An Energy-Aware Fault Tolerant Scheduling Framework for Soft Error Resilient Cloud Computing Systems

Yue Gao Sandeep K. Gupta Yanzhi Wang Massoud Pedram
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, USA
yuegao@usc.edu, sandeep@usc.edu, yanzhiwa@usc.edu, pedram@usc.edu

***Abstract*— For modern high performance systems, aggressive technology and voltage scaling has drastically increased their susceptibility to soft errors. At the grand scale of cloud computing, it is clear that soft error induced failures will occur far more frequently, but it is unclear as to how to effectively apply current error detection and fault tolerance techniques in scale. In this paper, we focus on energy-aware fault tolerant scheduling in public, multi-user cloud systems, and explore the three-way tradeoff between reliability (in terms of soft error resiliency), performance and energy. Through a systematically optimized resource allocation, error detection approach selection, virtual machine placement, spatial/temporal redundancy augmentation and task scheduling process, the cloud service provider can achieve high error coverage and fault tolerance confidence while minimizing global energy costs under user deadline constraints. Our scheduling algorithm includes a static scheduling phase that operates on task graph based workload inputs prior to execution, and a light-weight dynamic scheduler that migrates tasks during execution in case of excessive re-executions. All schedules are evaluated on a runtime simulation engine that (1) mimics the performance fluctuations in cloud systems, and (2) supports the injection of arbitrary fault patterns. Compared to current virtual machine or task replication techniques, we are able to reduce overall application failure rates by over 50% with approximately 76% total energy overhead.**

I. INTRODUCTION

Soft error resiliency has become a major concern for modern computing systems as CMOS technology and voltage continues to scale [1]. *Soft errors* can occur in circuits due to *transient and intermittent faults* (henceforth referred to as *faults*) induced by noise, high energy cosmic particles, and hardware fatigue. As errors propagate through the system, they may manifest as different forms of *failures* such as corrupted outputs or system crash. At the grand scale of cloud computing, this problem can only worsen [2, 3, 4, 5, 6], especially for cost-effective cloud systems built with commodity components [7]. Researchers have witnessed unacceptably high failure rates when running scientific workloads in cloud or grid systems [8, 9].

Although it is impossible to entirely eliminate spontaneous soft errors, they can be masked from users to promote a satisfactory Quality-of-Service (QoS). This is achieved by predicting the faults/errors and applying the appropriate *error detection* and *fault tolerance* methods. Doing so will often incur large power consumption overheads, which will significantly impact operating costs [10]. In addition, in a multi-user cloud, protecting one application may stifle the performance for other applications because of resource sharing.

Although ad-hoc fault tolerance techniques such as virtual machine (VM) replication [11] and idempotent task retry [8] have already seen commercial success, as more and more providers and users are drawn to the cloud [12], a systematic approach is needed, especially for users running deadline or data accuracy sensitive applications. In this paper, we focus on soft error resiliency in public cloud systems from the perspective of the *cloud service provider* (CSP). We introduce a unified resource allocation and fault tolerant scheduling (FTS) framework that leverages the three core aspects in cloud computing: *reliability*, *performance* and *energy*. To achieve this, the CSP must select the appropriate *error detection* and *fault tolerance* measures for each user. Our framework is inspired by FTS techniques used for chip multiprocessors [13, 14], but properly updated for cloud computing systems. To the best of our knowledge, this is the first paper that analyzes the impact of error detection and fault tolerance mechanisms in order to co-optimize global energy and soft error resiliency under deadline constraints in a multi-user cloud environment.

We acknowledge the fact that cloud computing fault tolerance is still an emerging field without standardization. For example, how to accurately model hardware faults in the cloud is still largely unknown [5, 8]. With this in mind, our framework depicted in Fig. 1 is designed to be modular. It allows us to "plug-in" almost any fault model, any error detection and any fault tolerance option. In this paper, we will determine these inputs based on recent research.

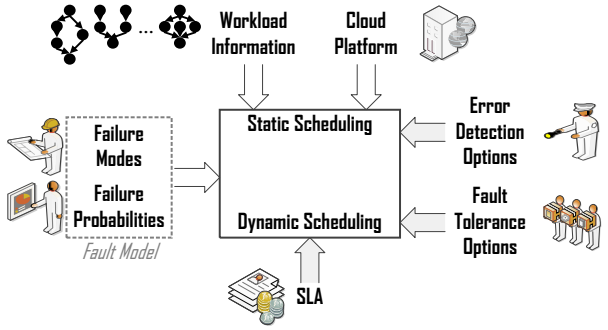

Fig. 1 Overview of the optimization framework and its inputs

II. RELATED WORK

For cloud computing systems, fault tolerance is demanded in data storage, transmission and computation. Providing fault tolerance in data storage [15] is relatively straightforward. RAID [16] setups and Amazon's EBS (Elastic Block Storage) [17] can tolerate disc failures through local redundant storage mediums. Many well known networking protocols can be revised to cover data transmission faults in cloud computing. Queue based message retrieval/retry mechanisms have already been utilized in Windows Azure [18].

978-3-9815370-2-4/DATE14/©2014 EDAA

Addressing errors that occur during computation in cloud systems is orthogonal to the two aforementioned items, and is the focus of this paper. Although cloud users can continue to utilize traditional algorithm level fault tolerance approaches [19], due to the opaqueness of the public cloud [20], it is significantly more efficient for the CSP to handle fault tolerance. The most popular high level approaches include *spatial redundancy* and *temporal redundancy*.

The concept of spatial redundancy can be traced back to Triple Modular Redundancy (TMR) [21]; when instantiated in the cloud computing context, modules translate to replicated VMs or tasks operating in lock-step [22, 23]. Such implementations can already be seen in high availability cloud solutions such as VMware [11] and VGrADS [24].

Temporal redundancy typically entails re-execution. Tasks can be resubmitted in case of a failure as seen in Condor-G [25] and MODISAzure [8], or restarted from an intermediate valid state [26]. Similarly, VM checkpointing methods periodically stores a checkpoint image of the primary VM in another backup VM. The VM pairs in Remus [27] operate in a leader-follower fashion. Kemari [28] and HydraVM [29] are other examples of VM checkpointing.

Thus far one important subject has escaped our discussion, namely error detection and error coverage. Error coverage is defined as the fraction of total errors that will be detected by the given error detection approach. Current cloud systems only address VM crash and timeout exceptions due to their ease of detection. However, not all faults manifest as such easily observable events [30]. Understanding application behaviors in the presence of soft errors is an active area of research. Statistical and proton irradiation fault injection experiments on the IBM POWER6 machine report that near 30% of unmasked latch flips lead to incorrect architectural state, which is the precursor to silent data corruptions [31]. In cloud systems, researchers have noticed that 30% - 60% of failed applications were unrecoverable [8, 9]. For cloud users that demand high reliability, we argue that both robust error detection and fault tolerance should be offered *transparently* by the CSP.

III. SYSTEM MODEL

Workload Model

We use directed acyclic graphs (DAGs) to model user *workloads*. The entire workload is represented as a collection of N disjoint DAGs: $\{G_1, G_2, ..., G_N\}$. Each DAG G_a ($1 \le a \le N$) represents a *workload request*, and each vertex T_i^a in G_a embodies a *task*. Without the loss of generality, we assume that each workload request belongs to a separate user. A directed edge from T_i^a to T_j^a denotes that T_j^a is dependent on the output of T_i^a. The weight of the edge $W_{i,j}^a$ represents the amount of data that needs to be passed from the predecessor task (T_i^a) to the successor task (T_j^a), or to the final output when $i = j$. By definition $W_i^a = \max(W_{i,j}^a)\ \forall j$. Other cloud frameworks that use similar task graph based workload models include Dryad [32] and Nephele [20]. The deadline for G_a is denoted as $L_{deadline}^a$. In this paper all deadlines are hard.

Tasks are run on VMs, which are categorized into K distinct types: $\{VM_1, VM_2, ..., VM_K\}$, each VM_g is coupled with a two-tuple integer set that specifies the CPU and memory resource requirements: $\{R_{CPU}^g, R_{MEM}^g\}$ [33].

Each task T_i^a is also coupled with a two-tuple integer input set $\{\theta_i^a, L_i^a\}$. θ_i^a represents the type of VM on which T_i^a can execute. This information can be provided externally, or deduced internally through collected statistics [20]. L_i^a is the estimated execution time of T_i^a, derived from approximation methods such as machine learning [20] or benchmark probing [9]. Scheduling optimizations will operate on L_i^a values, while the runtime simulation engine will account for VM performance fluctuations [9].

Cloud Platform Model

The cloud consists of a set of M servers: $\{D_1, D_2, ..., D_M\}$. The power consumption of D_x at time t includes the static power consumption $P_{static}^x(t)$ and the dynamic power consumption $P_{dynamic}^x(t)$. Both are correlated with the utilization rate of D_x at time t: $Util_x(t)$. We calculate $Util_x(t)$ by aggregating the CPU requirements of active VMs.

$P_{static}^x(t)$ is constant when $Util_x(t) > 0$, 0 otherwise. Servers have optimal utilization level in terms of performance-per-watt, which we define as Opt_x for D_x. It is commonly accepted that for modern servers $Opt_x \approx 0.7$, and the increase in dynamic power consumption beyond this operating point is more drastic than when $Util_x(t) < Opt_x$ [10, 34]. Even for identical utilization levels, the energy efficiency of different servers may vary [35]. This is captured by the coefficients α_x, β_x and γ_x, representing the power consumption increase of D_x when $Util_x(t) < Opt_x$ and $Util_x(t) \ge Opt_x$. The dynamic power consumption $P_{dynamic}^x(t)$ is then calculated as:

$$\begin{cases} Util_x(t) \cdot \alpha_x & (Util_x(t) < Opt_x) \\ Opt_x \cdot \alpha_x + (Util_x(t) - Opt_x)^{\gamma_x} \cdot \beta_x & (Util_x(t) \ge Opt_x) \end{cases}.$$

The total energy consumption is the sum of the power consumption across all servers throughout the timeline:

$$Total\_Energy = \sum_{x=1}^{M} \left(\sum_{t=1}^{t_{Max}} \left(P_{static}^x(t) + P_{dynamic}^x(t) \right) \right).$$

IV. FAULT MODEL

In this paper we use a fault model which is inspired by microprocessor fault tolerance research.

Task Failure Modes

Tasks can fail in the following two ways: (1) crash or (2) silent data corruption (SDC). Crashes are triggered by exceptions such as memory misalignment or transmission timeout. SDC refers to when a task produces erroneous output without triggering crashes [36]. We do not consider failures caused by middleware or software bugs.

Failure Probabilities

The failure probability of a particular task depends on many factors. We correlate this probability to two primary factors: (1) task execution duration and (2) the underlying hardware, i.e. the host server. If we use μ_x to characterize the failure rate of D_x, and adopt the widely used Poisson model for failure occurrences, then the failure rate of T_i^a would be:

$$FAIL_i^a = 1 - e^{-\mu_x L_i^a}.$$

The discrepancy between server failure rates are justified by the fact that servers in the cloud are often heterogeneous, in

terms of both hardware infrastructure and operating environments: temperature, supply voltage [37], and age [3].

The probability of a task failure being a crash (CRS_i^a) or silent data corruption (SDC_i^a) are calculated as $FAIL_i^a \cdot \rho_1$ and $FAIL_i^a \cdot \rho_2$, respectively. ρ_1 and ρ_2 can be approximated with empirical data.

V. ERROR DETECTION

In this paper we consider two error detection approaches corresponding to the two types of VM failures: crash detection and explicit output comparison (EOC).

VM sensors are very cost effective in detecting application crashes [8]. In this paper, crash detection is presumed to be ubiquitously deployed. With EOC, tasks are spatially replicated to enable output data comparison. EOC is error detection in its most powerful form, reaching near perfect error coverage [13]. This confidence comes with the price of extra VM allocations to run the task replicas and the additional time to perform output comparisons, which we reasonably relate to the amount of data that needs to be compared (W_i^a).

VI. FAULT TOLERANT CLOUD SCHEDULING

Our fault tolerant cloud scheduling framework is composed of two phases: static scheduling and dynamic scheduling.

Static Scheduling

During this phase, the CSP manipulates workloads at the granularity of users $\{G_1, G_2, ..., G_N\}$ and performs three assignments. First, the CSP performs resource allocation for each user G_a by setting the integer array $\{\tau_1^a, \tau_2^a, ... \tau_k^a\}$ where τ_i^a is the number of VM_i's allocated to G_a. The goal of this process is to provide each user with sufficient amount VMs so that the deadline can be met during execution.

Second, the CSP must establish, for each user G_a, the level of "effort" devoted to error detection and fault tolerance by associating each user G_a with the following:

***DETECT**$_a$* 0: Crash Detection Only, 1: EOC
***REP**$_a$* Replication Factor, ≥ 2 if $DETECT_a$ = 1

The detection method ($DETECT_a$) influences the task runtimes. When EOC is used, output comparisons will increase the task execution time of T_i^a from L_i^a to $L_i^a + W_i^a$. The replication factor (REP_a) denotes the number of task replicas.

Third, each user is mapped to a server (we assume that one application does not span across multiple servers, but one server may host multiple applications), and a temporal schedule is generated. The objective of the CSP in this stage is to *systematically maximize fault tolerance confidence and error coverage while minimizing global energy consumption under deadline constraints*. While energy consumption can be calculated using the aforementioned formula, the quantification of fault tolerance confidence and error coverage is not so straightforward. A brute force method would be to implement a comprehensive fault simulator, but such an approach is incompatible with optimization algorithms due to repeated evaluations during the solution space exploration. We propose a computationally attractive approach based on dynamic programming (DP) described below.

The fault tolerance confidence for each user G_a is measured with two probabilities:

1) P_DROP_a: The probability of the user request being dropped due to errors being detected but unable to be tolerated within the deadline. High P_DROP_a values indicate low fault tolerance confidence.
2) P_ERR_a: The probability of the CSP delivering erroneous outputs to the user due to undetected SDC. High P_ERR_a values indicate low error coverage.

P_ERR_a can be approximated in linear time as the probability of at least one non-detectable error occurring during execution:

$$P\_ERR_a = \prod\nolimits_{\forall i} (1 - FAIL_i^a \cdot \rho_2).$$

P_DROP_a is calculated using a dynamic programming based algorithm of complexity $O(N\_Tasks_a \cdot L_{deadline}^a)$ described in Algorithm 1. This fast evaluation of fault tolerance confidence is a key enabler for the static scheduling algorithm.

Algorithm 1. Calculating P_DROP_a

```
N_Tasks_a = Number of tasks in G_a;
SLACK_a = L^a_deadline − Static_Schedule_Length(G_a);
DP_TABLE[][] = {0};
for (i = N_Tasks_a; i ≥ 1; i--)
  if (DETECT_a = 0)        Adj_L^a_i = L^a_i;
  else if (DETECT_a = 1)   Adj_L^a_i = L^a_i + MAX(W^a_{i,x});
Max_a = Σ_{i=0}^{N_Tasks_a} Adj_L^a_i;
for (t = L^a_deadline; t ≥ 0; t--)
  for (i = N_Tasks_a; i ≥ 1; i--)
    if (Σ_{j=i}^{N_Tasks_a} Adj_L^a_i + t > L^a_deadline)
      DP_TABLE[i][t] = 0;
    else
      DP_TABLE[i][t] = Success(T^a_i) * DP_TABLE[i + 1][t + Adj_L^a_i]
                     + [1 - Success(T^a_i)] * DP_TABLE[i][t + Adj_L^a_i];
return (1 - DP_TABLE[0][0]);
```

Both dropping requests and delivering erroneous outputs will cause customer dissatisfaction and potential profit loss for the CSP. In this paper we do not presume any profit model, and only operate on raw failure probabilities. This allows for the CSP to superimpose cost functions on our framework.

We examine a simple case study for a single task (T_1^1) user G_1 below to gain some insight on the advantages and disadvantages of different error detection mechanisms. Three example schedules are presented in Table I. Formulas for calculating P_ERR_1 and P_DROP_1 values are listed in Table II, extrapolated to a generic replicator factor k.

Table I. Error detection and active replication combinations (static schedule)

	$DETECT_1 = 0$ $REP_1 = 1$	$DETECT_1 = 0$ $REP_1 = 2$	$DETECT_1 = 1$ $REP_1 = 3$
Example Schedule for T_1^1	L_1^1	L_1^1 L_1^1	L_1^1 L_1^1 L_1^1 =
Single Crash	Re-execute	Seamless	Seamless
Single SDC	Erroneous	Erroneous	Seamless

In the first column of Table I, the task is not enhanced with any error detection or fault tolerance beyond crash detection and task retry. The second column showcases the basic implementation of spatial redundancy in cloud systems [11, 22]. This schedule can seamlessly recover from a single task crash by gathering outputs from the healthy replica. In the event of both replicas crashing, the task can be re-executed. In this case $1 - P\_DROP_1$ follows the geometric distribution. The rightmost column in Table I is a almost bulletproof TMR

schedule. It not only reaches full error coverage, but also supports seamless recovery through output voting.

We plot the values from Table II in Fig. 2. Clearly, the benefits of having longer slacks and higher replication factors suffer from the phenomena of diminished returns. Conceptually, the optimization procedure seeks for the knee of these two curves so that the failure rates can be reduced without excessive VM allocations.

Table II. P_ERR_1 and P_DROP_1 values of the example schedules

	P_ERR_1	P_DROP_1
$DETECT_1$=0 REP_1=k	$1-[1-FAIL_1^1\cdot\rho_2]^k$	$(FAIL_1^1\cdot\rho_1)^{k\left\lfloor\frac{L_{deadline}^1}{L_1^1}\right\rfloor}$
$DETECT_1$=1 REP_1=k	0	$\left[FAIL_1^{1\,k}+k\cdot FAIL_1^{1\,k-1}(1-FAIL_1^1)\right]^{\left\lfloor\frac{L_{deadline}^1}{L_1^1+W_1^1}\right\rfloor}$

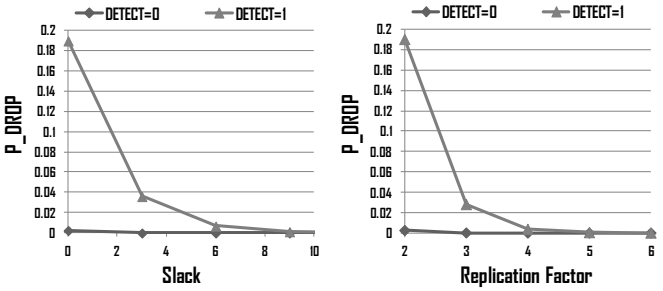

Fig. 2. Analysis of the tradeoffs during scheduling

Next we present the static scheduling algorithm in high level pseudo code form in Algorithm 2. It first operates in batch mode (batch mode and dependency mode scheduling classifications are defined in survey [38]) to keep the algorithm runtime tractable. In this stage, we combine bin-packing which has been proven useful for cloud scheduling [39], and core concepts from genetic algorithm [40]. However, the actual procedure is not evolutionary, it is deterministic with a complexity of $O(M\cdot N^2\cdot N\_Tasks_a\cdot L_{deadline}^a)$. Next, the static schedule is placed on the timeline for dependency mode scheduling. VM allocations and task migrations will be performed to accelerate applications with deadline violations.

Algorithm 2. Static Scheduling Algorithm

```
call resource_allocation();
DETECT_a = 0 and REP_a = 1 ∀ 1 ≤ a ≤N
Schedule[0] = call BEST_FIT_FOR_POWER_BIN_PACKING();
for (k = 0 and k = 1)
    DETECT_a = k and REP_a = 1 ∀1 ≤ a ≤N
    Schedule[k] = call BEST_FIT_FOR_RELIABILITY_BIN_PACKING();
    select G_i that REP_i++ maximizes increase in (1 - P_DROP_i)/Total_Energy;
    if (increase in P_DROP < 1.5x)
        REP_i--;
for (i = 1; i ≤ N; i++)
    if (P_DROP_i[2] < P_DROP_i[1] - P_ERR_i[1])
        accept(G_i, Schedule[2]);
for (all unscheduled applications)
    if (valid(accept(G_i, Schedule[1])))
        accept(G_i, Schedule[1]);
for (all unscheduled applications)
    accept(G_i, Schedule[1]) or place(G_i, select(D_x));
call implement_placement _schedule_to_timeline();
for (i = 1; i ≤ N; i++)
    if (deadline_violated(G_i))
        if (can_execute_sooner(T_k^i) == 1)
            allocate_and_migrate(T_k^i);
    for (j = 1; j ≤ N_TASKS[i]; i++)
        add_replica_oppurtunistic(i, j);
output can_execute_sooner[];
output Static_Schedule;
```

Dynamic Scheduling

During cloud operation, the static schedule will only serve as a reference, since it cannot handle task runtime variations and spontaneous faults. Distributed dynamic schedulers will:

1) *Manage output comparisons and initiate re-execution when appropriate*. We illustrate with a simple example. Suppose the cloud is servicing a single user G_1 (Table III). Fig. 3 displays a dynamic trace, hence all latencies shown (L') are actual latencies, which may or may not be the same as the estimated values (L) due to VM performance fluctuations. G_1 is augmented with EOC with REP_1 = 2. Re-execution was initiated at t = 10 due to the comparison mismatch for T_2^1, delaying subsequent tasks. At t = 23, one T_5^1 replica crashed, hence the comparison at t = 25 was nullified, since only one set of output remain, which may have been corrupted by SDC. We conservatively initiate re-execution in this scenario.
2) *Initiate dynamic allocation and task migration when appropriate*. Based on the re-executions, dynamic scheduling is carried out with Algorithm 3. The dynamic scheduler is designed to be as simple as possible to comply with real time constraints. For example, the dynamic scheduler does not examine whether a re-executed task is on the critical path or not, it simply assumes all re-executions will extend the schedule length.

Algorithm 3. Dynamic Scheduling Algorithm

```
if (application i task k initiated re-execution at time t)
    Adjusted_Static_Schedule_Length(G_a) += L_k^i;
    if (Adjusted_Static_Schedule_Length(G_a) ≥ L_deadline^i)
        for (k' = 1; k' ≤ number of tasks in G_a; k'++)
            if (can_execute_sooner(T_k'^i) == 1 && start_time(T_k'^i) > t)
                migrate_and_allocate(T_k'^i) ∀T_k'^i statically scheduled after T_k^i;
```

Returning to previous case study, the re-execution at t = 10 caused the dynamic allocation of a pair of VM_1's (fast VM allocations has been explored in SnowFlock [41]), allowing for the migration of T_3^1, T_4^1 and T_5^1. This solved the anticipated deadline violation, and created opportunities for future re-executions, which came into effect when T_5^1 was re-executed.

Table III. Task graph, deadline and task latency information

User 1 Deadline: 30	Task (T/θ)	Estimated Latency (L)	Observed Latency (L')
	$T_1^1/1$	4	4
	$T_2^1/1$	3	4
	$T_3^1/1$	4	4
	$T_4^1/1$	6	6
	$T_5^1/1$	4	3

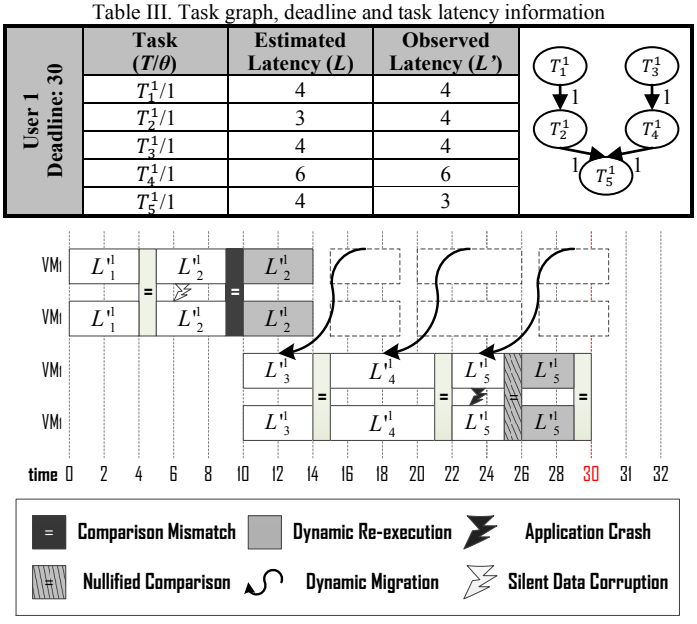

Fig. 3 Dynamic allocation and migration

Runtime Simulation Engine

In order to break away from analytical evaluations, we implemented a runtime simulation engine that is capable of simulating the cloud environment, integrating the abilities to:

1) *Reflect VM performance fluctuations*. The task latencies during runtime can deviate from the estimated latencies.
2) *Inject failures at arbitrary times*. Failures can occur at arbitrary times, not necessarily matching the fault model assumed by the scheduling algorithm. Fail logs can also be used to recreate realistic fault/failure patterns.

VI. EXPERIMENTAL RESULTS

Experiments on Large Scale Workloads

We demonstrate the effectiveness of our scheduling framework with large scale workload inputs on large scale cloud platforms. Table IV summarizes the input ranges, along with some fault model parameters.

Table IV. System model and fault model input parameter ranges

Cloud Platform Parameters		User Workload Characteristics			Fault Model Parameters	
No. of Servers	μ_x Variation	Total No. of Users	Tasks per User	Task Latency	ρ_1	ρ_2
10 - 30	1x - 2x	30 - 60	20 - 100	1 - 10	90%	10%

We created 12 sets of indexed experiments (*EX_1* to *EX_12*) with randomly generated task graphs. The cloud platform is capable of hosting all applications if $REP_a = 4 \; \forall \, a$. Results are presented in Table V. For each experiment we compare our optimized schedule with three reference schedules introduced in Table I. The first reference schedule (Ref_1) is a bare bones schedule with no added detection or replication. This schedule is extremely energy efficient but highly susceptible to soft errors. The second reference schedule (Ref_2) uses a simple one-size-fits-all methodology found in commercial tools like VMware to combat faults: VM/task duplication ($REP_a = 2 \; \forall \, a$). Ref_2 aims to reduce failure induced deadline violations. It is important to note that we assume 90% of all failures will trigger crashes, so as to give a clear advantage to Ref_1 and Ref_2, since both are bottlenecked by the error coverage of native crash detection. The third reference schedule (Ref_3) employs TMR plus support for re-execution across the time domain. Without our optimization procedure, this would be a feasible solution if high error coverage and fault tolerance confidence is critical.

Table V. Results for large scale workload inputs

Index	Total Energy Overhead			Averaged *P_DROP* Improvement (Δ*P_DROP*)			Averaged *P_ERR* Improvement (Δ*P_ERR*)		
	Over Ref_1	Over Ref_2	Over Ref_3	Over Ref_1	Over Ref_2	Over Ref_3	Over Ref_1	Over Ref_2	Over Ref_3
EX_1	149.9%	57.5%	-25.2%	4.1%	-0.4%	0.6%	8.4%	21.1%	-4.7%
EX_2	140.4%	84.5%	-11.4%	31.7%	1.7%	0.0%	12.7%	33.6%	-0.6%
EX_3	219.8%	105.1%	-4.2%	31.3%	-7.3%	0.3%	7.7%	50.3%	-17.7%
EX_4	103.8%	95.9%	23.5%	45.9%	11.4%	10.1%	28.8%	64.6%	0.0%
EX_5	100.3%	36.5%	-7.6%	68.6%	3.8%	0.7%	9.3%	45.9%	-1.4%
EX_6	73.0%	20.3%	-23.3%	74.1%	9.0%	0.0%	4.8%	46.2%	0.0%
EX_7	99.3%	38.3%	-15.8%	69.1%	9.7%	0.0%	6.6%	49.1%	-0.3%
EX_8	202.4%	84.3%	-21.4%	43.7%	7.1%	0.3%	21.4%	58.9%	-1.7%
EX_9	352.3%	192.2%	61.1%	61.6%	24.0%	2.5%	29.6%	65.6%	0.0%
EX_10	179.0%	69.1%	-6.7%	79.5%	3.5%	0.0%	10.0%	59.8%	0.0%
EX_11	118.5%	49.7%	-30.9%	4.6%	-0.3%	1.3%	7.9%	19.9%	-5.5%
EX_12	158.7%	80.3%	-15.5%	32.9%	1.6%	0.0%	10.5%	32.7%	-0.6%
Avg.	**158.1%**	**76.2%**	**-6.5%**	**45.6%**	**5.3%**	**1.3%**	**13.1%**	**45.6%**	**-2.7%**

For each index we carry out 100 fault injection runs on the runtime simulation engine to obtain averaged statistics. We evaluate the optimized schedule in terms of three aspects: observed power overhead, observed *P_ERR* improvement and observed *P_DROP* improvement. Due to space limitations, the latter two are derived from averaging the *P_ERR* and *P_DROP* values across all users.

The error detection and replication factor configurations for the first three experiments are show in Table VI to prove that the optimized schedules are not trivially uniform. For *EX_1*, the optimized schedule has a 2.5 times higher (149.9% overhead) energy consumption compared to that of Ref_1. This trades for 4.1% and 8.4% improvement in *P_DROP* and *P_ERR*, respectively, which can be roughly translated to a 12.5% decrease in overall failure rate. Compared to Ref_2 which is designed to minimize *P_DROP*, their *P_DROP* values are almost identical, but our optimized schedule improved *P_ERR* by 21.1%. Compared to Ref_3 (TMR), *P_DROP* values are again almost identical, but energy consumption decreased by over 25%, at the cost of raising *P_ERR* by 4.7%. The situation is similar when the optimized schedule from *EX_2* is compared with its Ref_3: with identical *P_DROP* values, 0.6% *P_ERR* was sacrificed for 11.4% energy improvement. On the other hand, the algorithm behavior for *EX_4* is quite different: seeing that Ref_3 still maintains a high *P_DROP*, an additional 23.5% of energy is devoted to decreasing *P_DROP* by over 10%, while allowing the result schedule to remain immune to SDCs. Due to the diminishing returns when increasing *REP*, in some cases, chasing after the last few percent of *P_DROP* could become challenging. This is reflected in *EX_9*: when the Ref_3 saw a *P_DROP* value of 3.45%, the algorithm returned with a schedule that decreased this value to below 1%, at the cost of 60% energy overhead.

Table VI. Error detection and replication factor value compositions

Index	Error Detection Percentage		Replication Factor Percentage			
	Crash	EOC	1	2	3	4
EX_1	40%	60%	40%	43%	17%	0%
EX_2	4%	96%	3%	93%	4%	0%
EX_3	17%	83%	17%	20%	57%	6%

The two key takeaways from this analysis are:

1) Addressing error coverage and fault tolerance in large scale cloud systems is expensive in terms of energy consumption. To prevent this overhead from spiraling out of control, the scheduler must be intelligent, and the CSP should use optimization frameworks like the one presented in this paper to assess the economic viabilities of guaranteeing fault tolerance to the users.
2) Our framework achieves a balance between minimizing *P_DROP*/*P_ERR* and maximizing energy efficiency. The chosen operating point is significantly more reliable than Ref_1 and Ref_2 with average total energy overheads of 158.1% and 76.1% respectively. It is more energy efficient than TMR when TMR is sufficient, but reaches beyond TMR when needed.

Fault Tolerance Confidence Evaluation

The calculation of *P_ERR* and *P_DROP* (Algorithm 1) is crucial to the scheduling algorithm. In Fig. 4 we plot the static (calculated) and observed (simulated) *P_DROP* and *P_ERR*

values for Ref_1 of EX_1, at the granularity of individual applications. Our evaluation methods performed rather well.

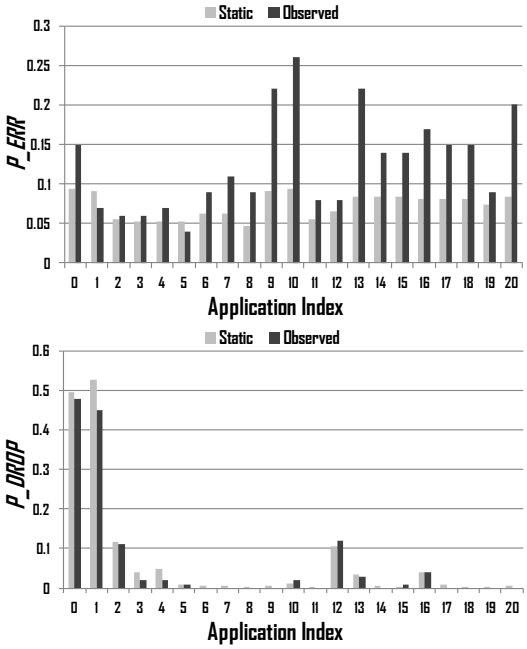

Fig. 4. User failure statistics

VI. CONCULSION

Soft error resiliency is a major concern for future microprocessors. At the grand scale of cloud computing, this issue can only worsen without appropriate countermeasures. Although ad-hoc methods such as VM duplication or task retry have seen success, current cloud systems are far from fully protected. At least two critical aspects remain unclear: (1) whether current approaches will be sufficient for deadline or data accuracy sensitive applications, and (2) how to manage the efforts devoted to error detection and fault tolerance in a large scale multi-user environment.

This paper focuses on energy-aware FTS in public cloud systems, and explores the three-way tradeoff between reliability, performance and global energy consumption. Our static plus dynamic scheduling and optimization framework enables the CSP to achieve high error coverage and fault tolerance confidence while minimizing global energy costs under user deadline constraints. The CSP can superimpose profit models on our framework, and through our runtime simulation engine, evaluate the financial gains and cost of providing highly reliable computations to the cloud users.

REFERENCES

[1] C. Weaver et al., "Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor," *Int'l. Symp. on Computer Architecture*, 2004.
[2] L. A. Barroso and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis Lectures on Computer Architecture*, 2009.
[3] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," *Symp. on Cloud Computing*, 2010.
[4] R. Garg and A. K. Singh, "Fault tolerance in grid computing: State of the art and open issues," *Int'l Journal of Computer Science & Engineering Survey*, 2(1), 2011.
[5] Y. Liang et al., "BlueGene/L failure analysis and prediction models," *Int'l Conf. on Dependable Systems and Networks*, 2006.
[6] T. Nguyen and J.-A. Desideri. "Resilience issues for application workflows on clouds," *Int'l Conf. on Networking and Services*, 2012.
[7] J. Hamilton, "An architecture for modular data centers," *CIDR*, 2007.
[8] J. Li et al., "Fault tolerance and scaling in e-Science cloud applications: observations from the continuing development of MODISAzure," *Int'l. Conf. on e-Science*, 2010.
[9] G. Kandaswamy et al., "Fault tolerance and recovery of scientific workflows on computational grid," *Int'l Cluster Computing*, 2008.
[10] M. Pedram, "Energy-efficient datacenters," *IEEE Trans. on CAD*, 2012.
[11] http://www.vmware.com/products/fault-tolerance.
[12] B. Hayes, "Cloud Computing," *Communications of the ACM*, 2008.
[13] Y. Gao et al., "Using explicit output comparisons for fault tolerant scheduling (FTS) on modern high-performance processors," *Design Automation & Test in Europe*, 2013.
[14] V. Izosimov et al., "Design optimization of time- and cost-constrained fault-tolerant distributed embedded systems," *Design Automation & Test in Europe*, 2005.
[15] D. J. Abadi, "Data management in the cloud: Limitations and opportunities," *IEEE Data Eng. Bull*, 32(1): 3-12, 2009.
[16] D. A. Patterson et al., "A case for redundant arrays of inexpensive disks (RAID)," *Int'l. Conf. on Management of Data*, 1988.
[17] http://aws.amazon.com/ebs.
[18] W. Lu et al., "AzureBlast: a case study of developing science applications on the cloud," *High Performance Distributed Computing*, 2010.
[19] J. Deng et al., "Fault-tolerant and reliable computation in cloud computing," *GLOBECOM Workshops*, 2010.
[20] D. Warneke and O. Kao, "Nephele: efficient parallel data processing in the cloud," *SC-MTAGS*, 2009.
[21] R. E. Lyons and W. Vanderkulk, "The Use of Triple Modular Redundancy to Improve Computer Reliability," *IBM Journal of Research and Development*, 7(2):200–209, 1962.
[22] R. Jhawar et al., "A comprehensive conceptual system-level approach to fault tolerance in cloud computing," *Int'l. Systems Conf.,* 2012.
[23] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," *ACM Trans. on Computer Systems*, 14(1): 80-107, 1996.
[24] L. Ramakrishnan et al., "VGrADS: enabling e-Science workflows on grids and clouds with fault tolerance," *High Performance Computing Networking, Storage and Analysis*, 2009.
[25] J. Frey et al., "Condor-G: A computation management agent for Multi-Institutional Grids," *High Performance Distributed Computing*, 2001.
[26] N. R. Rejinpaul and L. Maria Michael Visuwasam, "Checkpoint-based intelligent fault tolerance for cloud service providers," *Int'l. Journal of Computers & Distributed Systems*, 2(1): 59-64, 2012.
[27] B. Cully et al., "Remus: High availability via asynchronous virtual machine replication," *USENIX Symp. on Networked Systems Design and Implementation*, 2008.
[28] Y. Tamura et al., "Kemari: Virtual machine synchronization for fault tolerance," *USENIX Annual Technical Conf.*, 2008.
[29] K. Y. Hou et al., "HydraVM: Low-cost, transparent high availability for virtual machines," HP Laboratories Tech. Rep., 2011.
[30] S. Chandra and P. M. Chen, "The impact of recovery mechanisms on the likelihood of saving corrupted state," *Int'l Symp. on Software Reliability Engineering*, 2002.
[31] P. N. Sanda et al., "Soft-error resilience of the IBM POWER6 processor," *IBM Journal of Research and Development*, *52*(3): 275-284. 2008.
[32] M. Isard et al., "Dryad: distributed data-parallel programs from sequential building blocks," *EuroSys*, 2007.
[33] Y. Gao et al., "An energy and deadline aware resource provisioning, scheduling and optimization framework for cloud systems," *Int'l Conf. on Hardware/Software Codesign and System Synthesis*, 2013.
[34] G. Chen et al., "Energy-aware server provisioning and load dispatching for connection-intensive internet services," *NSDI*, 2008.
[35] P. Greenhalgh, "Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7," *ARM White Paper*, 2011.
[36] O. Khalili et al., "Measuring the performance and reliability of production computational grids," *Int'l Conf. on Grid Computing*, 2006.
[37] V. Chandra and R. Aitken, "Impact of technology and voltage scaling on the soft error susceptibility in nanoscale CMOS," *Int'l. Symp. on Defect and Fault Tolerance of VLSI Systems*, 2008.
[38] S. Xavier and S. J. Lovesum, "A survey of various workflow scheduling algorithms in cloud environment," *International Journal of Scientific and Research Publications*, 3(2), 2013.
[39] S. Srikantaiah et al., "Energy aware consolidation for cloud computing," *Cluster Computing*, 12(1): 1–15, 2009.
[40] D. E. Goldberg and J. H. Holland, "Genetic algorithms and machine learning," *Machine learning*, 3(2): 95-99, 1988.
[41] H. A. Lagar-Cavilla et al., "SnowFlock: rapid virtual machine cloning for cloud computing," *European conference on Computer systems*, 2009.