CRC-based Memory Reliability for Task-parallel HPC Applications

Omer Subasi,[12] Osman Unsal,[1] Jesus Labarta,[12] Gulay Yalcin,[3] Adrian Cristal,[124]
[1]Barcelona Supercomputing Center, Spain,
[2]Universitat Politecnica de Catalunya, Spain, [3]Abdullah Gul University, Turkey
[4]IIIA - Artificial Intelligence Research Institute CSIC - Spanish National Research Council, Spain
{omer.subasi, osman.unsal, jesus.labarta, adrian.cristal}@bsc.es, gulay.yalcin@agu.edu.tr

Abstract—**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 (ECC) 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 and 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 work, we propose a Cyclic Redundancy Checks (CRCs) based software mechanism for task-parallel HPC applications. Our mechanism incurs only 1.7% 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. Results show that our CRC-based mechanism reduces the memory vulnerability by 87% on average with up to 32-bit burst (consecutive) and 5-bit arbitrary error correction capability.**

I. Introduction

As we get closer to Exascale era, reliability is becoming one of the main concerns for high performance computing (HPC). 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 [28] and memory reliability will be even more crucial due to massive increase in memory capacity in future HPC and exascale systems [9]. Additionally, issues with future transistor scaling will decrease memory lifetimes and make main memory more vulnerable to multiple bit flips [29].

Current HPC memory systems are mostly protected by hardware Error Correcting Codes (ECCs), such as Chipkill [15] 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 [38], [28]. In particular, Sridharan et al. [38] 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 (Section II). 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 work, 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 task-based parallelism and data-driven execution are becoming widely used to implement HPC applications for achieving higher performance [5], HPC programming platforms such as OpenMP 4.0 [34] and Intel Threading Blocks (TBB) [36] have recently added support for task-based dataflow parallelism (Section II). This programming paradigm offers some properties which can be leveraged for memory reliability such as data awareness (Section II-B). However, we find that application memory is particularly vulnerable to faults in task-based dataflow applications (Section IV) 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 work 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 [35] 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.*[1] 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 III). 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

[1]At the time of this manuscript, DDR4 DRAM standard introduced CRCs; however they are utilized to protect the RAM interconnect rather than providing end-to-end memory reliability as we do.

1530-2075/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPS.2016.70

IEEE computer society

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 IV).

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 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 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 IV-A) and the memory vulnerability analysis (Section IV-C).

II. Background and Motivation

In Section II-A we overview the state-of-the-art for memory reliability and ECCs. In Section II-B, we discuss fault tolerance and memory characteristics of task-parallel programs.

A. HPC Memory Reliability

1) Background: With shrinking feature sizes, error rates in main memory are expected to increase [11]. 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) [21] and Chipkill [15] are the most widely used ECCs. Chipkill [15] is used to mitigate DRAM chip failures. Considering future HPC and Exascale systems, on one hand Sridharan et al. [38] report that Chipkill will not be sufficient to mitigate the projected Exascale error rates. On the other hand, stronger ECCs, such as BCH [12], in hardware will incur prohibitive overheads given the predicted fault rates [9]. For BCH codes, Strukov et al. [40] 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 proportional 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) Hardware-based Memory Reliability: There have been hardware-based studies [28], [24] and [47] that propose stronger ECCs for HPC systems. For instance, Li et al. [28] 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 [24] provides more flexibility in terms of data granularity and coding strength than Chipkill. As opposed to our design, these techniques require custom hardware to deploy the proposed solution. Nevertheless, our framework 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. [27] 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) Software-based Memory Reliability: In comparison to hardware, software-based memory error fault-tolerance has not been investigated thoroughly for HPC. Our work 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. [32] 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. [18] 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. 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. The work of Borchert et al. [10] proposes flexible software-based ECCs which are in essence designed for operating system data structures.

4) Cyclic Redundancy Checks: Cyclic Redundancy Checks [35] 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 [14] [25]. Therefore, we 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 (Section III). This way, as the multibit errors are becoming more frequent, it becomes inevitable to provide stronger detection mechanisms in an adaptive way so that the system managers and/or the application users have the flexibility to tailor fault detection capabilities according to their needs.

B. Task-parallel Dataflow Programming

Fault Tolerance Advantages: We first discuss some properties of task-parallel data-driven programming that help us to design and implement our efficient and low-overhead fault-tolerance techniques. First, since task-parallel data-driven programming achieves higher performance [5] than fork-join programming such as in Open 3.0, HPC programming platforms such as OpenMP 4.0 [34] and Intel Threading Blocks (TBB) [36] have recently added support for task-based data-driven parallelism. Another example is the OmpSs programming model [16] and its Nanos [46] runtime, a task-based dataflow platform that is developed specifically for materialising data-driven and asynchronous execution. Second, in task-parallel dataflow programs, the programmer specifies the task inputs and outputs. Thus, the exact data that needs to be protected, i.e. task inputs and outputs, is already known - which we term as *data-awareness*. This enables the runtime to be aware of the minimal amount of memory to be protected. Third, error detection and recovery process is very efficient since application tasks that do not have dependencies to a task whose arguments experienced a memory error can continue their computations, in parallel and asynchronously with the memory recovery process of the affected task.

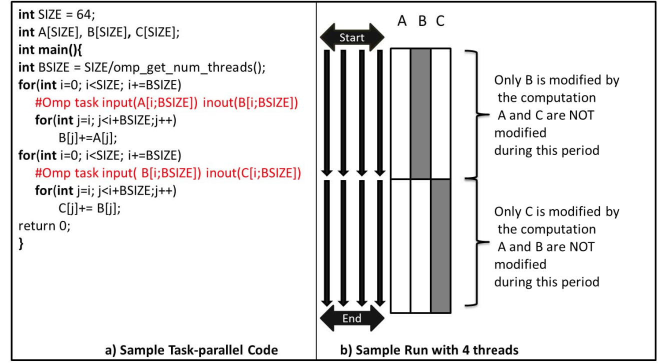

Fig. 1. Sample Task-parallel Program

Annotating tasks with inputs and outputs is mostly automatic with the recent advanced tools. Well-established tools such as Tareador and Temanejo [44], [45] help the programmer not only to correctly and completely specify the inputs and outputs but also to explore the potential of different parallelization strategies. Their framework automatically detects data dependencies among tasks in order to estimate the potential parallelism in HPC applications. In fact, a recent tool [43] automatizes the process of exploring parallelism, that is, the specification of task annotations.

Memory Characteristics: We now present the memory characteristics of task-parallel programs. We illustrate this with a simple example with two nested loops in Figure 1 with task annotations. In Figure 1 a), we see the sample program where in the two nested loops, parts of arrays A, B and C are processed in parallel and all three arrays are considered as part of the program output. In the first nested loop, array A is an input and array B is an input and output (called inout) and each element of array B is increased by the corresponding element of array A. In the second loop, array B is an input and array C is an output. Each element of array C increased by the corresponding element of B. Figure 1 b) shows the execution graph of the program assuming four threads. It also shows which array is modified during the computation phases by filling it grey. From the graph, we deduce that i) array A is not modified by any program computation at all ii) arrays B and C are modified by the program computation roughly 50% of the time. This means on average 66% of task arguments and 66% task data is not modified during the entire computation. Moreover after the first nested loop, array A stays idle and vulnerable in memory until the program finishes.

Moreover, assuming that the number of idle cores is less than four in the system, there will be ready tasks to be executed, however they will be waiting due to lack of idle cores. This would increase the memory vulnerability: more memory would be idle - therefore vulnerable - waiting for computation to commence on ready-tasks. Parallel runtimes usually produce more work than computation resources could execute attempting to exploit as much as parallelism they can. We will see in Section IV-A that these so called *look-ahead capabilities* of the runtimes, such as pre-fetching and executing independent tasks out-of-order, cause memory to be relatively more vulnerable to errors raising memory fault-

tolerance concerns. In contrast, in terms of performance concerns, look-ahead capabilities provide important advantages such as increasing throughput, deeper parallelism and faster computation.

III. Design and Implementation

We present our failure model in Section III-A. We then discuss the design and implementation of SNAP, CHKS, and CRC in Sections III-B, III-C and III-D respectively. In Section III-E we present the hardware acceleration of CRC. Our CRC implementation is open source and can be downloaded from [1].

A. Failure Model and Implementation Infrastructure

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 IV-A) and the memory vulnerability analysis (Section IV-C), 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. 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.

We implement our framework in publicly available OmpSs PM [16] and its Nanos runtime [46]. However, our technique is applicable for other task-parallel dataflow platforms. Nevertheless, the performance of OmpSs and Nanos is on par with the optimized commercial and open source implementation of OpenMP [6], [7]. The hybrid OmpSs+MPI model combines dataflow execution model with message passing achieving higher performance compared to MPI only by overlapping MPI communications with computation [30]. Our framework can be directly applied to hybrid MPI/OpenMPI PM given that MPI/OpenMPI programs are taskified, and inputs and outputs are inferred. Alternatively, MPI/OpenMPI programs can be ported to OmpSs+MPI PM. There is already research on fault recovery for task computations in pure OmpSs [41] [22] and in OmpSs+MPI PM [31].

B. Snapshots (SNAP) Design

Figure 2(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.

C. Checksum-based Scheme (CHKS) Design

Figure 2(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

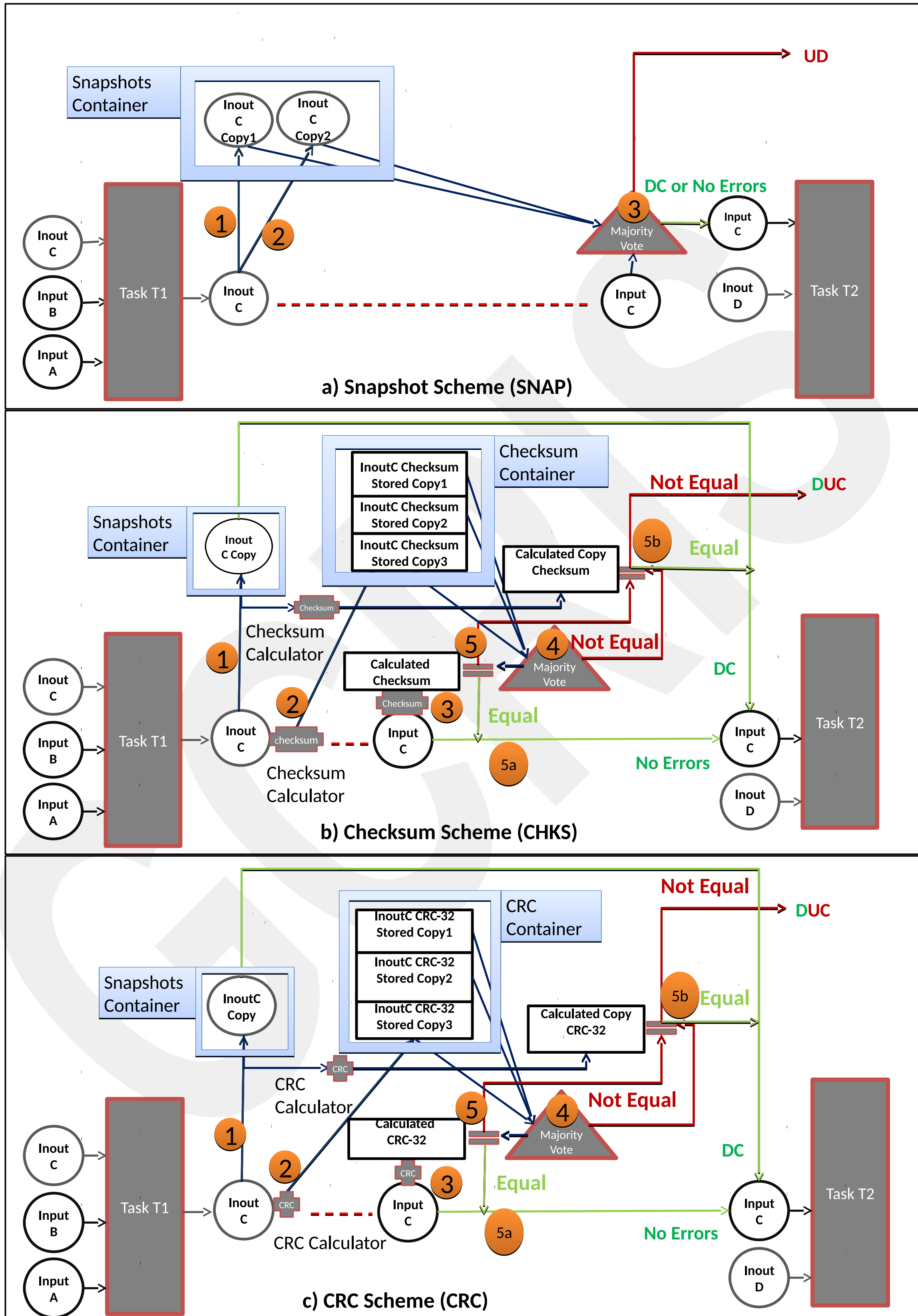

Fig. 2. The overall flow of the three memory reliability mechanisms.

Algorithm 1: 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);
```

the majority vote among its three stored copies. If they agree, the computation continues. Otherwise the application aborts.

D. 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 [25], 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 [14] polynomial, 0x11EDC6F41, which has a hamming distance of 4 for very long data words (up to 2^{31} bits). Figure 2(c) shows the 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.

E. 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 [20] ($_mm_crc32_u64$) 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 [37] and ARM [8] have already implemented this instruction staring with their Power8 and with ARMv8 64-bit architecture respectively. AMD implemented PCLMULQDQ instruction starting with Bulldozer processors [4] 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 1 shows details.

IV. EVALUATION

We implement our techniques in Nanos runtime [46]. We run our experiments in MareNostrum III supercomputer [2] hosted at Barcelona Supercomputing Center. We run experiments on both task-parallel shared-memory and task-parallel distributed MPI benchmarks listed in Table I [13]. 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 IV-A. 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 IV-B. Third, in Section IV-C, we evaluate the reliability of three mechanisms. We conclude the evaluation section with the Section IV-D where we compare and contrast the three schemes.

A. Benchmark Behavior Analysis

Table II 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 sampling and profiling

TABLE I. 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 II. 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

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 3 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

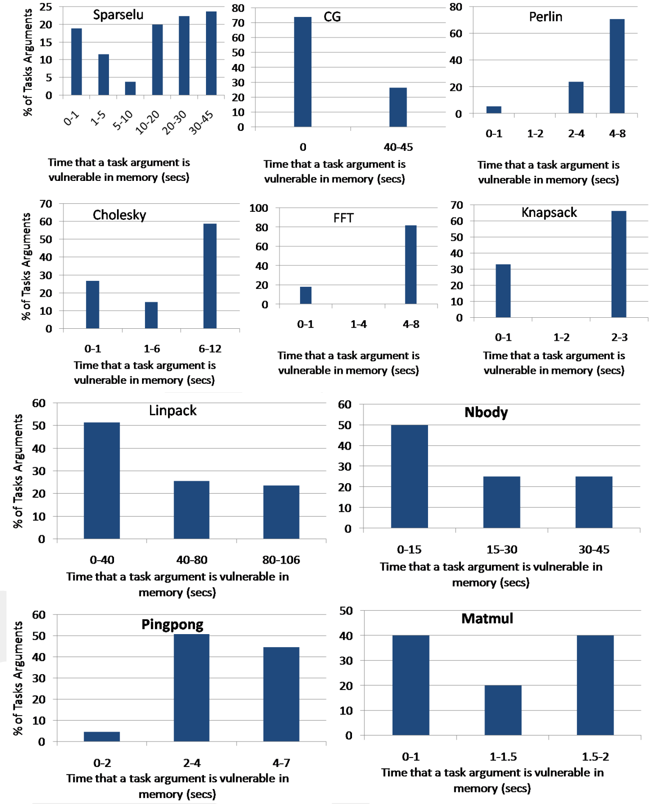

Fig. 3. Arguments waiting idle or being used but not modified by task computations

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.

B. Overhead and Scalability Results

Figure 4 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

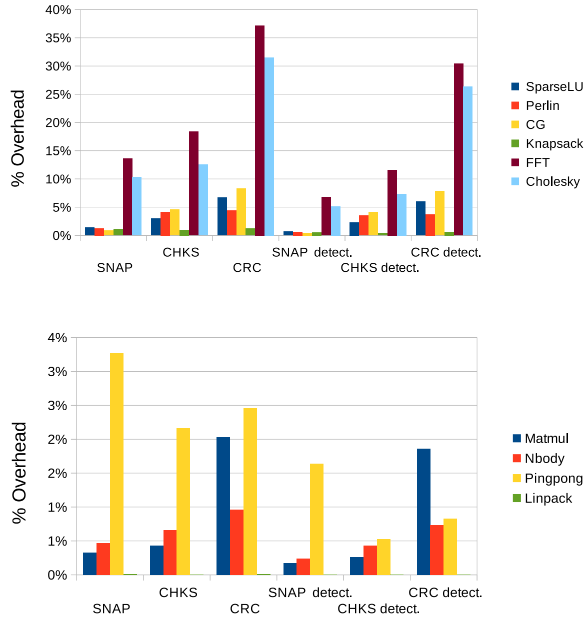

Fig. 4. Performance Overheads of Our Schemes

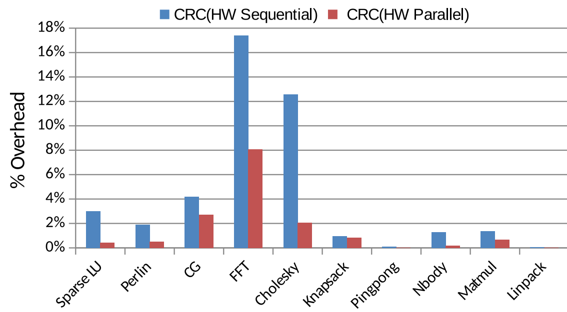

Fig. 5. CRC Hardware-based Overheads

granularity can resolve this issue. Figure 4 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 5 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 6 shows the scalability of three mechanisms for all

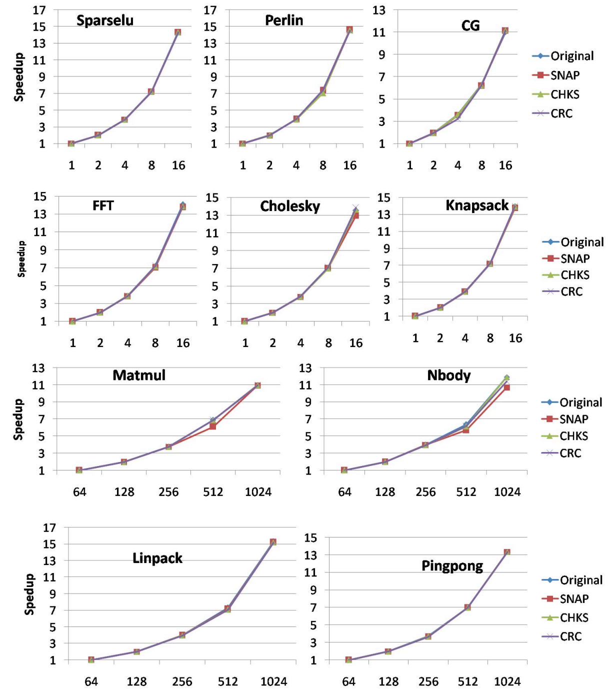

Fig. 6. Scalability of the Benchmarks

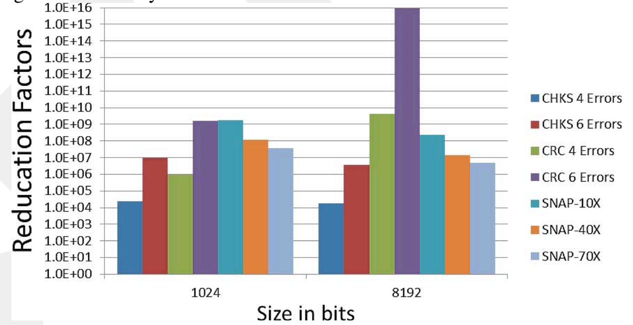

Fig. 7. Reduction Factors of Our Schemes

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 6 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.

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.

To protect from memory errors occurring during task computations, the computations themselves must be protected. Although this is not in the scope of this study, we implemented task replication as part of our framework. In task replication duplicates are executed on spare processors and wait for each other to compare their outputs to detect errors. The fault-free performance overheads (omitted) of replication vary from 0.1% (0.11% in total, Pingpong) to 7% (15% in total, FFT) showing that replication is also lightweight due to information available from dataflow. For more discussion and techniques on replication and selective replication, we refer reader to our complementary work [42]. In addition to task computation, for very high failure coverage, task-parallel runtimes themselves can be protected by what is called micro-checkpointing [23]. It is shown to have low overhead, 9% on average [23].

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

C. 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. [38] (Section IV-C1). In Section IV-C2, we perform memory vulnerability analysis of the benchmarks.

1) 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. [39], 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. [38], 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 $p = 1.68\times10^{-6}$. 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:

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

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 III-D. 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. [26] 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 [26]. Figure 7 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:

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

where N is the total number of bits in a task argument. Figure 7 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, 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:

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

where N is the total number of bits in a task argument, k is the number of errors occurring. Figure 7 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 7 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 III-E, 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 [44], [45], can be consulted to obtain the optimal task granularity for a task-parallel program.

TABLE III. 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%

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

2) 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 III (2nd and 5th columns) shows the reduced percentages of Architectural Vulnerability Factors (AVFs) [33] for all benchmarks. AVFs [33] 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:

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

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 as in our example program in Section II. 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 III (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.

D. 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) [17]. 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) [19], which corrupt the outputs of HPC applications without being detected. Table IV summarizes this discussion.

TABLE IV. 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

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. [32] and for systems like the Mont-blanc prototype [3].

V. CONCLUSIONS

In this work, 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.

VI. ACKNOWLEDGMENTS

This work was supported by FI-DGR 2013 scholarship and the European Community's Seventh Framework Programme [FP7/2007-2013] under the Mont-blanc 2 Project (www.montblanc-project.eu), grant agreement no. 610402 and TIN2015-65316-P.

REFERENCES

[1] 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.

[2] Marenostrum III system architecture: http://www.bsc.es/marenostrum-support-services/mn3, Accessed in January 2016.

[3] Mont-Blanc 2. http://www.montblanc-project.eu/.

[4] 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.

[5] Abdelhalim Amer, Naoya Maruyama, Miquel Pericas, 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.

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

[7] Michael Andersch, Ben Juurlink, and Chi Ching Chi. A benchmark suite for evaluating parallel programming models. Workshop on Parallel Systems and Algorithms, pages 7–17, 2011.

[8] 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.

[9] K. Bergman et al. Exascale computing study: Technology challenges in achieving exascale systems. 2008.

[10] 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.

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

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

[13] BSC. Application repository. https://pm.bsc.es/projects/bar/wiki/Applications, Accessed in January 2016.

[14] 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.

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

[16] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. In *Parallel Processing Letters*, volume 21, pages 173–193, 2011.

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

[18] 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.

[19] 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*, pages 78:1–78:12, 2012.

[20] 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.

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

[22] 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.

[23] Pavlos Katsogridakis and Polyvios Pratikakis. Micro-checkpointing in fault tolerant runtimes. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, pages 13:1–13:10, 2014.

[24] 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.

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

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

[27] 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.

[28] 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.

[29] 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.

[30] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, 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.

[31] Tatiana V. Martsinkevich, Omer Subasi, Osman S. Unsal, Franck Cappello, and Jesús Labarta. Fault-tolerant protocol for hybrid task-parallel message-passing applications. In *IEEE International Conference on Cluster Computing, CLUSTER*, pages 563–570, 2015.

[32] 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.

[33] 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.

[34] OpenMP Architecture Review Board. OpenMP application programming interface version 4.0. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, 2013.

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

[36] J. Reinders. *Intel Threading Building Blocks*. 2007.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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 *23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 99–102, 2015.

[42] 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*, pages 47:1–47:2, 2015.

[43] Vladimir Subotic, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Automatic exploration of potential parallelism in sequential applications. In *Supercomputing - 29th International Conference, ISC, Leipzig, Germany, June 22-26, 2014. Proceedings*, pages 156–171, 2014.

[44] Vladimir Subotic, Steffen Brinkmann, Vladimir Marjanovic, Rosa M. Badia, José Gracia, Christoph Niethammer, Eduard Ayguadé, Jesús 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.

[45] 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.

[46] 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*, pages 256–259, 2007.

[47] 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.