 becomes 0000, the new checksum is 0110 which is the same as before and the errors will go undetected. Since two bit positions have 4 possible values and only two of them (they should have opposite values) can cause undetected errors, only half of even errors occurring in the same positions will lead to undetected errors. Moreover, since we have a checksum of 64-bit, for k errors to occur in the same position, there are $\binom{64}{k/2}$ combinations in a single-block and $\binom{\frac{N}{64}}{k}$ combinations across the entire data. However, there is a corner case regarding the most significant bits (MSBs). The even errors occurring in MSBs will not be detected regardless whether they are compensating or not if they are not carefully handled: For instance, in our running example if the blocks become 1010 and 1100, the new checksum will be the same, 0110, due to the overflow. However this overflow can be detected in software by checking register flags thus this exception for the MSBs can be eliminated. With this discussion the reduction factors of

64-bit CHKS in Chipkill's error rates for k errors are calculated:

$$\frac{\binom{N}{k} \times p^k \times (1-p)^{N-k}}{\frac{1}{2} \times \binom{64}{k/2} \times \binom{\frac{N}{64}}{k} \times p^k \times (1-p)^{N-k}} = \frac{\binom{N}{k}}{\frac{1}{2} \times \binom{64}{k/2} \times \binom{\frac{N}{64}}{k}} \tag{8.3}$$

where N is the total number of bits in a task argument, k is the number of errors occurring. Figure 8.6 shows the CHKS's reduction factors for 4 and 6-bit errors (2-bit errors are already mitigated by Chipkill). As seen, the reduction factors are high enough to decrease Chipkill's predicted exascale rates to today's acceptable error rates. Moreover, CHKS detects all odd number of bit errors, and 64-bit and less than 64-bit burst errors for which the factors are infinite.

Reliability Comparison of Schemes: Figure 8.6 provides significant insight regarding the strength of schemes in terms of reliability. i) We see that CRC is the most reliable scheme. With hardware acceleration being widely available as we stated in Section 8.2.5, it is the best option whenever possible. In addition, CRC's reduction factors increase as the number of error or the data size increases. ii) CHKS's reliability increases as the number of bits in error increases. Even though we did not plot more bits in error due to lack of space, this trend continues. iii) SNAP is more reliable than CHKS. However its reliability decreases when data size or the Chipkill's uncorrected rate increases. This makes CHKS more affordable when resources (memory, cores) are limited. iv) Task granularity, that is, sizes of task argument should be carefully considered since it both affects the reliability of schemes and the performance overheads. Coarse-grain tasks, as we saw in Cholesky and FFT, can prevent better overlapping of computation and fault-tolerance, e.g. CRC calculation, snapshot taking, thereby degrading performance. Automatic tools, such as [16, 116], can be consulted to obtain the optimal task granularity for a task-parallel program.

Takeaway 3: *All schemes reduce Chipkill's undetected error rates significantly and sufficiently, i.e. more than 70×.*

Memory Vulnerability Analysis

In this section, we perform experiments to see how much our framework reduces the memory vulnerability since it does not protect all application memory. Table 8.3 (2nd and 5th columns) shows the reduced percentages of Architectural Vulnerability Factors (AVFs) [117]

Table 8.3: Reduced % AVF and % Final Wait Times

Bench.	Reduced % AVF	% Wait Times	Bench.	Reduced % AVF	% Wait Times
Sparse LU	96%	21%	Cholesky	99%	60%
Perlin	91%	25%	Matmul	76%	48%
CG	99%	26%	Nbody	79%	11%
Knapsack	87%	13%	Linpack	81%	43%
FFT	89%	16%	Pingpong	77%	9%

for all benchmarks. AVFs [117] refers to the probability that a fault will result in a visible error in a process structure. Our scheme reduces this probability and here we provide how much it reduces it. The reduced percentages are measured as follows: We measure the quantities $prottime(TA_i) \times vol(TA_i)$ and $unprottime(TA_i) \times vol(TA_i)$ for each task argument TA_i when it is protected and when it is not protected respectively during program executions. $prottime(TA_i)$ and $unprottime(TA_i)$ refers the time duration that the task argument TA_i is protected and not protected respectively. $vol(TA_i)$ refers to the size of memory of the task argument TA_i. Then we calculate the reduced % of AVF as follows:

$$\frac{(\sum_{i=1}^{n} prottime(TA_i) \times vol(TA_i)) \times 100}{\sum_{i=1}^{n} (prottime(TA_i) + unprottime(TA_i)) \times vol(TA_i)} \tag{8.4}$$

where n is the total number of task arguments of a benchmark for an execution of it. As seen from the table, the reductions in memory vulnerability are high for all benchmarks and 87% on average. There are two main reasons for this: First, the input-only task arguments or data are much larger than the output arguments. In fact, in task-parallel programs and also in our benchmarks there is usually one output argument and several input-only arguments for each task. Second, most importantly, while the execution of a task-based computation continues, the application stops accessing some task arguments. Towards the end, the number of these arguments increases. Our scheme keeps the relevant snapshots and calculated CRCs. Therefore, just before finishing the execution, the runtime makes a final check for the arguments that are part of the output of the program and detects and recovers from any error since the last time they were used. This way we not only utilize the already existing redundancy but also increase the error coverage significantly. We perform experiments to gauge the effect of this. Table 8.3 (3rd and 6th columns) shows the average percentages of the wait times of task arguments

since the last time they were used with respect to the execution times of benchmarks. As we see, the relative percentages of these wait times are high. Additionally, the overheads for this final check are small enough to be insignificant. For the temporary memory utilized within task computations, since its vulnerability window is very short, its vulnerability is negligible.

Moreover we conduct experiments to measure the interval from the time when an output is produced to the time when its snapshot is taken and its CRC is calculated. During this interval memory is not protected. Results show this interval is very short for all benchmarks and ranges from 0.4 (5.3×10^{-7}%, CG) to 3464 (2×10^{-3}%, FFT) microseconds. This leaves very small probability of undetected errors.

Takeaway 4: *Our framework reduces the memory vulnerability of the benchmarks significantly with 87% on average.*

Takeaway 5: *In task-parallel programs, more memory is vulnerable when being idle or read-only as task arguments than when being modified by task computations. This is evident from our measurements of the product of space and time when memory is protected (87%) versus when memory is not protected (13%).*

To maximize the error coverage of our framework, an outer task can be defined with proper input arguments such that this outer task encompasses the entire body of the application program. This way, with minimal overhead the error coverage is increased significantly where most of the program execution progresses within task computations. This outer task can further improve the reduction in memory vulnerability (which is already 87% on average) enabling our framework to cope with exascale error rates.

8.3.4 Comparison and Usage of Schemes

In terms of performance overheads, software CRC and CHKS have higher overheads than SNAP. However with hardware acceleration CRC calculation incurs the lowest overhead. In terms of memory, SNAP is the most expensive one while CRC and CHKS have similar memory overhead. In terms of reliability, CRC is the most reliable scheme followed by SNAP and CHKS. In general, hardware-based CRC is the best option and other schemes should only be considered if it is not available. SNAP can be used when memory usage is not a limitation. CHKS can be used when software CRC is not affordable and memory usage is a limitation. Fault-detection-only variants can be used for the cases where error recovery is already available, such as Algorithm Based Fault-tolerance (ABFT) [118]. ABFT provides error recovery at

algorithm level and usually assumes error detection. Finally, detection-only variants can also be used for detecting Silent Data Corruptions (SDC) [2], which corrupt the outputs of HPC applications without being detected. Table 8.4 summarizes this discussion.

Generally we propose our CRC scheme to be used on top of existing hardware ECCs, such as Chipkill, where hardware ECCs protect system memory and our scheme protects the critical and minimal application data. However, our scheme can still be used for the systems without any hardware ECCs, similar to the work of Maruyama et al. [87] and for systems like the Mont-blanc prototype [119].

Table 8.4: Comparison and Usage of the Mechanisms

Mechanism	Memory Cost	Computa. Cost	Reliability	Scenario
CRC	Medium	High	High	Computation affordable Handling aging transparently Augmenting hardware ECCs
CRC-Hardware	Medium	Low	High	Best option when available Same scenarios w/ software CRC
SNAP	High	Low	Medium	Memory affordable
CHKS	Medium	Medium	Medium	Trade-off between SNAP and CRC
Detect. only CRC	Low	High	High	ABFT Silent Data corruption
Detect. only SNAP	Medium	Low	Low	ABFT Silent Data corruption
Detect. only CHKS	Low	Low	Medium	ABFT Silent Data corruption

8.4 Conclusions

In this chapter, we present our CRC-based mechanism to provide error detection and recovery for memory errors in task-parallel applications. Our scheme can cooperate with the underlying hardware ECCs e.g. Chipkill where Chipkill protects all system memory and CRC provides an additional and stronger layer of protection for critical application data for the mitigation of the

projected exascale memory error rates. Our design space analysis shows that CRC scheme is the best tradeoff. In addition, we show that CRC scheme is low-overhead and highly scalable with high core counts. Moreover, we show that hardware acceleration significantly decreases CRC computation. Our study of the reliability of our CRC mechanism demonstrates that it reduces Chipkill's uncorrected error rate significantly and sufficiently for the Exascale rates. Finally, we find that CRC scheme reduces memory vulnerability in task-parallel dataflow programs significantly.

9 Conclusion

As we get closer to the Exascale era, reliability becomes one of the main concerns for future high performance computing (HPC) and Exascale systems. It is widely believed that hardware-based fault tolerance techniques alone will not be sufficient to provide the required level of reliability for such systems. Therefore, complementary techniques are sought in software to strengthen the systems' fault tolerance.

Currently task-parallel dataflow programming models are becoming widely used to implement HPC applications for achieving higher performance. Moreover, it has been shown that dataflow execution model improves performance of HPC applications with asynchronous execution. As a result, we find that it is important to provide software-based fault-tolerance for task-parallel dataflow HPC applications.

In this thesis, we propose software-based mechanisms to mitigate errors in task-parallel dataflow HPC application computations. Our mechanisms are designed to protect both computation and memory. In addition, we develop mathematical performance models to assess the performance of our techniques when unified with system-wide checkpointing or modelling task computations. The following provides a summary for each chapter.

In Chapter 4, we introduced NanoCheckpoints which was a low-overhead and scalable checkpoint/restart scheme for task-parallel dataflow HPC applications. We developed two variants and showed the singleton variant significantly reduced the memory and performance overheads. Experiments also showed that our framework is scalable when error occurs frequently.

In Chapter 5, we developed a mathematical model that unifies NanoCheckpoints with system-wide checkpointing. The experiments showed the benefits of this unification, especially when the error rates are high as expected for the Exascale era.

In Chapter 6, we presented our novel message logging protocol for hybrid distributed task-parallel programs on top of NanoCheckpoints. We showed that our protocol is low-overhead and scalable. In addition, we proposed a performance model to optimize the checkpoint interval of system-wide checkpointing when our protocol is integrated with system-wide checkpointing. Experiments showed significant performance gain from this integration.

In Chapter 7, we discussed our selective replication mechanisms and reliability model. We evaluated our mechanisms and showed their effectiveness as well as their overheads which were found to be low. We highlighted several important conclusions from this study.

In Chapter 8, we proposed a runtime-based memory reliability scheme for task-parallel data-driven programs. Our scheme was designed based on cyclic redundancy checks. Experiments showed that our framework is low-overhead and highly scalable. Mathematical analysis and experimental study showed the effectiveness of our framework. We concluded that runtime-based memory reliability is feasible in programming models such as dataflow task-based ones.

With the techniques in Chapter 4 and Chapter 6 we addressed fail-stop errors in task-parallel dataflow applications' computations. With the techniques in Chapter 7 we addressed SDCs or (silent errors) in these applications' computations. As a result, we provide a high failure coverage for these computations against different types of errors. Finally with techniques in Chapter 8 we protect memory of task-parallel dataflow applications. Thus, in this thesis, we provide both memory and computation protection for our target applications achieving a high failure coverage against main types of errors.

9.1 Future Work

As future work, we might work on supporting fault-tolerance for heterogeneous architectures since most programming models support executions on such architectures. Second, we might

be working on providing reliability for cluster computations in programming models such as OmpSs.

10

Publication List

The content of this thesis led to following publications:

Journals

- **Omer Subasi**, Tatiana V. Martsinkevich, Ferad Zyulkyarov, Osman Unsal, Jesus Labarta and Franck Cappello, Unified Fault-tolerance Framework for Hybrid Task-parallel Message-passing, International Journal of High Performance Computing Applications (IJHPAC), 2016

Conferences

- **Omer Subasi**, Gulay Yalcin, Jesus Labarta, Osman Unsal and Adrian Cristal, CRC-based Memory Reliability for Task-parallel HPC Applications, 30th IEEE International Parallel & Distributed Processing Symposium, May 2016
- **Omer Subasi**, Ferad Zyulkyarov, Osman Unsal and Jesus Labarta, Marriage Between Coordinated and Uncoordinated Checkpointing for the Exascale Era, 17th IEEE International Conference on High Performance Computing and Communications, August 2015

- Tatiana V. Martsinkevich, **Omer Subasi**, Osman S. Unsal, Franck Cappello and Jesus Labarta, Fault-Tolerant Protocol for Hybrid Task-Parallel Message-Passing Applications, 2015 IEEE International Conference on Cluster Computing, CLUSTER, 2015

- **Omer Subasi**, Javier Arias, Jesus Labarta, Osman Unsal and Adrian Cristal, Programmer-directed Partial Redundancy for Resilient HPC, ACM International Conference on Computing Frontiers, May 2015

- **Omer Subasi**, Javier Arias, Jesus Labarta, Osman Unsal and Adrian Cristal, NanoCheckpoints: A Task-based Asynchronous Dataflow Framework for Efficient and Scalable Checkpoint/Restart, The 23rd Euromicro International Conference on Parallel, Distributed and Network-based Processing, March 2015

Workshops

- **Omer Subasi**, Gulay Yalcin, Ferad Zyulkyarov, Osman Unsal and Jesus Labarta, A runtime heuristic to selectively replicate tasks for application-specific reliability targets, 2nd International Workshop on Fault Tolerant Systems (FTS), September 2016

- **Omer Subasi**, Javier Arias, Jesus Labarta, Osman Unsal and Adrian Cristal, Leveraging a Task-based Asynchronous Dataflow Substrate for Efficient and Scalable Resiliency, Joint Euro-TM/MEDIAN Workshop on Dependable Multicore and Transactional Memory Systems (DMTM), In conjunction with Hipeac 2014 Conference, Jan 2014

The following studies are not included in the thesis

- **Omer Subasi**, Sheng Di, Leonardo Bautista-Gomez, Prasanna Balaprakash, Osman Unsal, Jesus Labarta, Adrian Cristal and Franck Cappello, Spatial Support Vector Regression to Detect Silent Errors in the Exascale Era, 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, May 2016

Bibliography

[1] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward exascale resilience. *Int. J. High Perform. Comput. Appl.*, 23(4):374–388, November 2009. ISSN 1094-3420. doi: 10.1177/1094342009347767. URL http://dx.doi.org/10.1177/1094342009347767. 1, 2

[2] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron 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*, SC '12, pages 78:1–78:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. URL http://dl.acm.org/citation.cfm?id=2388996.2389102. 1, 6, 19, 20, 79, 102, 145

[3] Jack Dongarra, Pete Beckman, and Terry Moore et al. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, 2011. ISSN 1094-3420. doi: 10.1177/1094342010391989. URL http://dx.doi.org/10.1177/1094342010391989. 1, 49, 50, 74

[4] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, 2009. doi: 10.1109/TPDS.2008.105. URL http://dx.doi.org/10.1109/TPDS.2008.105. 1, 6

[5] Priyanka Ghosh, Yonghong Yan, Deepak Eachempati, and Barbara Chapman. A prototype implementation of openmp task dependency support. In Alistair P. Rendell,

Barbara M. Chapman, and Matthias 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. ISBN 978-3-642-40698-0. doi: 10.1007/978-3-642-40698-0_10. URL http://dx.doi.org/10.1007/978-3-642-40698-0_10. 1, 19

[6] Intel tbb 4.3 update 2. http://www.threadingbuildingblocks.org/documentation. 1, 6

[7] Abdelhalim Amer, Naoya Maruyama, Miquel Pericàs, Kenjiro Taura, Rio Yokota, and Satoshi 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. 1, 6, 41

[8] Vilas Sridharan and Dean Liberty. A study of dram failures in the field. In *SC 2012*, pages 76:1–76:11. ISBN 978-1-4673-0804-5. URL http://dl.acm.org/citation.cfm?id=2388996.2389100. 6

[9] Mojtaba Ebrahimi, Hossein Asadi, and Mehdi 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. ISBN 978-1-4503-2071-9. doi: 10.1145/2463209.2488858. URL http://doi.acm.org/10.1145/2463209.2488858. 6

[10] Christopher Weaver, Joel Emer, Shubhendu S. Mukherjee, and Steven K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 264–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2143-6. URL http://dl.acm.org/citation.cfm?id=998680.1006723. 6

[11] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A DeBardeleben, Pedro C Diniz, Christian Engelmann, Mattan

Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, 28(2), 2014. doi: 10.1177/1094342014522573. 6

[12] Bernd Panzer-Steindel. Data integrity. In *CERN/IT Draft 1.3*, 2007. 6

[13] Dimitrios Chasapis, Marc Casas, Miquel Moretó, Raul Vidal, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Parsecss: Evaluating the impact of task parallelism in the PARSEC benchmark suite. *TACO*, 12(4):41, 2016. doi: 10.1145/2829952. URL http://doi.acm.org/10.1145/2829952. 6

[14] Melvin 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. doi: 10.1145/1463822.1463838. URL http://doi.acm.org/10.1145/1463822.1463838. 6

[15] John L. Kelly, Carol Lochbaum, and V.A. Vyssotsky. A block diagram compiler. *The Bell System Technical Journal*, 1961. 6

[16] Vladimir Subotic, Steffen Brinkmann, Vladimir Marjanovic, Rosa M. Badia, Jose Gracia, Christoph Niethammer, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Programmability and portability for exascale: Top down programming methodology and tools with StarSs. *J. Comput. Science*, 4(6):450–456, 2013. 6, 142

[17] 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. ISSN 1070-9924. doi: 10.1109/99.660313. 6

[18] Alejandro Duran, Eduard Ayguade, Rosa M. Badia, Jesus Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173–193, 2011. 6, 10, 110

[19] OpenMP Architecture Review Board. OpenMP application programming interface version 4.0, 2013. 6

[20] Sheng Di, Eduardo Berrocal, Leonardo Bautista-Gomez, Katherine Heisey, Rinku Gupta, and Franck Cappello. Towards effective detection of silent data errors for hpc applications (poster). 2014. 8, 20

[21] Xavier Teruel, Xavier Martorell, Alejandro Duran, Roger Ferrer, and Eduard Ayguadé. Support for openmp tasks in nanos v4. In *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, CASCON '07, pages 256–259, Riverton, NJ, USA, 2007. IBM Corp. doi: 10.1145/1321211.1321241. URL `http://dx.doi.org/10.1145/1321211.1321241`. 8, 110, 131

[22] Mercurium overview: http://pm.bsc.es/mcxx. 8

[23] M. Andersch, Chi Ching Chi, and B. Juurlink. Using OpenMP superscalar for parallelization of embedded and consumer applications. In *2012 International Conference on Embedded Computer Systems (SAMOS)*, pages 23–32, July 2012. doi: 10.1109/SAMOS.2012.6404154. 10, 24, 82

[24] Vladimir Gajinov, Srcan Stipic, Igor Eric, Osman S. Unsal, Eduard Ayguade, and Adrian Cristal. Dash: A benchmark suite for hybrid dataflow and shared memory programming models: with comparative evaluation of three hybrid dataflow models. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, 2014. 10

[25] Michael Andersch, Ben Juurlink, and Chi Ching Chi. A benchmark suite for evaluating parallel programming models. In *Workshop on Parallel Systems and Algorithms (PARS)*, pages 7–17, 2011. 10, 24, 82

[26] Vladimir Marjanovic, Jesus Labarta, Eduard Ayguade, and Mateo 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. 10, 25

[27] S. Borkar. The exascale challenge. In *2010 International Symposium on VLSI Design Automation and Test (VLSI-DAT)*, pages 2–3, 2010. 11

[28] R. Hamming. Error detecting and error correcting codes. 1950. 11

[29] T. Dell. A white paper on the benefits of chipkill-correct ecc for pc server main memory, ibm microelectronics division, technical report. 1997. 11, 123

[30] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–310, 2015. 11, 124, 139

[31] R.C. Bose and D.K. Ray-Chaudhuri. On a class of error correcting binary group codes. *Information and Control*, 1960. 11

[32] K. Bergman et al. Exascale computing study: Technology challenges in achieving exascale systems. 2008. 11, 123

[33] D. Strukov. The area and latency tradeoffs of binary bit-parallel BCH decoders for prospective nanoelectronic memories. In *Fortieth Asilomar Conference on Signals, Systems and Computers*, pages 1183–1187, 2006. 11

[34] W.W. Peterson and D.T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 1961. 13, 124

[35] G. Castagnoli, S. Brauer, and M. Herrmann. Optimization of cyclic redundancy-check codes with 24 and 32 parity bits. volume 41 of *IEEE Transactions on Communications*, pages 883–892, 1993. 13, 129

[36] P. Koopman. 32-bit cyclic redundancy codes for internet applications. In *International Conference on Dependable Systems and Networks.*, pages 459–468, 2002. 13, 129

[37] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. volume 34, pages 375–408, New York, NY, USA, September 2002. ACM. doi: 10.1145/568522.568525. URL http://doi.acm.org/10.1145/568522.568525. 15, 16

[38] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: High performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 32:1–32:32, New York, NY, USA, 2011. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063427. URL

http://doi.acm.org/10.1145/2063384.2063427. 16, 17, 19, 39, 48, 76

[39] Kento Sato, Naoya Maruyama, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R. de Supinski, and Satoshi Matsuoka. Design and modeling of a non-blocking checkpointing system. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 19:1–19:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. URL http://dl.acm.org/citation.cfm?id=2388996.2389022. 16, 17, 39

[40] Rolf Riesen, Kurt Ferreira, Dilma Da Silva, Pierre Lemarinier, Dorian Arnold, and Patrick G. Bridges. Alleviating scalability issues of checkpointing protocols. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012. ISBN 978-1-4673-0804-5. 16

[41] Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack J. Dongarra. Correlated set coordination in fault tolerant message logging protocols. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, 2011. 16

[42] Esteban Meneses, Xiang Ni, and L. V. Kale. A Message-Logging Protocol for Multicore Systems. In *Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, Boston, MA, June 2012. 16

[43] Thomas Ropars, Tatiana V. Martsinkevich, Amina Guermouche, André Schiper, and Franck Cappello. Spbc: Leveraging the characteristics of mpi hpc applications for scalable checkpointing. In *SC 2013*, pages 8:1–8:12. ISBN 978-1-4503-2378-9. 16, 59, 60

[44] Paul H. Hargrove and Jason C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In Proceedings of SciDAC 2006, June 2006. 16

[45] Victor C. Zandy and Barton P. Miller. chkpt: set of libraries and programs for user-level process checkpointing. Available in http://pages.cs.wisc.edu/~zandy/ckpt/. 16

[46] Wenjing Ma and Sriram Krishnamoorthy. Data-driven fault tolerance for work stealing computations. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 79–90, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1316-2. doi: 10.1145/2304576.2304589. URL http://doi.acm.org/10.1145/2304576.2304589. 17

[47] Jinsuk Chung, Ikhwan Lee, Michael Sullivan, Jee Ho Ryoo, Dong Wan Kim, Doe Hyun Yoon, Larry Kaplan, and Mattan Erez. Containment domains: a scalable, efficient, and flexible resilience scheme for exascale systems. In *SC 2012*, pages 58:1–58:11. ISBN 978-1-4673-0804-5. URL http://dl.acm.org/citation.cfm?id=2388996.2389075. 17

[48] Joseph Sloan, Rakesh Kumar, and Greg Bronevetsky. Algorithmic approaches to low overhead fault detection for sparse linear algebra. In *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4673-1624-8. URL http://dl.acm.org/citation.cfm?id=2354410.2355166. 17, 20

[49] Greg Bronevetsky, Keshav Pingali, and Paul Stodghill. Experimental evaluation of application-level checkpointing for openmp programs. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 2–13, New York, NY, USA, 2006. ACM. ISBN 1-59593-282-8. doi: 10.1145/1183401.1183405. URL http://doi.acm.org/10.1145/1183401.1183405. 17

[50] Hongyi Fu and Yan Ding. Using redundant threads for fault tolerance of openmp programs. In *ICISA 2010*, pages 1–8, 2010. doi: 10.1109/ICISA.2010.5480321. 17

[51] Oussama Tahan and Mohamed Shawky. Using dynamic task level redundancy for openmp fault tolerance. In *Proceedings of the 25th international conference on Architecture of Computing Systems*, ARCS'12, pages 25–36, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28292-8. doi: 10.1007/978-3-642-28293-5_3. URL http://dx.doi.org/10.1007/978-3-642-28293-5_3. 17

[52] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, September 1974. ISSN 0001-0782. doi: 10.1145/361147.

361115. URL http://doi.acm.org/10.1145/361147.361115. 18, 46, 67

[53] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, February 2006. ISSN 0167-739X. doi: 10.1016/j.future.2004.11.016. URL http://dx.doi.org/10.1016/j.future.2004.11.016. 18

[54] H. Jin, Y. Chen, H. Zhu, and X. H. Sun. Optimizing hpc fault-tolerant environment: An analytical approach. In *2010 39th International Conference on Parallel Processing*, pages 525–534, Sept 2010. doi: 10.1109/ICPP.2010.80. 18

[55] L. Wang, Karthik Pattabiraman, Z. Kalbarczyk, R. K. Iyer, L. Votta, C. Vick, and A. Wood. Modeling coordinated checkpointing for large-scale supercomputers. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 812–821, June 2005. doi: 10.1109/DSN.2005.67. 18

[56] Z. Zheng and Z. Lan. Reliability-aware scalability models for high performance computing. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–9, Aug 2009. doi: 10.1109/CLUSTR.2009.5289177. 18

[57] S. Di, M. S. Bouguerra, L. Bautista-Gomez, and F. Cappello. Optimization of multi-level checkpoint model for large scale hpc applications. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1181–1190, May 2014. doi: 10.1109/IPDPS.2014.122. 18, 47

[58] Sheng Di, Leonardo Bautista-Gomez, and Franck Cappello. Optimization of a multilevel checkpoint model with uncertain execution scales. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 907–918, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-5500-8. doi: 10.1109/SC.2014.79. URL http://dx.doi.org/10.1109/SC.2014.79. 18

[59] L. A. B. Gomez, P. Balaprakash, M. S. Bouguerra, S. M. Wild, F. Cappello, and P. D. Hovland. Poster: Energy-performance tradeoffs in multilevel checkpoint strategies. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 278–279, Sept 2014. doi: 10.1109/CLUSTER.2014.6968749. 18

[60] George Bosilca, Aurélien Bouteiller, Elisabeth Brunet, Franck Cappello, Jack Dongarra, Amina Guermouche, Thomas Herault, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. Unified model for assessing checkpointing protocols at extreme-scale. *Concurrency and Computation: Practice and Experience*, 26(17):2772–2791, 2014. ISSN 1532-0634. doi: 10.1002/cpe.3173. URL http://dx.doi.org/10.1002/cpe.3173. 18, 47

[61] Omer Subasi, Ferad Zyulkyarov, Osman S. Unsal, and Jesús Labarta. Marriage between coordinated and uncoordinated checkpointing for the exascale era. In *17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015, 7th IEEE International Symposium on Cyberspace Safety and Security, CSS 2015, and 12th IEEE International Conference on Embedded Software and Systems, ICESS 2015, New York, NY, USA, August 24-26, 2015*, pages 470–478, 2015. doi: 10.1109/HPCC-CSS-ICESS.2015.150. URL http://dx.doi.org/10.1109/HPCC-CSS-ICESS.2015.150. 18

[62] Daniel P. Siewiorek and Robert S. Swarz. *Reliable Computer Systems (3rd Ed.): Design and Evaluation*. 1998. 18

[63] Bradford M. Beckmann, Michael R. Marty, and David 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. ISBN 0-7695-2732-9. doi: 10.1109/MICRO.2006.10. URL http://dx.doi.org/10.1109/MICRO.2006.10. 18, 19

[64] Franz Josef Grüneberger, Thomas Heinze, and Pascal 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. 19

[65] Yixin Luo, Sriram Govindan, Bikash Sharma, Mark Santaniello, Justin Meza, Aman Kansal, Jie Liu, Badriddine Khessib, Kushagra Vaid, and Onur 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. ISBN 978-1-4799-2233-8. doi: 10.1109/

DSN.2014.50. URL http://dx.doi.org/10.1109/DSN.2014.50. 19, 80, 85

[66] M. Ripeanu S. Gurumurthi B. Fang, K. Pattabiraman. Evaluating the error resilience of parallel programs. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2014)*, pages 720–725, 2014. 19, 99

[67] George Tzenakis, Angelos Papatriantafyllou, John Kesapides, Polyvios Pratikakis, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. Bddt:: Block-level dynamic dependence analysisfor 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. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145864. URL http://doi.acm.org/10.1145/2145816.2145864. 19

[68] Kurt B. Ferreira, Jon Stearley, James H. Laros III, Ron Oldfield, Kevin T. Pedretti, Ron Brightwell, Rolf Riesen, Patrick G. Bridges, and Dorian C. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*, pages 44:1–44:12, 2011. doi: 10.1145/2063384.2063443. URL http://doi.acm.org/10.1145/2063384.2063443. 19

[69] Thanadech Thanakornworakij, Raja Nassar, Chokchai Box Leangsuksun, and Mihaela Paun. Reliability model of a system of k nodes with simultaneous failures for high-performance computing applications. *Int. J. High Perform. Comput. Appl.*, 27(4):474–482, November 2013. ISSN 1094-3420. doi: 10.1177/1094342012464506. URL http://dx.doi.org/10.1177/1094342012464506. 19

[70] S. R. Welke, B. W. Johnson, and J. H. Aylor. Reliability modeling of hardware/software systems. *IEEE Transactions on Reliability*, 44(3):413–418, Sep 1995. ISSN 0018-9529. doi: 10.1109/24.406575. 19

[71] James Elliott, Kishor Kharbas, David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Combining partial redundancy and checkpointing for hpc. In *Proceedings of the 2012 IEEE 32nd International Conference on Distributed Computing Systems*, ICDCS '12, pages 615–626, Washington, DC, USA, 2012. IEEE Computer Society.

ISBN 978-0-7695-4685-8. doi: 10.1109/ICDCS.2012.56. URL http://dx.doi.org/10.1109/ICDCS.2012.56. 19, 20

[72] M. Turmon, R. Granat, D.S. Katz, and J.Z. Lou. Tests and tolerances for high-performance software-implemehted fault detection. *IEEE Transactions on Computers*, 52(5):579–591, May 2003. ISSN 0018-9340. doi: 10.1109/TC.2003.1197125. 20

[73] E. Ciocca, I. Koren, Z. Koren, C. M. Krishna, and D. S. Katz. Application-level fault tolerance in the orbital thermal imaging spectrometer. In *Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'04)*, PRDC '04, pages 43–48, Washington, DC, USA, 2004. ISBN 0-7695-2076-6. URL http://dl.acm.org/citation.cfm?id=977407.978747. 20

[74] Luc Jaulmes, Marc Casas, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Exploiting asynchrony from exact forward recovery for DUE in iterative solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, pages 53:1–53:12, 2015. 20

[75] Marc Casas, Bronis R. de Supinski, Greg Bronevetsky, and Martin Schulz. Fault resilience of the algebraic multi-grid solver. In *International Conference on Supercomputing, ICS'12, Venice, Italy, June 25-29, 2012*, pages 91–100, 2012. doi: 10.1145/2304576.2304590. URL http://doi.acm.org/10.1145/2304576.2304590. 20

[76] Greg Bronevetsky and Bronis R. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22nd Annual International Conference on Supercomputing, ICS 2008, Island of Kos, Greece, June 7-12, 2008*, pages 155–164, 2008. doi: 10.1145/1375527.1375552. URL http://doi.acm.org/10.1145/1375527.1375552. 20

[77] Omer Subasi, Javier Arias, Osman Unsal, Jesus Labarta, and Adrian Cristal. Programmer-directed partial redundancy for resilient hpc. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 47:1–47:2, New York, NY, USA, 2015. ISBN 978-1-4503-3358-0. doi: 10.1145/2742854.2742903. URL http://doi.acm.org/10.1145/2742854.2742903. 20

[78] Eduardo Berrocal, Leonardo Bautista-Gomez, Sheng Di, Zhiling Lan, and Franck Cappello. Lightweight silent data corruption detection based on runtime data analysis

for hpc applications. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 275–278, New York, NY, USA, 2015. ISBN 978-1-4503-3550-8. doi: 10.1145/2749246.2749253. URL http://doi.acm.org/10.1145/2749246.2749253. 20

[79] Sheng Di, Eduardo Berrocal, and Franck Cappello. An efficient silent data corruption detection method with error-feedback control and even sampling for HPC applications. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen, China, May 4-7, 2015*, pages 271–280, 2015. doi: 10.1109/CCGrid.2015.17. 20

[80] Vishal Sharma, Ganesh Gopalakrishnan, and Greg Bronevetsky. Detecting soft errors in stencil based computations. In *The 11th Workshop on Silicon Errors in Logic - System Effects*, 2015. 20

[81] Omer Subasi, Sheng Di, Leonardo Bautista-Gomez, Prasanna Balaprakash, Osman S. Ünsal, Jesús Labarta, Adrián Cristal, and Franck 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. doi: 10.1109/CCGrid.2016.33. URL http://dx.doi.org/10.1109/CCGrid.2016.33. 20

[82] Sheng Li, Ke Chen, Ming-Yu Hsieh, N. Muralimanohar, C.D. Kersey, J.B. Brockman, A.F. Rodrigues, and N.P. Jouppi. System implications of memory reliability in exascale computing. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, 2011. 21, 123, 124

[83] Jungrae Kim, Michael Sullivan, and Mattan Erez. Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 101–112, 2015. 21

[84] Doe Hyun Yoon, Min Kyu Jeong, and Mattan Erez. Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, pages 295–306, 2011. 21

[85] Scott Levy, Patrick G. Bridges, Kurt B. Ferreira, Aidan P. Thompson, and Christian Trott. Evaluating the feasibility of using memory content similarity to improve system

resilience. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, pages 7:1–7:8, 2013. 21

[86] Omer Subasi, Osman S. Ünsal, Jesús Labarta, Gulay Yalcin, and Adrián Cristal. Crc-based memory reliability for task-parallel HPC applications. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pages 1101–1112, 2016. doi: 10.1109/IPDPS.2016.70. URL http://dx.doi.org/10.1109/IPDPS.2016.70. 21

[87] N. Maruyama, A. Nukada, and S. Matsuoka. A high-performance fault-tolerant software framework for memory on commodity GPUs. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, 2010. 21, 145

[88] David Fiala, KurtB. Ferreira, Frank Mueller, and Christian Engelmann. A tunable, software-based DRAM error detection and correction library for HPC. In *Euro-Par 2011: Parallel Processing Workshops*, volume 7156, pages 251–261, 2012. 21

[89] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Generative software-based memory error detection and correction for operating system data structures. In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12, 2013. 22

[90] Marenostrum iii system architecture: http://www.bsc.es/marenostrum-support-services/mn3. 27, 52, 69, 89, 110, 131

[91] Ken Perlin. Improving noise. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 681–682, New York, NY, USA, 2002. ACM. ISBN 1-58113-521-1. doi: 10.1145/566570.566636. URL http://doi.acm.org/10.1145/566570.566636. 27

[92] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *TCCA Newsletter*, 1995. 27, 119

[93] A. Petitet, R. C. Whaley, Jack Dongarra, and A. Cleary. Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. http://www.netlib.org/benchmark/hpl/. 28

[94] S. Ajit A. Verma and D. Karanki. *Reliability and Safety Engineering*. Springer, 2010. 43

[95] M. Ripeanu S. Gurumurthi B. Fang, K. Pattabiraman. Evaluating the error resilience of parallel programs. pages 720–725, 2014. 47

[96] 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. doi: 10.1109/PDP.2015.17. 48

[97] Saman Amarasinghe et al. Exascale software study: Software challenges in extreme scale systems, 2009. 49, 50, 74

[98] BSC Application Repository. https://pm.bsc.es/projects/bar/wiki/applications. 52, 110, 131

[99] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978. 59

[100] MPI: A message passing interface standard, version 3.0. 2012. URL `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`. 59

[101] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. On the use and implementation of message logging. In *24th International Symposium on Fault-Tolerant Computing*, pages 298–307, 1994. 63

[102] Tatiana V. Martsinkevich, Omer Subasi, Osman S. Unsal, Franck Cappello, and Jesús 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. doi: 10.1109/CLUSTER.2015.104. URL `http://dx.doi.org/10.1109/CLUSTER.2015.104`. 78

[103] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., 1990. 98

[104] A.K. Verma, S. Ajit, and D.R. Karanki. *Reliability and Safety Engineering*. Springer, 2010. 102, 107

[105] 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. ISSN 1530-4388. doi: 10.1109/TDMR.2012.2192736. 108

[106] Kevin M. Lepak and Mikko 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. ISBN 1-58113-196-8. doi: 10.1145/360128.360133. URL http://doi.acm.org/10.1145/360128.360133. 109

[107] John Shalf, Sudip Dosanjh, and John 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. ISBN 978-3-642-19327-9. URL http://dl.acm.org/citation.cfm?id=1964238.1964240. 115

[108] M. Maniatakos, M.K. Michael, and Y. Makris. Investigating the limits of AVF analysis in the presence of multiple bit errors. In *IEEE 19th International On-Line Testing Symposium (IOLTS)*, pages 49–54, 2013. 123

[109] Software for CRC-based memory relability. https://pm.bsc.es/sites/default/files/ftp/nanox/ad-hoc/nanox-mem_reliability-0.9a-2016-02-06.tar.gz. 126

[110] V. Gopal et al. Fast CRC computation for iscsi polynomial using CRC32 instruction. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/crc-iscsi-polynomial-crc32-instruction-paper.pdf, 2011. 130

[111] B. Sinharoy, R. Swanberg, N. Nayar, B. Mealey, J. Stuecheli, B. Schiefer, J. Leenstra, J. Jann, P. Oehler, D. Levitan, S. Eisen, D. Sanner, T. Pflueger, C. Lichtenau, W.E. Hall, and T. Block. Advanced features in IBM POWER8 systems. *IBM Journal of Research and Development*, 59(1):1:1–1:18, 2015. 130

[112] ARM. ARM CRC-32 instructions. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0802a/a64_general_alpha.html, Accessed in January 2016. 130

[113] AMD. Bios and kernel developer's guide (bkdg)for amd family 16h models 30h-3fh processors. http://support.amd.com/TechDocs/52740_16h_Models_30h-3Fh_BKDG.pdf, Accessed in January 2016. 130

[114] Vilas Sridharan, Jon Stearley, Nathan DeBardeleben, Sean Blanchard, and Sudhanva Gurumurthi. Feng shui of supercomputer memory: Positional effects in DRAM and SRAM faults. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 22:1–22:11, 2013. 139

[115] P. Koopman. CRC hamming weight data. http://users.ece.cmu.edu/~koopman/crc/hw_data.html, Accessed in January 2016. 140

[116] Vladimir Subotic, Roger Ferrer, Jose Carlos Sancho, Jesús Labarta, and Mateo Valero. Quantifying the potential task-based dataflow parallelism in MPI applications. In *Proceedings of the 17th International Conference on Parallel Processing*, Euro-Par'11, pages 39–51. 2011. 142

[117] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 29–, 2003. 142, 143

[118] A. Emmanuel et al. Towards resilient parallel linear krylov solvers: recover-restart strategies. Number 00843992, Accessed in January 2016. 144

[119] Montblanc project, http://www.montblanc-project.eu/. 145