Noname manuscript No.
(will be inserted by the editor)

System-Wide Tradeoff Modeling of Performance, Power, and Resilience on Petascale Systems

Li Yu · Zhou Zhou · Yuping Fan · Michael E. Papka · Zhiling Lan

Received: date / Accepted: date

Abstract While performance remains a major objective in the field of high-performance computing (HPC), future systems will have to deliver desired performance under both reliability and energy constraints. Although a number of resilience methods and power management techniques have been presented to address the reliability and energy concerns, the tradeoffs among performance, power, and resilience are not well understood, especially in HPC systems with unprecedented scale and complexity. In this work, we present a co-modeling mechanism named TOPPER (system-wide TradeOff modeling for Performance, PowEr, and Resilience). TOPPER is build with colored Petri nets which allow us to capture the dynamic, complicated interactions and dependencies among different factors such as workload characteristics,

Li Yu
Department of Computer Science
Illinois Institute of Technology
E-mail: lyu17@hawk.iit.edu

Zhou Zhou
Department of Computer Science
Illinois Institute of Technology
E-mail: zzhou1@hawk.iit.edu

Yuping Fan
Department of Computer Science
Illinois Institute of Technology
E-mail: yfan22@hawk.iit.edu

Michael E. Papka
Department of Computer Science
Illinois Institute of Technology
E-mail: papka@anl.gov

Zhiling Lan
Department of Computer Science
Illinois Institute of Technology
E-mail: lan@iit.edu

hardware reliability, runtime system operation, on a petascale machine. Using system traces collected from a production supercomputer, we conducted a series of experiments to analyze various resilience methods, power capping techniques, and job characteristics in terms of system-wide performance and energy consumption. Our results provide interesting insights regarding performance-power-resilience tradeoffs on HPC systems.

Keywords Performance-Power-Resilience Modeling · Tradeoff Analysis · Petaflop Systems · Colored Petri Nets

1 Introduction

Over the past three decades we have witnessed advances of roughly nine orders of magnitude in computing capability. These advances allow computational and multi-scale simulations of unprecedented scope and accuracy [34]. Today, the widespread availability of HPC systems enables computational scientists to attack problems that are much larger and more complex, leading to revolutionary scientific discoveries. As supercomputers continue to grow in scale and complexity, *reliability* and *energy* become paramount concerns. Studies have shown that the mean-time-between-failures (MTBF) of production HPC systems, even those built using ultra-reliable components, are only on the order of 10-100 hours [40]. Meanwhile, the days of "performance at all costs" are quickly coming to an end, as the energy costs of operating system components and cooling the machine room are becoming excessive. The tight energy budget introduces the need for power management on extreme scale systems.

With the growing concern of system reliability, a number of resilience methods have been developed for preventing or mitigating failure impact. Well-known resilience methods include checkpoint/restart and replication. Checkpoint/restart is a long-standing fault tolerance technique [43]. A number of techniques such as multi-level checkpointing have been proposed to improve checkpoint efficiency [33]. Replication-based approaches improve application resiliency by replicating data or computation [29]. While many studies have been presented to evaluate checkpoint-based methods or replication-based techniques, existing studies mainly focus on *application-level* performance analysis (i.e., focusing on performance impact on a single application's execution time). Furthermore, it is unclear how different resilience methods could impact system performance such as overall resource utilization and system-wide energy cost.

Meanwhile, in order to address the energy concern, a number of power management mechanisms have been presented in the past years [24,8,38]. Power-capping (or power budgeting) is a well-known approach to limit the maximal power of a system. Power-capping methods consist of both software-based (e.g., through task/job scheduling) [52,9,11] and hardware-based (e.g., through dynamic voltage and frequency scaling (DVFS)) [22,17,30]. To control the aggregated power within a predefined threshold, the former controls the overall system power by dynamically scheduling the queued jobs according to

their expected power consumption, while the latter limits the overall system power by adjusting processor or core frequency.

While resilience methods and power-capping techniques continue to evolve, tradeoffs among execution time, energy efficiency, and resilience operations are not well understood in the field of HPC. Existing studies mainly focus on pairwise tradeoff analysis, such as tradeoffs between performance and reliability [20,41,16], or tradeoffs between performance and energy [30,52,9,11]. Understanding the tradeoffs among all three factors (i.e., performance, energy, and reliability) is crucial, as future machines will be built under both reliability and energy constraints. As an example, although both power-aware job scheduling and DVFS address the HPC energy concern, they can make different impacts on hardware reliability, hence affecting system performance. Studies indicate that DVFS could impact hardware lifetime reliability, resulting in up to a 3 times higher failure rate [44]. Consequently, it is not clear how these power management techniques impact system-wide performance if considering their potential reliability impact. Tradeoff analysis of performance, energy, and resilience on extreme scale systems is challenging. In particular, the analysis is influenced by various *dynamic* factors such as dynamic job submission, component failure, dynamic frequency tuning, and runtime operations (e.g., adaptive scheduling to enforce a power cap).

Models are ideal tools for navigating a complex design space and allow for rapid evaluation and exploration of detailed what-if questions. Existing modeling methods include analytical modeling, simulation/emulation, and queuing theory based modeling. Nevertheless, none of these methods provide sufficient support for building a high-fidelity model for the aforementioned tradeoff analysis due to various limitations. Analytical modeling methods are fast, but cannot capture dynamic changes of the system. Trace-based simulation can provide highly-accurate representations of system behaviors; however, extending existing simulators that are developed for performance analysis is not trivial, and would require a significant amount of engineering efforts. The conventional queuing methods (e.g., Markov modeling) suffer from the state explosion problem, thus are not scalable enough.

In this paper, we present a modeling and analysis mechanism named *TOPPER* (system-wide TradeOff modeling for Performance, PowEr, and Resilience). It is designed for *system-wide* quantitative analysis of execution time, energy efficiency, and reliability on HPC systems. Here, system-wide analysis means that our study concentrates on the *aggregated* performance and energy consumption of a batch of applications (or jobs), rather than the performance of a single application (or job). Specifically, we seek to answer the following fundamental questions:

Q1: How would different resilience methods (e.g., checkpoint/restart and replication) impact system-wide performance and energy consumption?
Q2: How would different parameter settings (e.g., checkpoint overhead, redundancy degree, and application communication/computation ratio) affect the performance of resilience methods?

Q3: How would different power management techniques (e.g., power-aware job scheduling and DVFS) impact system-wide performance by considering their potential reliability impact?

Encouraged by our prior study of power performance tradeoffs [51], in this work we continue to explore colored Petri nets (CPNs) for the construction of TOPPER. CPN is a recent advancement in the field of Petri nets. It combines the capabilities of Petri nets with a high-level programming language, where Petri nets provide the primitives for process interaction and the programming language provides the primitives for the definition of data types and manipulations of data value. As we will present in §2, the use of CPN offers several benefits for our model building: (1) being more powerful than traditional Petri nets, (2) allowing fast model construction, (3) providing inherent validation capability, and (4) allowing users to capture complex system dynamics in an intuitive way.

We validate TOPPER by means of real system traces collected from the 10-petaflop machine named Mira at Argonne [36]. We conduct a series of experiments to examine the impact of various resilience methods, power-capping techniques and job characteristics on system-wide performance, such as of system utilization, job performance and energy consumption. We analyze the interactions and dependencies of different methods by answering the questions Q1-Q3 based on the Mira traces. To our best knowledge, TOPPER is the first of its kind for co-modelling all three factors (i.e., performance, power, and resilience) on petascale systems.

The rest of the paper is organized as follows. Section II presents background information. Section III describes model design. Section IV gives model validation. Section V presents experiments to answer the proposed questions. Section VI discusses the advantages as well as limitations of our method. Section VII shows related work. Finally, we present our conclusions in Section VIII.

2 Background

This section gives an overview of colored Petri Nets, including its basic formalization, its evolution, as well as its comparison with commonly-used modeling methods for large-scale systems.

2.1 Colored Petri Net

A Petri net is one of several mathematical modeling languages for the description of distributed systems. It is particularly well suited to systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, or stochastic. A Petri net consists of a set of *places* and *transitions* linked by *edges*, with *tokens* in some places. Places are used to represent system states, transitions are used to indicate system events, and edges are used to specify the relationships between system states and events. To indicate the

change of system state, tokens that reside in one place will move to another place according to the firing rules imposed by the transition [6].

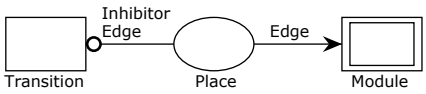

Fig. 1 Basic elements used in colored Petri nets.

Colored Petri Nets (CPNs) make several extensions from the traditional Petri Nets. First, CPN provides a better expressibility than Petri Nets by adding *colors* to tokens, thus enabling an accurate simulation of the real system. For instance, job-related attributes such as job arrival time, job size and job runtime can be described naturally by token colors. Second, CPN allows a *hierarchical* design, in which a module at a lower level can be represented by a transition at a higher level. Such an extension make it scalable to model large and complex systems. In addition, the availability of various well-developed Petri Nets simulation tools allow us to build models at a high level, making the model construction fairly convenient. Figure 1 summarizes the basic elements used in CPN. CPN has been widely used for modeling large-scale systems like biological networks [32]. To our knowledge, this is the first attempt to apply CPN to model power, performance and resilience in HPC.

2.2 Comparison with Other Methods

Commonly-used modeling methods include analytical modeling and trace-based simulation. In order to make predictive design decisions, application scientists and architects rely on a set of modeling and simulation tools, aiming to balance *speed*, *accuracy* and *flexibility*. *Speed* evaluates how fast the method can generate predictive results once the model is built. *Accuracy* measures the extent to which the prediction provided by the model agrees with the actual measurement from a real system. *Flexibility* indicates the ease of model generation, model modification, and model extension. Every technique has its strengths and weaknesses. Analytical models are often fast and flexible, but they can offer only limited accuracy. Trace-based simulators can have a highly-accurate representation of system behaviors, but extending a simulator for other functionalities is not trivial, which often requires a significant amount of engineering work.

CPN provides a viable approach to bridge the gap between analytical modeling and trace-based simulators. Compared with analytical modeling methods, CPN can provide better accuracy by capturing system dynamics naturally among different components. Compared with the conventional trace-based simulators, CPN makes model building much easier and faster by providing graphical representation and modeling language. Users can easily add, modify,

or remove system components or functionalities in an intuitive way. Moreover, CPN models are formal in the sense that the CPN modelling language has a mathematical definition of both its syntax and its semantics. Hence, a CPN model can be formally verified, e.g., to prove that certain desired properties are fulfilled or certain undesired properties are guaranteed to be avoided. This feature is critical for model validation.

3 Model Design

TOPPER consists of three interacting modules, namely, *batch scheduling, fault tolerance, and power management*. Figure 2 shows the top level design, in which three modules (double border boxes in blue) are connected through seven places (ellipses in green). *Batch Scheduling* module arranges jobs in the *Queuing* state according to a scheduling policy, and allows small jobs to skip ahead as long as they do not delay the job at the head of the queue. *Fault Tolerance* module depicts system failures and fault management operations such as checkpoint/restart and redundancy. It interacts with jobs in the *Running* state. *Power Management* module models two power-capping mechanisms: power-aware job scheduling and DVFS. The former provides a coarse-grained power cap by allocating queued jobs onto computer nodes based on the overall system power and the job power requirement, and the latter dynamically adjusts processors frequency to meet the power cap.

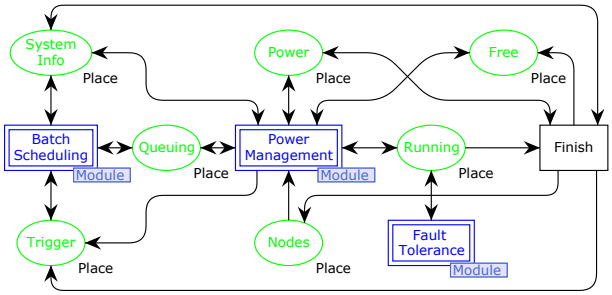

Fig. 2 The top level design of TOPPER. There are three hierarchical modules (denoted by double border boxes in blue) and seven states (denoted by ellipses in green).

Dynamic system states are modeled by seven places, each of which possesses different information. The *Queuing* state keeps a list of queued jobs, whose order can be changed dynamically by the *Batch Scheduling* module. The *Running* state holds jobs that are under execution, which interacts with the *Fault Tolerance* and the *Power Management* modules. The *Power* state indicates the current power level of the system, based on which the *Power Management* module is able to conduct power-capping. The *System Info* state provides the *Batch Scheduling* module with the runtime resource information and the *Trigger* state launches job reordering in the *Batch Scheduling* module. The *Nodes*

state represents the number of available compute units. The *Free* state receives jobs that have just finished execution for the release of system resources. We will give the details of the three modules in following subsections, using the inscriptions summarized in Table 1.

Table 1 Major inscriptions used in the TOPPER modules.

Inscriptions	Type	Description
job	variable	A job has eight attributes: js, rt, jp, ft, et, pf, [] and st.
jobs,ws	variable	A job list.
pow,old_pow	variable	System power level at the current and the previous time steps.
P1,...,Pm	variable	Processor power rates.
f,pf,new_f,f1,...,fm	variable	Processor frequency rates.
[expr]	symbol	A guard on a transition that is able to fire if expr evaluates to true.
@+ expr	symbol	A time delay specified by expr.

3.1 Batch Scheduling

On production HPC systems, batch scheduling is typically used for resource management and job scheduling. It includes a job waiting queue, a resource manager and a job scheduler. The waiting queue receives and stores jobs submitted by users. The resource manager is responsible for obtaining information about resource availability, waiting queue, and the running status of compute nodes. The job scheduler periodically examines the jobs in the waiting queue and the available resources via the resource manager, and determines the order in which jobs will be executed. The order is decided based on a set of attributes associated with jobs such as job arrival time, job size, the estimated runtime, etc.

A variety of scheduling policies have been developed for today's HPC systems [18]. Mira uses WFP-EASY, which employs a utility function to determine the order of jobs, and allows subsequent jobs to jump over the head job as long as they do not violate the reservation of the head job (so called EASY backfilling) [45]. It favors large and old jobs, adjusting their priorities based on the ratio of wait time to the requested wall-clock time. The utility function is defined as:

$$utility\_score = job\_size \times (\frac{job\_queued\_time}{job\_wall\_time})^3. \quad (1)$$

Figure 3 presents the module design for WFP with EASY backfilling. As shown in the figure, the jobs leave from the Users state according to their arrival times. Every job is described in the form of $(js, rt, jp, ft, et, pf, [])@+st$, where js is job size, rt is job runtime, jp is job power profile, st is job arrival time, ft stores the next failure time of the job, et records the time when the job enters a new place, pf indicates the frequency rate of the processor the job runs on, and [] is a list to keep the compute units that will be assigned

to the job. Here, js and rt are supplied by users, jp is an estimate that can be obtained from historical data [50], and all the others are maintained by TOPPER.

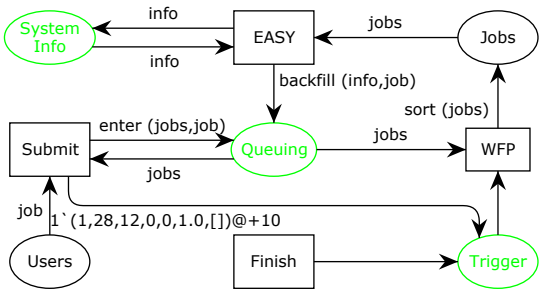

Fig. 3 Batch scheduling module for Mira [51].

In the module, the *Queuing* state accepts jobs submitted by the users. When a job enters or leaves the system, the *Trigger* state receives a signal and launches a job reordering process. The transition *WFP* fires and uses function *sort* to reorder the queued jobs according to the utility score given by equation 1. After that, transition *EASY* fires and uses the function *backfill* to "backfill" jobs according to the runtime resource information from the *System Info* state. In addition to WFP-EASY, we have also built a module for FCFS (First-come, first-served) with EASY backfilling, which is presented in our prior work [51].

3.2 Fault Tolerance

In the field of HPC, both checkpoint/restart and replication are well-known fault tolerant methods.

Checkpoint/Restart: Coordinated checkpointing is the widely-used method to mitigate the impact of failures [41]. It periodically coordinates the checkpoint across multiple application processes and stores a consistent snapshot of the application. In case of failure from any process, all the processes roll back to the last checkpoint for recovery. Coordinated checkpointing effectively reduces work loss at the cost of checkpointing and recovery overhead. A short checkpoint interval results in unnecessary checkpoints and thus higher overheads, but a long checkpoint interval leads to greater loss of computation due to failures. In this study, we use Daly's optimal checkpoint interval for job checkpointing [14]:

$$\delta_{opt} = \sqrt{\frac{2O_c}{\lambda_{app}}}[1 + \frac{1}{3}(\frac{O_c\lambda_{app}}{2})^{\frac{1}{2}} + \frac{1}{9}(\frac{O_c\lambda_{app}}{2})] - O_c, \quad (2)$$

where O_c is the overhead for one checkpoint operation, λ_{app} is the application failure rate which is the sum of failure rate of every node. Then given the application execution time rt, the checkpoint overhead for the whole application is calculated as:

$$O_{ckp} = \frac{rt}{\delta_{opt}} * O_c. \tag{3}$$

Replication: The state-of-the-art replication technique in HPC is commonly employed at the process level [20]. Unlike checkpoint/restart, replication mitigates the impact of failures by running multiple copies of every application simultaneously. If an application stops running due to some failures, a replica of the application can take over the computation, in which case an application fails only when all its replicas become non-functional. Thus, replication can increase the mean-time-to-failure (MTTF) of an application and allows less frequent checkpoints while maintaining the same resiliency level.

For a replication method, its overhead includes checkpoint overhead and overhead brought by the communications among replicas. Let rt be the application execution time, γ be the *communication/computation ratio* of the application and r be the redundancy degree (e.g., $r = 2$ for double replication and $r = 3$ for triple replication), the total overhead for the replication method is estimated as:

$$O_{rep} = O_{ckp} + rt * \gamma * (r - 1). \tag{4}$$

In real applications, the ratio γ vary from 10% to 70% and the average value is about 30% [12]. It is worth noting that to get O_{ckp} in equation 4, we must calculate λ_{app} in equation 2 in a different way because replication may decrease application failure rate. Here, we adopt the method used in [16], where the application failure rate with replication is expressed as a function of application execution time and redundancy degree.

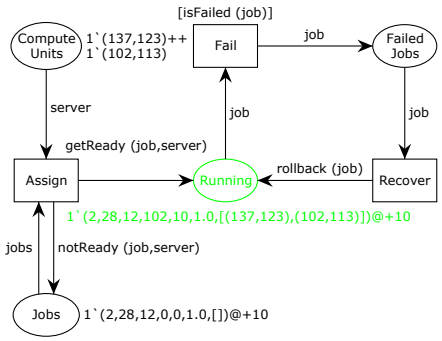

Fig. 4 Fault tolerance module.

Figure 4 presents the module design. Every compute unit is described in the form of (ft, mt), where ft indicates the next failure time of the compute unit

and mt specifies its MTTF. The failure time ft follows Poisson distribution and is updated according to mt once a failure occurs. Here, compute units can represent computing resources at different system levels such as rack, node and process, depending on the granularity of the modeling.

In the module, the transition *Assign* keeps allocating compute units to the job until its request is satisfied. When a job enters the *Running* state, its execution time rt is updated according to equation 3 or 4. It is assigned to a list of compute units and its failure time ft is set to the earliest failure time of all compute units. If the failure time of the job is earlier than its completion time, it will enter the *Failed Jobs* state, then the function *rollback* is used to recover the failed compute units and sets the job to the last checkpoint. The recovery overhead is also added by this function.

3.3 Power Management

Power-capping is used to control the peak power of a system within a cap. Current studies on power-capping in HPC can be classified as either software-based (e.g., power-aware job scheduling) or hardware-based (e.g., DVFS). In this work, we build both modules in TOPPER.

Power-aware job scheduling: For each job at the head of the queue, if its estimated power requirement makes the overall system power exceed the power cap, it is held in the queue and yields to jobs in the queue that are less power-hungry. Figure 5 presents the net design. We use *Pcap* to represent the power cap imposed on the system, w to indicate the maximum wait time of a job in the wait queue, and l to restrict the length of the wait queue. The module is centered around three transitions: *Allocate*, *Wait* and *Allocate W*. The head job in the *Queuing* state is either being allocated onto nodes or being held in the wait queue. The transition *Allocate* fires if the job's power requirement does not make the overall system power exceed the power cap; otherwise, the transition *Wait* fires as long as the wait queue is not full. Jobs in the wait queue can be allocated onto computer nodes by the firing of the transition *Allocate W*, which is set to a higher firing priority than the transition Allocate.

The parameter l is set to a positive integer. If the head job breaks the power cap, it will be moved to the *Wait Queue* state. Once the system power changes, the transition *Power Change* fires and starts a selection process for jobs in the wait queue. The function *pow-allocate* selects some jobs that can be allocated under the current system power and puts them at the head of the wait queue. However, if any job in the wait queue exceeds the maximum wait time (denoted by w), the job will be kept at the head and a signal will be sent to the *Block* state. The successors are blocked until the job at the head of the wait queue has been allocated.

DVFS: In the case that a job arrival makes the total system power exceed the power cap, DVFS adjusts the system to run at a lower power state, thus limiting the overall power within the cap. For a computing node in a system,

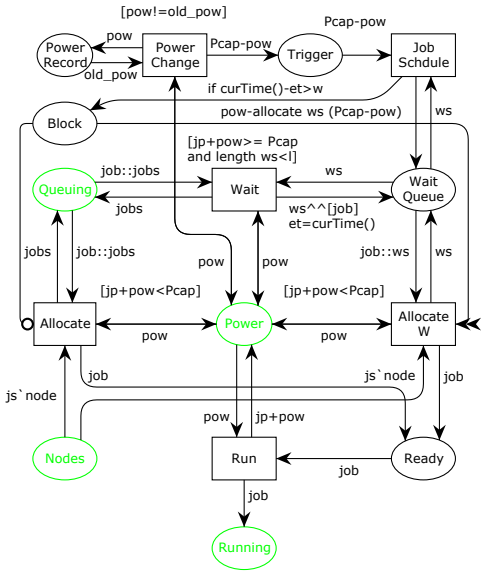

Fig. 5 Power-aware job scheduling module [51].

the power is mainly consumed by its CMOS circuits, which is captured by $P = V^2 * f * C_E$, where V is the supply voltage, f is the clock frequency and C_E is the effective switched capacitance of the circuits. According to different environments, the power consumption can be further approximated by $P \propto f^{\alpha}$. This indicates lowering the CPU speed can save power consumption. Meanwhile, lowering the CPU speed also decreases the maximum achievable clock speed, which leads to a longer time to complete an operation. Typically the time to finish an operation is inversely proportional to the clock frequency and can be represented as $t \propto \frac{1}{f^{\beta}}$.

Figure 6 presents the net design. When system power changes due to job arrival or leaving, the power indicated by the *New Power* state will be updated. According to this new system power and the power cap, one of the transitions in $\{T1, T2, ..., Tm\}$ fires, meaning that the processors are going to run at power rate P_i. Once T_i fires, the processor frequency rate is changed, and the remaining job execution time is modified by the function *adjust*. We assume there is no latency involved in DVFS. It is worth noting that while this work focuses on examining the ideal no-latency case, DVFS typically introduce a range of 1 us to 100 us latency [28]. It could impact system performance and scheduling, which will be part of our future work.

3.4 Tool Implementation

We implemented TOPPER using CPN Tools. CPN Tools is a popular tool for editing, simulating, and analyzing colored Petri nets, which has been in active

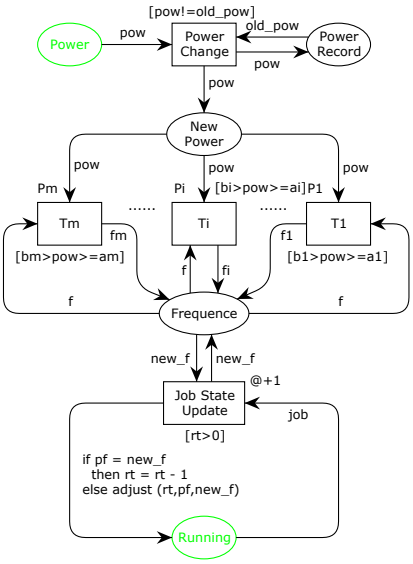

Fig. 6 DVFS module [51].

development since 1999 [2]. TOPPER is freely available [5], and is directly executable by a number of tools like CPN Tools, ExSpecT, and ReNeW [3, 4]. Currently, TOPPER accepts job traces in the standard workload format adopted by the community [1].

4 Model Validation

We validated TOPPER by means of real system logs collected from the production supercomputer *Mira* at Argonne Leadership Computing Facility. Mira contains 49,152 nodes and 789,432 cores, offering a peak performance of 10 petaflops. We collected three system logs, namely job trace, RAS log, and environmental log, from Mira between January and April of 2013. The details of these logs are presented in the next subsection. Job attributes such as job arrival time, job size, job execution time, and job power profile were extracted from the job log and the environmental log and fed to TOPPER. Real failure arrivals were also extracted from the RAS log and fed to TOPPER. Our model validation was performed by comparing TOPPER output with real output extracted from the system logs. Essentially, our validation intended to verify whether TOPPER is capable of truly simulating job arrival, job execution, and job completion (i.e., batch scheduling) under various resilience and power management mechanisms. Three evaluation metrics were used for model validation, including system utilization, average job wait time, and system-wide energy consumption. They are three critical factors to measure a supercom-

puter's performance and efficiency [51]. We expect TOPPER to provide a good approximation of all of them. Figure 7 presents the validation process.

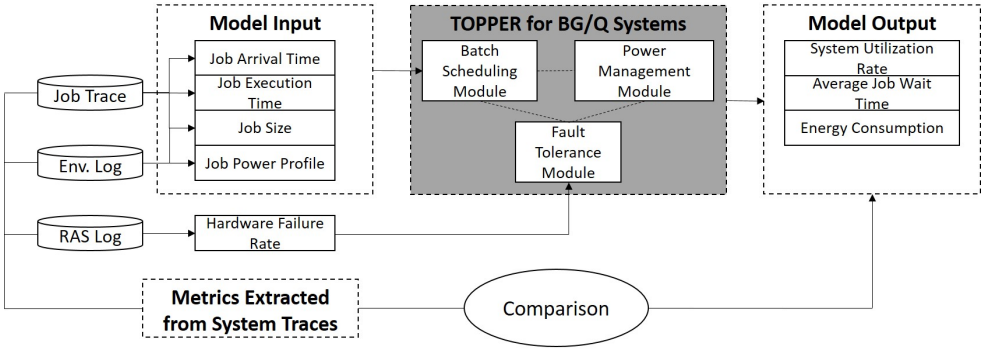

Fig. 7 Model validation workflow.

4.1 System Traces from Mira

Job Trace. Mira uses Cobalt [35] to accept job submissions from users and dispatch jobs to available compute nodes. Cobalt records these submissions, as well as start and end events in chronological order in an IBM DB2 database. Each record is basically composed of timestamp, event type, executable filename, job size, location, wall-time, etc. We collected a four-month job trace from Mira (January - April of 2013), which consisted of 16,044 jobs.
RAS Log. On Mira, the Core Monitoring and Control System (CMCS) is responsible for collecting the Reliability, Availability and Serviceability (RAS) events and storing them in a backend DB2 database. This information comes from hardware components, including compute nodes, I/O nodes and various networks, and are the primary source used by administrators to locate a failure when the system begins to malfunction. In this work, we used the RAS log to obtain failure information.
Environmental Log. On Mira, various sensors are deployed at different locations to gather environmental data (e.g., temperature, voltage and current, coolant flow, pressure, etc.) of system components. These data are stored in the DB2 database. In this work, we used the environmental log to retrieve system and job power information.

4.2 Model Accuracy

In this work, we measured job size in number of racks, job time in minutes, and job power profile in watts. We also rounded these measured numbers to the nearest integers for the purpose of modeling. The default model simulation

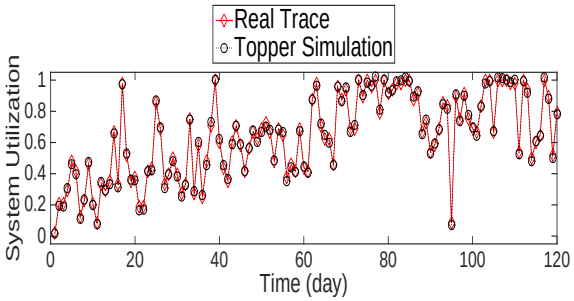

Fig. 8 Validation of system utilization rate.

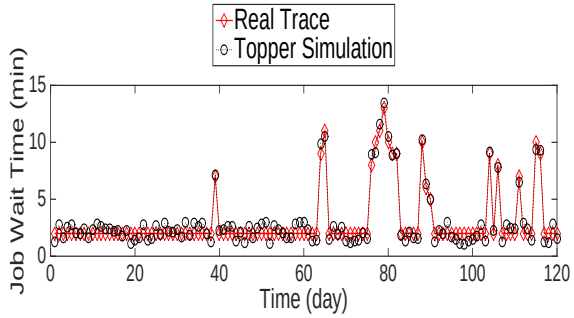

Fig. 9 Validation of average job wait time.

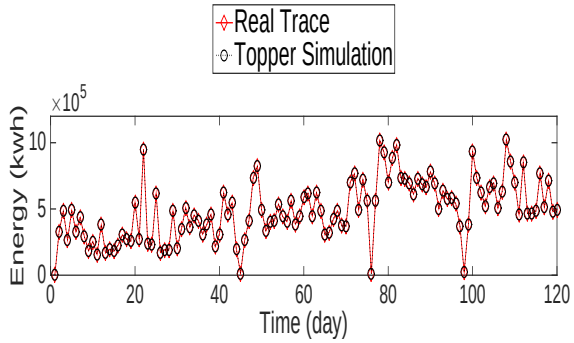

Fig. 10 Validation of total energy consumption.

time step is set to 1.0 minute. The energy consumption within the time step $[t, t+1]$ is approximated as the number of tokens in the place *Power* at time t. For example, if at the 5th minute the number of tokens in *Power* is 600, then the energy usage during the following time step is estimated as $600/1000 * 1/60 = 0.01kWh$.

With dynamic job arrival, execution, and completion, each with different power consumption, a good model should be able to simulate these dynamics,

thus providing utilization rate, job wait time and total energy consumption close to the real system behaviors. We compared the model output with the values extracted from the system logs. Figure 8-10 present the comparison results. The average error is less than 4% in terms of all three metrics. These results clearly show that TOPPER is able to truly reflect the dynamic job scheduling and allocation with high fidelity.

Moreover, in this study, we varied the simulation time step from 1 minute to 60 minutes, and assessed the maximum energy error obtained by TOPPER, along with the simulation overhead. All the experiments were conducted on a local PC that is equipped with an Intel Quad 2.83GHz processor and 8GB memory, running Windows 7 Professional 64-bit operating system. The results are shown in Table 2. TOPPER takes about 5 minutes to simulate the 4-month trace from Mira.

Table 2 Impact of simulation time step on model accuracy and simulation overhead.

Simulation Time Step (min)	Maximum Energy Error	Simulation Overhead
1	3.84%	5.40 min
5	5.03%	5.25 min
15	8.85%	4.95 min
30	15.32%	4.60 min
60	18.09%	4.40 min

5 Case Studies

In this section, we demonstrate the use of TOPPER for quantitative tradeoff analysis. The goal is to answer questions Q1-Q3 listed in §1. We used *System Efficiency* to evaluate the efficiency of the system, which is defined as the ratio of node hours doing useful work [49] to the total elapsed system node hours; *Average Job Wait Time* and *Makespan* to measure job scheduling effectiveness; and *Energy Consumption* to measure energy efficiency of the system.

Checkpoint/restart is typically influenced by its overhead and recover overhead. Checkpoint overhead O_c in large scale systems consists of two parts, namely I/O overhead and message passing overhead. Based on our experience and the current literature[33, 15], we set O_c to 600 seconds if the job used less than 16 racks; otherwise, we set O_c to 1200 seconds. In addition, we set recovery overhead to 780 seconds. These default settings will be adjusted in the experiments to study how different fault tolerant values will impact system performance and energy consumption under a given set of assumptions. Note that all the experiments were conducted and presented in a month by month manner. This allows us to analyze the impact of different workload characteristics.

Q1: How would different resilience methods (e.g., checkpoint/restart and replication) impact system-wide performance and energy consumption?

For this set of experiments, we compared checkpoint/restart and replication with the default configurations. For the replication method, the redundancy degree r was set to 2 and the application communication/computation ratio γ was set to 30%. The results are presented in Figure 11.

We made four observations from Figure 11. First, checkpoint/restart greatly outperformed replication by doubling system efficiency under different workload months. Second, with respect to average job wait time, checkpoint again outperformed replication, especially for the workloads with higher system utilization (i.e., in March and April). Third, both resilience methods led to similar scheduling makespan, and checkpoint slightly outperformed replication for the workloads with higher system utilization. Finally, with respect to system-wide energy consumption, checkpoint consumed less energy than replication.

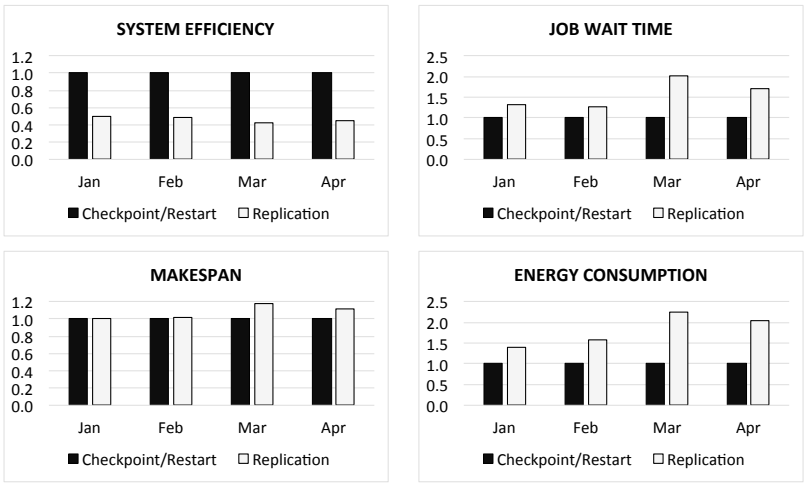

Fig. 11 Comparison of checkpoint/restart and replication on system performance and energy consumption with the default configurations. For replication, the redundancy degree r was set to 2, and the application communication/computation ratio γ was set to 30%. The results were normalized to checkpoint/restart.

The bad performance of replication was due to the fact that a replication method improves application resilience at the cost of maintaining extra copies of every application/job. The extra node hours consumed by replication not only cost extra energy, but also degrade system performance. Note that the extra node hours were not counted toward useful work. *An interesting observation is the benefits (i.e., scheduling performance and energy consumption) of using checkpoint over replication are especially striking when the system is under high workload demand.* In the case of a workload with a low system utilization rate, there is usually a long interval between job arrivals, thus extending a job execution hardly affects its successors. The contrary of this happens in a workload with a high system utilization rate. That is why the

performance difference between the two methods is more obvious for workloads with higher system utilization (i.e., in March and April).

Q2: How would different resilience parameter settings affect system performance and energy?

We conducted three sets of experiments to answer this question. In the first set, we varied checkpoint overhead O_c from 100% to 25% of the default value. A number of recent studies have presented various techniques to improve checkpoint by applying multi-level checkpoint or in-memory checkpoint [7]. In this study, we intended to investigate the performance gain that could be achieved by reducing checkpoint overhead.

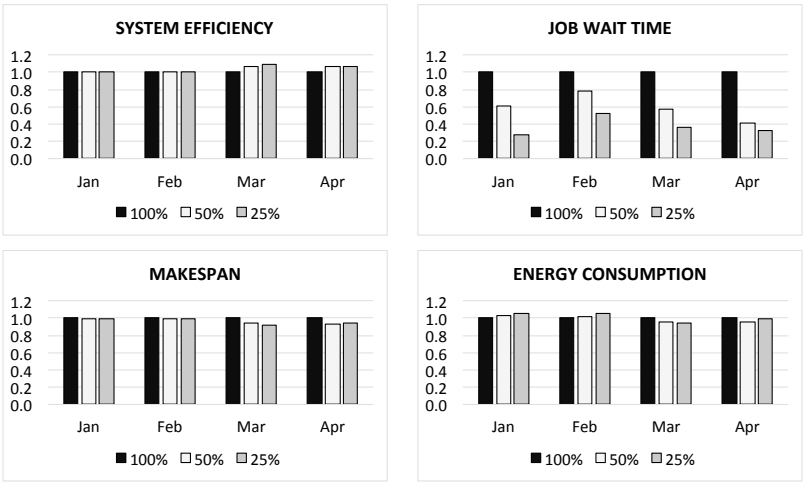

Fig. 12 Impact of different checkpoint overheads. The checkpoint overhead O_c was set to 100%, 50%, 25% of the default checkpoint overhead. The results were normalized to the default overhead.

The results are plotted in Figure 12. As we can see, *the reduction of checkpoint overhead greatly impact average job wait time, whereas it has trivial influence on the other scheduling metrics.* In general, the lower the checkpoint overhead is, the better the system-wide performance and energy efficiency are. In our studies, checkpoint overhead comprised a small fraction of the overall job execution time. As such, its impact on system efficiency, makespan and energy were not significant. We believe that if the checkpoint overhead has comprised a larger fraction of the overall job execution time (e.g., due to higher failure rate or inefficient checkpoint operation), the influence of checkpoint overhead would have been different. Interestingly, Figure 12 indicates that average job wait time, in contrast to other scheduling metrics, is very sensitive to checkpoint overhead.

In the second set of experiments we analyzed the performance impacts by varying redundancy degrees from 1.5 to 3. Here, the settings of 1.5 and 2.5 denote partial replication [20,16]. The results are presented in Figure 13.

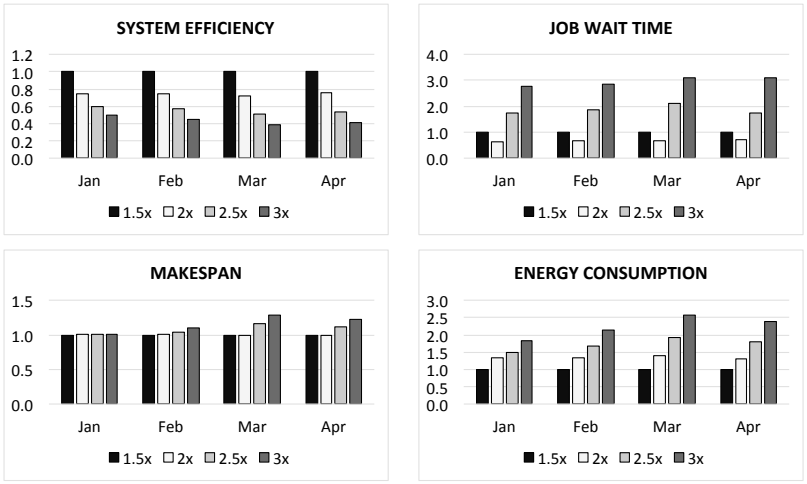

Fig. 13 Impact of different redundancy degrees. The redundancy degree r was set to 1.5, 2, 2.5 and 3. The results were normalized to a redundancy degree of 1.5.

The impact of redundancy degrees on job execution time is two-fold. A lower redundancy degree introduces less overhead but leads to a higher application failure rate, which in turn results in more overhead for failure recovery. Hence, a lower redundancy degree could introduce more overhead to job execution than a higher redundancy degree. While a lower redundancy degree may be beneficial in terms of system efficiency, makespan, and energy consumption, Figure 13 clearly points out that double replication provides the shortest job wait time. In other words, *a lower redundancy degree doesn't necessarily mean a better scheduling performance.*

For any replication-based method, application communication/computation ratio can greatly impact its performance [12]. Thus, in the third set of experiments, we varied the application communication/computation ratio from 10% to 70%, and examined how these changes would influence system performance and energy consumption. The results are presented in Figure 14.

Clearly, application communication/computation ratio had a great effect on all four system metrics, and the impact was higher for the workloads with higher system utilization rates (i.e., in March and April). By comparing Figures 13 and 14, we found that *application communication/computation ratio has significant impact on average job wait time and makespan, but less obvious impact on system efficiency and energy consumption.* This is because for job wait time and makespan, the dominant factor is the overhead added to jobs, while for system efficiency and energy consumption, the dominant factor is the number of replicas.

Another observation is that r*eplication may lead to better job performance than checkpoint/restart for applications with small communication/computation ratios (i.e., less than* 20%*).* By comparing Figures 11 and 14, we found that replication resulted in shorter job wait time than checkpoint, in cases that

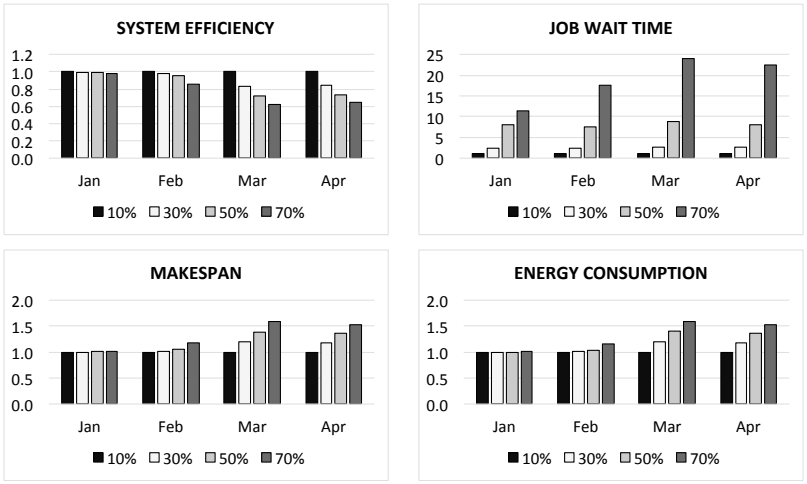

Fig. 14 Impact of communication/computation ratio on replication. The communication/computation ratio γ was set to 10%, 30%, 50% and 70%. The results were normalized to ratio of 10%.

the communication/computation ratio was less than 20%. Compared to the 30% communication/computation ratio used in Figure 11, when this ratio became 10%, the job wait time was greatly decreased because of the reduction of communication overhead.

Employing DVFS for power-capping may cause a 3 times higher system failure rate [44], whereas the power-aware job scheduling method is a software-based approach and does not affect component reliability. In this set of experiments, we sought to understand both power management mechanisms by considering their potential reliability effects. More specifically, we varied the failure rate from 1 to 3 times of the original failure rate for the use of D-VFS. Since power-aware job scheduling does not impact system reliability, no change was made to this method. In our experiments, the power cap was set to $45,720$ kw (70% of the maximum power in January). The frequency-to-power relationship α was set to 2, and the frequency-to-time relationship β was set to 1 (described in §3.3) [31]. The idle processor power was set to be 30% of the power at the full speed. The results are presented in Figure 15.

In general, power-aware job scheduling and DVFS-double (i.e., DVFS caused two times the failure rate) are comparable in terms of system efficiency, job wait time, and scheduling makespan. Second, if DVFS causes triple failure rate, the use of DVFS can significantly impact scheduling performance such as job wait time and makespan.

Power-aware job scheduling and DVFS use different mechanisms to keep the peak power consumption within a limit. By adjusting CPU frequency, DVFS potentially increases system utilization and decreases the energy consumption. On the contrary, power-aware job scheduling might have to delay job execution to limit the power consumption, thus resulting in lower system

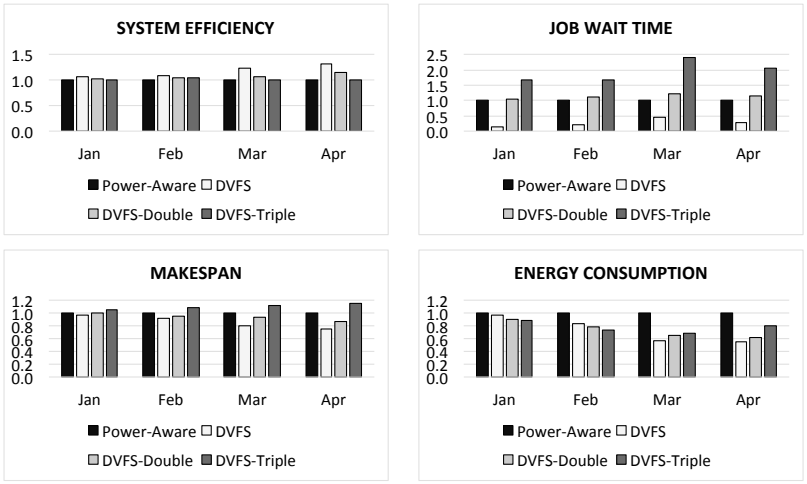

Fig. 15 Comparison of power-aware job scheduling and DVFS on system performance and energy consumption by taking into account the impact of DVFS on system reliability. For DVFS, the failure rate was set to 1, 2, 3 times that of the original failure rate, denoted by "DVFS", "DVFS-double" an "DVFS-triple" respectively. The power cap was set to 70% of the maximum power in January. The results were normalized to power-aware job scheduling method with the original failure rate.

utilization. Note that the use of power-aware job scheduling did not change the overall energy requirement for a workload. Since DVFS may increase the hardware failure rate, the use of DVFS can cause more frequent failures, thus more overhead for failure recovery.

As shown above, job characteristics could have a significant impact on system performance and energy consumption. Here, we further investigated the potential impacts of different job runtimes and job sizes. Specifically, under the same system configuration with 49,152 nodes, we varied job wall time, ranging between 50% to 400%, base on the original workload where the job wall time is denoted by "Wall Time * 1.0". Similarly, we varied job size (i.e., number of requested compute nodes), ranging between 50% to 400%, base on the original workload where the job size is denoted by "Job Size * 1.0". The main goal is to study what would be the possible performance impact and energy consumption if future workloads have different job sizes and runtimes. The results are presented in Figure 16 and 17.

Figure 16 presents several interesting results. First, a workload with shortened jobs greatly reduces system efficiency whereas a workload with prolonged jobs increases system efficiency slightly. This is because a workload with shortened jobs leads to more idle node-hours. Second, varying job wall time affects average wait time slightly when job arrival rate is low (Jan and Feb), but greatly impacts when job arrival rate is high (Mar and Apr). When job arrival rate is low, the change of job runtimes rarely affects the execution of the subsequent jobs. Third, the change of job runtime does not induce significant change on system makespan. A possible reason is that system makespan is

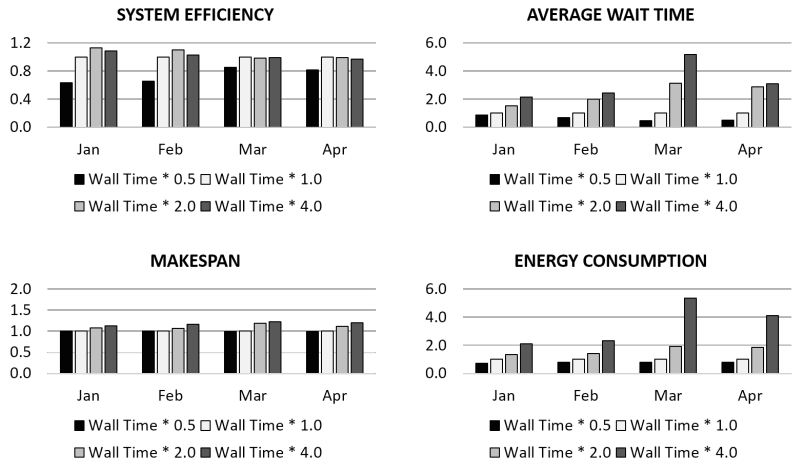

Fig. 16 The impact of job runtime on system performance and energy consumption. The results were normalized to Wall Time * 1.0.

influenced by both job arrival rate and job characteristics. Under the same job arrival rate, when a job wall time is varied, it only affects the execution of a set of subsequent jobs and has little impact on the aggregated performance of the entire workload with tens of thousands of jobs. Finally, we observed that job runtime has a significant influence on energy consumption because it affects both system efficiency and makespan.

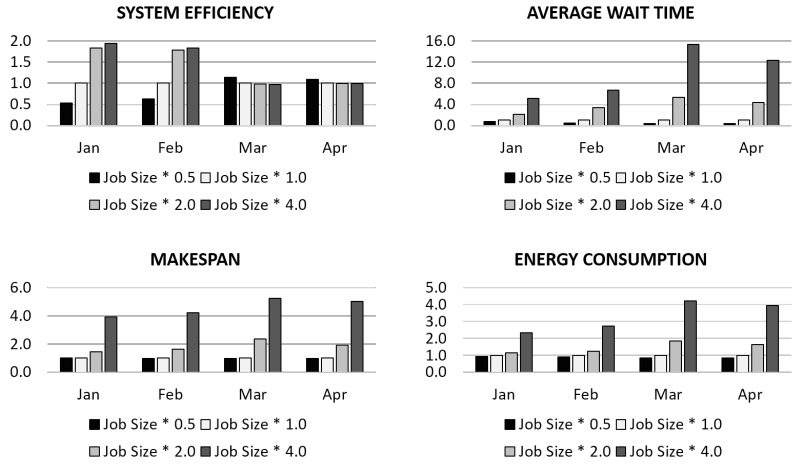

Fig. 17 The impact of job size on system performance and energy consumption. The results were normalized to Job Size * 1.0.

Figure 17 illustrates the impact caused by job size (i.e., number of nodes used by each user job). A smaller job size leads to a significant decrease of system efficiency under low job arrival rate (Jan and Feb); while it slightly increases system efficiency under high job rate (Mar and Apr). This makes sense because smaller jobs could result in a larger number of unoccupied compute nodes during job execution when job arrival rate is high, and this effect becomes less significant under low job arrival rate because the idle nodes can be occupied by subsequent jobs quickly. For a similar reason, we have opposite observations for larger job size, i.e., it greatly increases of system efficiency under low job arrival rate and slightly decreases system efficiency under high job arrival rate. In addition, the smaller job size always indicates lower average job wait time, shorter makespan, lower energy consumption; and vice versa. For the same reason mentioned above, the impact of job size become less significant when job arrival rate is low but become more obvious when job arrival rate is high.

6 Discussion

As mentioned earlier, TOPPER was designed for system-wide tradeoff analysis among power, performance and resilience. Specifically, it models the performance and energy consumption of a batch of user jobs, i.e., at the scheduler level. It can be used to analyze a petascale system under various job scheduling, power management and resilience configurations. In the current version of TOPPER, the batch scheduling module supports both FCFS-EASY and WFP-EASY job scheduling policies [45]; the power management module supports both power-aware job reallocation and DVFS power-capping methods; and the fault tolerance module supports both checkpoint/restart and replication resilience mechanisms. All these scheduling, power management and fault tolerance strategies have been widely used in today's petascale systems. The benefits of TOPPER include easy model building, naturally capturing complex systems dynamics, good extensibility and high scalability.

TOPPER was built on CPNs, which comes with solid mathematical foundation and has proven its efficiency and effectiveness of modeling complex large-scale systems [27]. As TOPPER provides a set of inscriptions (Table 1) that can generally describe a HPC system with the standard format adopted by the community, it is easy for users to tune its parameters for other systems with different scales of workload and resource. Also, with the nature of CPNs, TOPPER is able to model both the determinism and indeterminism in a system. In this study, the batch scheduling module models the job execution in a deterministic way as it follows the job trace dumped from the production system; while the fault tolerance module models the failure presence in a nondeterministic way by assuming failure arrival follows the poisson distribution. This provides a great flexibility for us to model the randomness from the perspective of real-world systems. For example, for systems without real

job trace, the job arrival can be modeled similarly as in the fault tolerance module in this study.

7 Related Work

Modern systems are designed with various hardware sensors that collect power-related data and store these data for system analysis. System level tools like LLView [37] and PowerPack [23] are developed to integrate the power monitoring capabilities to facilitate systematic measurement, modeling, and prediction of performance. Goiri et al. used external meters to measure the power consumption of a node during its running time [25]. Feng et al. presented a general framework for direct, automatic profiling of power consumption on high-performance distributed systems [19]. In our recent work, we developed a power profiling library called *MonEQ* for accessing internal power sensor readings on IBM Blue Gene/Q systems [48].

Co-modeling power and performance has been done on several systems. Analytical modeling is a commonly used method, which mainly focuses on building mathematical correlations between power and performance metrics of the system. Chen et al. proposed a system-level power model for online estimation of power consumption using linear regression [10]. Curtis-Maury et al. presented a online performance prediction framework to address the problem of simultaneous runtime optimization of DVFS and DCT on multi-core systems [13]. Tiwari et al. developed CPU and DIMM power and energy models of three widely-used HPC kernels by training artificial neural networks [47].

There also exist studies of applying stochastic models for performance or power analysis. B.Guenter et al. adopted a Markov model for idleness prediction and also proposed power state transitions to remove idle servers [26]. Qiu et al. introduced a continuous-time and controllable Markov process model of a power-managed system [39]. Rong et al. presented a stochastic model for a power-managed, battery-powered electronic system, and formulated a policy optimization problem to maximize the capacity utilization of the battery powered systems [42]. The studies closely related to ours are [21, 46]. Gandhi et al. used queueing theory to obtain the optimal energy-performance tradeoff in server farms [21], and Tian et al. proposed a model using stochastic reward nets (SRN) to analyze the performance and power consumption under different power states [46].

Our work differs from the existing studies in two ways. First, to our knowledge, our work is the first study of co-modeling performance, power, and resilience on an extreme-scale HPC system. Second, unlike the existing modeling studies using analytical or queue models, our work is built on colored Petri net, which overcomes a number of limitations of analytical modeling and queuing theory.

8 Conclusion

In this paper, we have presented TOPPER, a novel co-modeling mechanism for tradeoff analysis of performance, power, and resilience on HPC. Using the advanced features of colored Petri nets, TOPPER is capable of capturing the complicated interactions among execution time, energy efficiency, and resilience. Using real system traces collected from the 48-rack IBM Blue Gene/Q machine at Argonne, we have explored the use of TOPPER to study system performance and energy efficiency under different resilience and power management methods.

Our experiments provide a number of interesting observations. First, while some studies have shown that replication-based methods can tradeoff additional resource requirements against wall clock time [16], our study indicates that replication is incomparable to checkpoint in terms of system-wide performance and energy usage on HPC systems. In particular, checkpoint greatly outperforms replication in the system under high workload. Second, a lower redundancy degree doesn't necessarily mean a better scheduling performance. For instance, on Mira, double replication actually provides the shortest job wait time. Finally, without considering DVFS latency, power-aware scheduling is comparable to DVFS with two times higher failure rate in terms of scheduling performance. If DVFS causes triple failure rate, the use of DVFS can significantly impact scheduling performance such as job wait time and makespan, hence is not recommended to use. While this work focuses on examining the ideal case of DVFS, that is, DVFS without latency, DVFS with latency could impact system performance and scheduling, which will be part of our future work.

CPNs is a powerful technology to model large-scale systems that involve complex interactions among components. There are many CPNs tools under active development, which allow users from different areas to edit, simulate, and analyze CPNs in a convenient way. We make TOPPER freely available to the community [5] and we believe it has many other potential usages in addition to the case studies presented in this work. There are many directions for our future study. One of them is to use TOPPER for a variety of scalability study such as energy efficiency analysis by scaling up system size.

Acknowledgments

This work is supported in part by US National Science Foundation grant CCF-1618776 and CCF-1422009. It used data of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

1. The Standard Workload Format. Available at: `http://www.cs.huji.ac.il/labs/parallel/workload/swf.html`, 2007.
2. CPN Tools. Available at: `http://cpntools.org/`, 2015.
3. ExSpecT. Available at: `http://www.exspect.com/`, 2015.
4. ReNeW. Available at: `http://www.renew.de/`, 2015.
5. TOPPER. Available at: `http://bluesky.cs.iit.edu/topper/`, 2015.
6. G. Balbo. Introduction to Generalized Stochastic Petri Nets. In *Proc. of SFM*, 2007.
7. L. Bautista-Gomez, D. Komatitsch, N. Maruyama, S. Tsuboi, F. Cappello, and S. Matsuoka. FTI: High Performance Fault Tolerance Interface for Hybrid Systems. In *Proc. of SC*, 2011.
8. W. Bircher and L. John. Analysis of Dynamic Power Management on Multi-Core Processors. In *Proc. of ICS*, 2008.
9. D. Bodas, J. Song, M. Rajappa, and A. Hoffman. Simple Power-aware Scheduler to Limit Power Consumption by HPC System Within a Budget. In *Proc. of E2SC*, 2014.
10. X. Chen, C. Xu, R. Dick, and Z. Mao. Performance and Power Modeling in a Multi-Programmed Multi-Core Environment. In *Proc. of DAC*, 2010.
11. M. Chiesi, L. Vanzolini, C. Mucci, E. Scarselli, and R. Guerrieri. Power-Aware Job Scheduling on Heterogeneous Multicore Architectures. *IEEE Trans. Parallel Distrib. Syst.*, 26:868–877, 2015.
12. M. Crovella, R. Bianchini, T. Leblanc, E. Markatos, and R. Wisniewski. Using Communication-to-Computation Ratio in Parallel Program Design and Performance Prediction. In *Proc. of IPDPS*, 1992.
13. M. Curtis-Maury, J. Dzierwa, C. Antonopoulos, and D. Nikolopoulos. Online Power-performance Adaptation of Multithreaded Programs Using Hardware Event-based Prediction. In *Proc. of ICS*, 2006.
14. J. Daly. A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. *Future Generation Computer Systems*, 22:303–312, 2006.
15. S. Di, M.-S. Bouguerra, L.A. Bautista-Gomez, and F. Cappello. Optimization of Multi-level Checkpoint Model for Large Scale HPC Applications. In *Proc. of IPDPS*, 2014.
16. J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. Combining Partial Redundancy and Checkpointing for HPC. In *Proc. of ICDCS*, 2012.
17. X. Fan, W.-D. Weber, and L. Barroso. Power Provisioning for a Warehouse-sized Computer. In *Proc. of ISCA*, 2007.
18. D. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, and P. Wong. Theory and Practice in Parallel Job Scheduling. In *Proc. of JSSPP*, 1997.
19. X. Feng, R. Ge, and K. Cameron. Power and Energy Profiling of Scientific Applications on Distributed Systems. In *Proc. of IPDPS*, 2005.
20. K. Ferreira, J. Stearley, J. Laros III, R. Oldfield, and et al. Evaluating the Viability of Process Replication Reliability for Exascale Systems. In *Proc. of SC*, 2011.
21. A. Gandhi, M. Harchol-Balter, and I. Adan. Server Farms with Setup Costs. *Perform. Eval.*, 67:1123–1138, 2010.
22. R. Ge, X. Feng, and K. Cameron. Performance-constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters. In *Proc. of SC*, 2005.
23. R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. Cameron. PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications. *IEEE Trans. Parallel Distrib. Syst.*, 21:658–671, 2010.
24. C. Gniady, A. Butt, Y. Hu, and Y.-H. Lu. Program Counter-based Prediction Techniques for Dynamic Power Management. *IEEE Trans. Comput.*, 55:641–658, 2006.
25. I. Goiri, L. Kien, M. Haque, R. Beauchea, T. Nguyen, J. Guitart, J. Torres, and R. Bianchini. GreenSlot: Scheduling Energy Consumption in Green Datacenters. In *Proc. of SC*, 2011.
26. B. Guenter, N. Jain, and C. Williams. Managing Cost, Performance, and Reliability Tradeoffs for Energy-Aware Server Provisioning. In *Proc. of INFOCOM*, 2011.
27. K. Jensen. Colored Petri Nets and the Invariant-method. *Theoretical Computer Science*, 14:317–336, 1981.

28. S. Kanev, K.M. Hazelwood, G.-Y. Wei, and D.M. Brooks. Tradeoffs between Power Management and Tail Latency in Warehouse-Scale Applications. In *Proc. of IISWC*, 2014.
29. T. LeBlanc, R. Anand, E. Gabriel, and J. Subhlok. Volpexmpi: an MPI Library for Execution of Parallel Applications on Volatile Nodes. In *European PVM/MPI Users' Group Meeting*, 2009.
30. C. Lefurgy, X. Wang, and M. Ware. Server-Level Power Control. In *Proc. of ICAC*, 2007.
31. T. Martin and D. Siewiorek. Non-Ideal Battery and Main Memory Effects on CPU Speed-Setting for Low Power. *IEEE Trans. VLSI System*, 9:29–34, 2001.
32. W. Marwan, C. Rohr, and M. Heiner. *Petri Nets in Snoopy: A Unifying Framework for the Graphical Display, Computational Modelling, and Simulation of Bacterial Regulatory Networks*. Humana Press, 2012.
33. A. Moody, G. Bronevetsky, K. Mohror, and B. Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proc. of SC*, 2010.
34. NSF Cyberinfrastructure Framework for 21^{st} Century Science and Engineering Vision. Available at: `http://www.nsf.gov/pubs/2010/nsf10015/nsf10015.jsp`.
35. Cobalt Resoure Manager. Available at: `http://trac.mcs.anl.gov/projects/cobalt`.
36. Mira: Next-generation Supercomputer. Available at: `https://www.alcf.anl.gov/mira`, 2012.
37. LLview: Graphical Monitoring of LoadLeveler Controlled Cluster. Available at: `http://www.fz-juelich.de/jsc/llview/`, 2013.
38. T. Patki, D. Lowenthal, B. Rountree, M. Schulz, and B. de Supinski. Exploring Hardware Overprovisioning in Power-constrained, High Performance Computing. In *Proc. of ICS*, 2013.
39. Q. Qiu and M. Pedram. Dynamic Power Management based on Continuous-Time Markov Decision Processes. In *Proc. of DAC*, 1999.
40. D. Reed, C. Lu, and C. Mendes. Big Systems and Big Reliability Challenges. In *Proc. of ParCo*, 2003.
41. R. Riesen, K. Ferreira, D. Silva, P. Lemarinier, D. Arnold, and P. Bridges. Alleviating Scalability Issues of Checkpointing Protocols. In *Proc. of SC*, 2012.
42. P. Rong and M. Pedram. Battery-Aware Power Management Based on Markovian Decision Processes. In *Proc. of ICCAD*, 2006.
43. J. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang. Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance. In *Proc. of IPDPS*, 2005.
44. J. Srinivasan, S. Adve, P. Bose, and J. Rivers. The Impact of Technology Scaling on Lifetime Reliability. In *Proc. of DSN*, 2004.
45. W. Tang, N. Desai, D. Buettner, and Z. Lan. Analyzing and Adjusting User Runtime Estimates to Improve Job Scheduling on Blue Gene/P. In *Proc. of IPDPS*, 2010.
46. Y. Tian, C. Lin, and M. Yao. Modeling and Analyzing Power Management Policies in Server Farms using Stochastic Petri Nets. In *Proc. of e-Energy*, 2012.
47. A. Tiwari, M. Laurenzano, L. Carrington, and A. Snavely. Modeling Power and Energy Usage of HPC Kernels. In *Proc. of IPDPSW*, 2012.
48. S. Wallace, V. Vishwanath, S. Coghlan, Z. Lan, and M. Papka. Application Profilling Benchmarks on IBM Blue Gene/Q. In *Proc. of Cluster*, 2013.
49. J. Wingstrom. Overcoming The Difficulties Created By The Volatile Nature Of Desktop Grids Through Understanding. Technical report, Ph.D. thesis, University of Hawai'i at Manoa, 2009.
50. X. Yang, Z. Zhou, S. Wallace, Z. Lan, W. Tang, S. Coghlan, and M. Papka. Integrating Dynamic Pricing of Electricity into Energy Aware Scheduling for HPC Systems. In *Proc. of SC*, 2013.
51. L. Yu, Z. Zhou, S. Wallace, M. Papka, and Z. Lan. Quantitative Modeling of Power-Performance Tradeoffs on Extreme Scale Systems. *Journal of Parallel and Distributed Computinged Computing*, 84:1–14, 2015.
52. Z. Zhou, Z. Lan, W. Tang, and N. Desai. Reducing Energy Costs for IBM Blue Gene/P via Power-Aware Job Scheduling. In *Proc. of JSSPP*, 2013.