Effective Cost Reduction for Elastic Clouds under Spot Instance Pricing Through Adaptive Checkpointing

Itthichok Jangjaimon, *Student Member*, *IEEE*, and Nian-Feng Tzeng, *Fellow*, *IEEE*

Abstract—Cloud computing users are most concerned about the application turnaround time and the monetary cost involved. For lower monetary costs, less expensive services, like spot instances offered by Amazon, are often made available, albeit to their relatively frequent resource unavailability that leads to on-going execution being evicted, thereby undercutting execution performance. Meanwhile, multithreaded applications may take advantage of elastic resource availability and cost fluctuation inherent to the systems. However, their potential gains on utilizing spot instances would be contingent upon how they handle resource unavailability, calling for an effective checkpointing. This work presents design and implementation of our enhanced adaptive incremental checkpointing (EAIC) for multithreaded applications on the RaaS clouds under spot instance pricing. EAIC model takes into account spot instance revocation events, besides hardware failures, for fast and accurately predicting the desirable points of time to take checkpoints so as to markedly reduce the expected job turnaround time and the monetary cost. The experimental results from our established test bed on PARSEC benchmarks under real spot instance price traces from Amazon EC2 show that EAIC lowers both the application turnaround time and the monetary cost markedly (by up to 58% and 59%, respectively) in comparison to its recent checkpointing counterpart.

Index Terms—Adaptive checkpointing, delta compression, fault tolerance, incremental checkpointing, Markov models, RaaS clouds, spot instances

1 Introduction

Cloud service providers rent out their computing resources to clients on-demand as needed, making elastic computing resources available to the public. With economies of scale, cloud vendors sell computing resources at competing prices for different levels. Amazon's EC2 cloud, for example, offers (1) guaranteed compute resources always ready to accommodate job execution under reserved instance pricing, (2) on-demand compute resources, which may delay launching job execution until they are available but on-going execution will not be aborted, under on-demand pricing, and (3) bidden compute resources, which abort execution upon price hikes beyond the bidden amount, under spot instance pricing. Resources acquired under reserved instance (RI) pricing have highest availability, while those under spot instance (SI) pricing are least expensive, albeit to reduced availability. While SI pricing gives clients with chances to rent more compute resources, checkpointing is highly desired for such compute resources to boost overall execution performance and productivity [23], due to their relatively frequent resource unavailability resulting from revocation, as explored lately. Checkpointing saves the states of program execution to let it resume execution from the last checkpoint upon hardware failures or SI revocation.

Cloud systems have been envisioned recently to evolve toward the Resource-as-a-Service (RaaS) paradigm, where physical resources (e.g., processors/cores, memory, I/O and network bandwidth) are rented out separately for a fine-grained period (e.g., in the order of seconds) each time [4]. The future RaaS cloud differs from current IaaS (Infrastructure-as-a-Service) clouds, which sell bundles of compute resources as server-equivalent virtual machines for long durations (ranging from 5 minutes to hours) at a time. It offers clients with more flexibility in selecting and packing hardware resources to better fit job execution needs, permitting the client to acquire a networked multicore system with the specified number of nodes and designated link bandwidth among them. Users freely setup their software stack, as opposed to more restricted models like PaaS (Platform-as-a-Service) and SaaS (Software-as-a-Service).

While multithreaded applications can better utilize hardware resources present in a networked multicore system by letting application threads run on available cores concurrently, it has been observed that such an application run is more vulnerable to hardware failures [9], [15] since a single failure occurred at any execution thread terminates the whole program. This prompts an urgent need for effective checkpointing, and the checkpointing benefits are expected to rise on future RaaS clouds [4] with short rental periods, which result in highly dynamic resource (un)availability.

This work introduces design and implementation of enhanced adaptive incremental checkpointing (**EAIC**), an effective checkpointing support for multithreaded programs run on such a networked multicore system (NMS) acquired from the RaaS cloud under SI pricing for a lower cost, subject to

• *The authors are with Center for Advanced Computer Studies, the University of Louisiana, Lafayette, LA 70503.*
E-mail: {ixj0704, tzeng}@cacs.louisiana.edu.

Manuscript received 25 June 2013; revised 06 Nov. 2013; accepted 13 Nov. 2013. Date of publication 02 Dec. 2013; date of current version 16 Jan. 2015. Recommended for acceptance by K. Li.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.225

more resource unavailability. To lower the program turnaround time and the monetary cost, EAIC relies on our developed Markov model, which takes into account SI revocation events, besides hardware failures, for fast prediction with high accuracy on the desirable points of time for taking checkpoints. Our model permits to quantify the costs and expected turnaround times of multithreaded job execution under a wide range of possible price and resource unavailability ratios. EAIC leverages on the unique characteristics of SI in RaaS clouds to arrive at effective incremental checkpointing in multithreaded applications without kernel patching for high portability. Experimental results are obtained from our established testbed, which resembles an NMS rented from the RaaS cloud, revealing that EAIC yields lower application turnaround times, smallest aggregate file sizes, and less monetary costs than its prior counterparts. For PARSEC benchmarks under 8-month real spot instance price traces from Amazon EC2, EAIC is observed to shorten the application turnaround time by up to 58% and to lower the monetary cost by up to 59%, able to exhibit outcomes mostly close to those of the ideal clairvoyant mechanism.

2 Related Work

Checkpointing is aimed originally at fault tolerance for long-running high performance computing (HPC) jobs [9], [15], [18], [20], [21]. It is particularly useful for multithreaded and MPI applications as they involve more hardware components, thus subject to higher failure rates. Lately, checkpointing has been shown to help reduce costs and expected runtimes on cloud systems under SI pricing [23], adopting a model based on the probability density function of the revocation events. However, as will be seen in Section 3.1, the model in [23] fails to take into account the duration of a revocation event, assuming unrealistically that the restart process begins right after the revocation and hence unable to be applied practically. On the other hand, our new adjusted Markov model takes into account the revocation duration and yields faster prediction with higher accuracy while involving low complexity of O(1) in space and time. In general, our model enables conventional fault-tolerant middlewares (which rely on failure detection followed by task re-execution, also common for MapReduce) to handle SI revocations effectively without excessive task re-execution.

Hardware failure in large-scale systems was examined during 1996-2005 to arrive at the conjecture that a failure model with exponential distribution (like the Markov model) was not the best [16]. Recent studies (during 2008-2012), however, report high accuracy of the Markov model [9], [15]. We adopt the Markov model in this work, as it allows for fast prediction with sufficient accuracy, deemed crucial for our online checkpointing decision.

Checkpoint files may be taken at the program level [21] or the system level (e.g., virtual machine snapshots [10], [17]), and they can be transferred to remote storage in a networked system to tolerate local failures, leading to *multi-level checkpointing* [9], [15], [18]. Given expensive I/O operations to remote storage, multi-level checkpointing calls for size reduction, via such techniques as *incremental checkpointing* [6], [10], [17], [19], [20] and *delta compression* [13]. Incremental checkpointing saves only those memory pages (or VM data chunks [10], [17]) that are modified over the time interval after the immediate prior checkpoint. Delta compression further reduces checkpoint file sizes by differencing file contents. Asynchronous checkpointing [15] flushes checkpoint files to persistent storage concurrently with the process execution. Adaptive checkpointing [21] takes checkpoints at desirable points of time.

Aiming to reduce the application expected turnaround time, adaptive incremental checkpointing (AIC) was proposed earlier [8] for *single-threaded* programs in the networked multicore system (NMS). AIC incorporates aforementioned techniques, with such unique features as employing more aggressive compression specialized for checkpoint files and exploiting latency variations caused by delta compression to lower checkpointing overhead. In contrast, our EAIC employs the AIC framework for *multithreaded* programs under the *SI RaaS cloud*, exhibiting distinct system characteristics. First, unavailable events under spot instances have different behavior from hardware failures in NMS, calling for a new model. Second, an additional metric (monetary costs) is also of major concern, besides the file sizes and execution times. Third, while employing idle CPU cores for checkpointing is beneficial under NMS (as demonstrated recently [8]), there is usually no unused virtual cores under the cloud environment, given that the overall cost is proportionally to the total number of virtual cores leased. Fortunately, our EAIC is so light-weighted that its checkpointing process can be accommodated by involved virtual cores meant for task execution without noticeable performance degradation. Forth, our incremental checkpointing for multithreaded applications without kernel patching (for high portability) exhibits unique challenges, calling for new technical solutions(in sharp contrast to prior work found in [6], [10], [17], [19], [20]). In Section 5, the experimental results for RaaS clouds confirm that our new techniques lead to good performance. Our EAIC will also benefit other cloud paradigms(IaaS, PaaS, and SaaS) facing frequent resource revocation (e.g., IaaS under SI pricing [1]). EAIC concurrent multi-level checkpointing is related to the asynchronous model developed recently by Sato et al. [15], with two main differences. First, EAIC is simplified (without permutation of checkpoint levels) for its fast online decision. Second, EAIC uniquely embodies SI revocation for clouds. While Petri et al. studied the virtualization problem from a cost-profit perspective [11], our EAIC can boost such profits further.

Several recent virtual machine snapshot implementations adopt one form of incremental checkpointing, possibly with application-level checkpointing first, followed by VM disk incremental checkpointing [10]. Our work is complementary to that described in [10] as transparent application-level checkpointing. Separately, redundancy elimination (RE) is followed in [17] to send only non-redundant VM-image data chunks over the network.

3 Opportunities for Adaptive Checkpointing in RaaS Clouds

In this section, we present the model and experimental results exploring the potential of applying adaptive checkpointing for multithreaded applications in Resource-as-a-Service (RaaS) cloud systems, which could evolve as a new model

of buying and selling cloud computing resources [4]. In the RaaS cloud, providers sell physical resources (rather than conventional sever-equivalent virtual machines) for a relatively short duration (e.g., in the order of seconds) at a time. Our adaptive checkpointing will benefit pronouncedly especially under spot instance purchasing [1] (which represents the least expensive cost scenario to acquire Amazon's EC2 cloud resources). We first lay the background for unavailability events in cloud systems and develop the adjusted Markov model (AMM) that estimates the expected turnaround time of a running application. The proposed model is compared with the previous Yi's model [23], showing its superior accuracy with simpler calculation. Next, our AMM will be employed to uncover potential benefits of checkpointing in RaaS clouds. Finally, we present and discuss the experimental results of adaptive checkpointing (with delta compression) for multithreaded programs.

3.1 Adjusted Markov Model (AMM)

Cloud providers usually offer their customers options to purchase instances with variable levels of availability and prices. For example, Amazon customers can purchase *Reserved Instances* (RI), where a fixed amount of resources (e.g., CPU, memory) are always allocated and guaranteed beforehand. In this case, RI instances may still fail in the case of underlying hardware failures (albeit to very small likelihood). For multithreaded applications, such a hardware failure rate increases with a rising number of resources involved (e.g., physical CPU/cores, memory, etc.) because a failure in any of these resources may terminate all threads. Amazon also offers *Spot Instances* (SI), allowing customers to bid for its unused computational resources (usually with a lower price than that of RI). The SI price changes dynamically, possibly according to current market demands [23] or hidden reserved prices [3]. Amazon allocates resources for users whose bids are higher than the current SI price. However, the resources are revoked whenever the SI price hikes beyond the user bids. As a lower-price option, SI exhibits more likelihood for a long running job to be aborted resulting from SI revocation (besides hardware failures), especially under future RaaS clouds whose physical resources are sold for a relatively short duration at a time. RI and SI give Amazon users options to trade off performance and operational costs. Under SI, an application run will be terminated due to resource unavailability caused either by **(1)** actual hardware failures or **(2)** spot price hikes above the bidden price, deemed far more likely than hardware failures. On the other hand, RI is subject to resource unavailability caused only by hardware failures. Once an execution run is terminated due to a hardware failure, a substitute unit will be identified to replace the failed hardware unit for the terminated execution to resume its run using the latest checkpoint files. If the execution is terminated due to SI price hikes, the application run will resume later on, when RaaS cloud resources become available as their SI prices drop, utilizing the last checkpoint files on the newly obtained hardware resources. Effective checkpointing is thus especially crucial for application execution on RaaS clouds under SI pricing to minimize the cost involved, given that resource unavailability is then far more likely.

Next, we detail the model for calculating expected turnaround times and monetary costs when running multithreaded applications under SI or RI. Our analytical results will later reveal the potential gains (in terms of lower costs and application turnaround times) via adopting checkpointing for multithreaded tasks in RaaS cloud systems. Hardware failures in large data centers and clouds are modeled usually by probability distributions. Recent studies have revealed that the exponential distribution yields high accuracy [9], [15]. In this work, we adopt the exponential distribution with λ for unavailability to arrive at AMM, which enjoys fast calculation, where λ is the rate of events that terminate application runs due to resource unavailability. In the case of RI, these events reflect hardware failures, whose rate increases proportionally to the number physical resources involved in job execution. For a high-performance application, this number can be large and is expected to rise going forward due to the limitation of a single CPU or core (which tends to become smaller for better energy-efficiency, thereby requiring a job to run on many nodes; e.g., on 1024 nodes as commonly found in the current production system [9]). Under SI pricing, on the other hand, resource unavailability which leads to execution abortion is more likely caused by price hikes, besides relatively rare cases of hardware failures. Given a program with the base execution time w (i.e., runtime under no unavailability nor checkpointing), running in the ${t_p}^{th}$ bidding time slot, the state-of-the-art model for SI revocation predicts the expected turnaround time of the program, $T(w, t_p)$, as [23]:

$$T(w,t_p)=\frac{w\sum_{k=0}^{\infty}(k+t_p)+\sum_{k=0}^{w-1}(k+r)f(k+t_p,p_b,p_s)}{1-\sum_{k=0}^{w-1}f(k+t_p,p_b,p_s)}, \tag{1}$$

where r is the restart time, and $f(t, p_b, p_s)$ is the probability density function of revocation occurrence in the time interval of $[t_p, t_p + t]$ with user's bid price p_b and current SI price p_s. Yi et al. used the model to derive their checkpointing schemes [23], whose time complexity and space complexity equal $\mathrm{O}(t_p)$ and $\mathrm{O}(n_l \times n_p)$, respectively, where n_l is the number of bidding time slots for the learning period at the beginning (e.g., number of minutes in 112-day learning period) and n_p is the number of possible SI prices with granularity of interest (e.g., $\$0.001$ granularity). Computed *offline*, the model assumes a *constant* restart time r, implying that the next bidding process always starts right after a revocation. This assumption does not hold true in practical situations, because the recovery process can be launched only after the user wins the bid again (i.e., when the SI price falls below the bid price).

As fast prediction is crucial for EAIC *online* checkpointing, it calls for swift checkpoint decisions. To this end, we model SI revocation with a modified exponential distribution, to arrive at low complexity of O(1) in time and space. Additionally, our model merges seamlessly with hardware failure modeling for fast calculation. Using the techniques developed earlier [8], [9], [18] and based on the results of our analysis on the SI 8-month price trace on a production cloud system (studied in [23]), we devise the adjusted Markov model (AMM), with its revocation rate, λ, multiplied by the adjusting factor α. The devised model admits R, the average time duration when SI price falls below the bid price after revocation. Similar to Yi's model, AMM obtains λ, α, and R from the learning period. Its

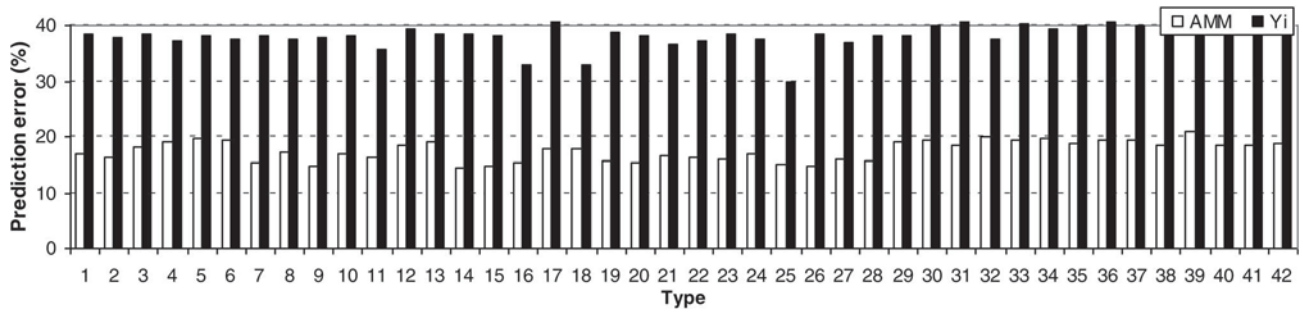

Fig. 1. Model prediction error for 10-minute program (10Min).

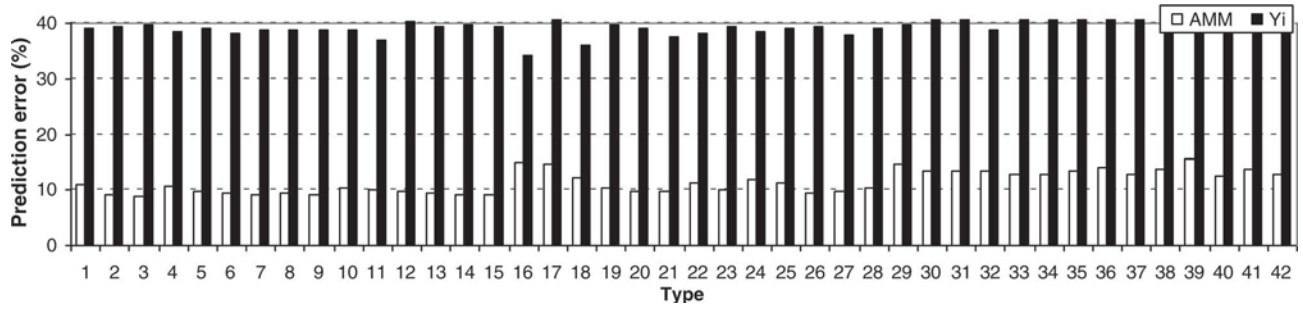

Fig. 2. Model prediction error for 1-hour program (1Hr).

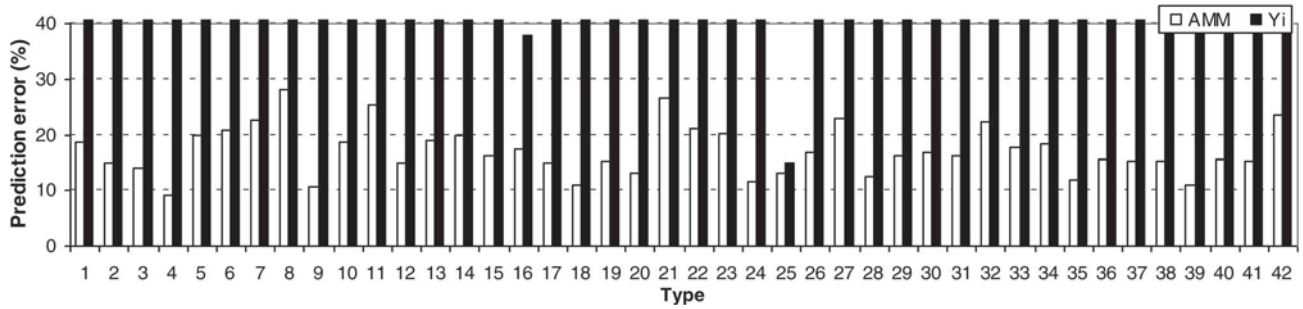

Fig. 3. Model prediction error for 10-hour program (10Hr).

space complexity equals O(1) since AMM keeps two parameters (λ and R), with α derived from λ and the observed turnaround time. AMM details with EAIC checkpointing will be presented in Section 4.2.

For evaluation, we simulate 10,000 samples of AMM and Yi's model by distributing those samples over 8-month period for Yi's real price traces of 42 type instances (i.e., $2\ \text{OS types} \times 3\ \text{regions} \times 7\ \text{machine types}$; see [23] for full details) of the Amazon EC2 spot instance. For each sample, the model learns the price traces for a given period and makes a prediction for the expected turnaround time of the running program, with the prediction error calculated from the observed turnaround time of actual program execution after that point. For Yi's model, we resort to their 28-day, 56-day, 84-day, and 112-day learning periods (as required, see [23]) and presented the best results among all. For AMM, on the other hand, the 1-day learning period is sufficient.

Figs. 1-3 demonstrate prediction errors of the AMM model (unfilled bars) and Yi's model (filled bars) for 10-minute (10Min), 1-hour (1Hr), and 10-hour (10Hr) program runs on 42 ITs (i.e., instance types). The AMM model is seen to lower prediction errors substantially in comparison to Yi's model across all ITs for a wide range of program execution durations examined. On an average, AMM yields 2.1x, 3.4x, 6x reduction in prediction errors when compared with Yi's model [23] for 10Min, 1Hr, and 10Hr program runs, respectively. The reduction gap increases with longer execution since more chances then exist for revocation, and each revocation under the Yi's model is assumed to have a *constant* restart time inappropriately. For most ITs, AMM consistently exhibits acceptable prediction errors, staying within 20%.

3.2 Checkpointing in RaaS Cloud Computing

This section employs AMM to uncover potential benefits of checkpointing in RaaS clouds. Generally, the occurrence of resource unavailability is far more likely under SI than under RI. To quantify the rate of resource unavailability under SI, we designate the *unavailable factor* (UF) as the ratio of resource unavailability under SI pricing to that under RI pricing. Let the *Price Factor* (PF) of SI refer to the ratio of the SI average price to the mean RI price. It is then expected that applications run under SI pricing with $\text{UF}\,(\text{unavailable factor, defined earlier}) = 100$ and $\text{PF} = 0.5$ face unavailable events 100 times more in comparison to run under RI pricing and

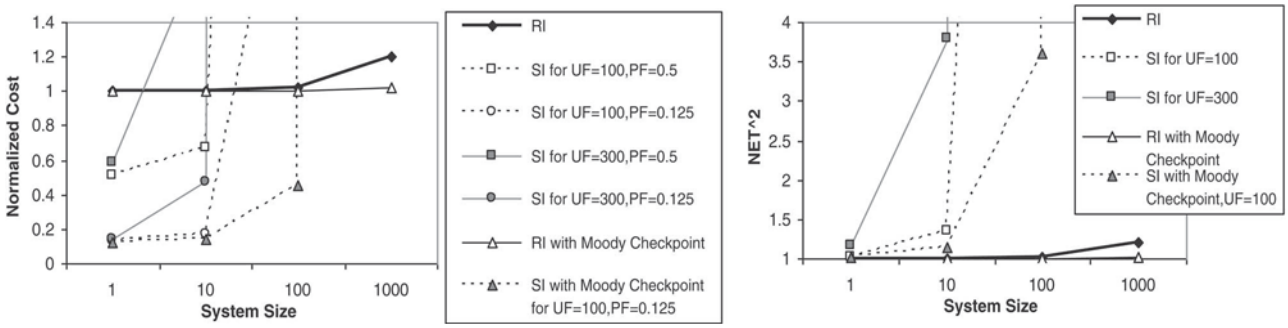

Fig. 4. Normalized cost and NET^2 under RI and SI pricing with and without Moody checkpointing for different UF, PF, and system sizes.

involve one half of the costs. Without checkpointing, terminated programs must restart from the beginning once hardware resources become available. With checkpointing, the aborted application runs resume execution from their latest checkpoints.

Given resource unavailable parameters of UF and PF, we use our AMM to predict the application expected runtimes under RI and SI. Our first performance metric of interest is the *normalized expected turnaround time* (NET^2), defined as $\mathrm{NET}^2 = T/t$, where T is the expected runtime from Markov model, and t equals the application runtime in the absence of resource unavailability. The second metric is the *normalized monetary cost* to finish executing the job, which is the total cost normalized by the cost of running the same job for t seconds under RI without unavailability. In the following results, we assume the base hardware failure rate, λ, to be 1×10^{-7} (i.e., once in 4 months), which stands in the same magnitude as observed in current data centers [9], [15]. In the case of checkpointing, the system profiles are taken from those of the Coastal cluster at Lawrence Livermore National Laboratory (LLNL) [9]. Given a failure rate under RI (which equals the resource unavailable rate), the unavailable rate under its SI counterpart is obtained by multiplying it with UF. The checkpointing results are calculated from our AMM augmented with the *static* multi-level checkpointing model (dubbed the Moody Model) available to the public [9]. These settings represent multi-level checkpointing systems for high performance applications. The application profiles for results demonstrated in Fig. 4 are based on `blackscholes`, one of our target benchmarks. Results for other benchmarks can be obtained in a similar way and are omitted due to space limitation. The *system size* is defined as the number of nodes used for application runs. In general, more running threads or MPI processes require larger system sizes to yield desirable efficiency [9], [14]. However, the speedup generally is not linear and a larger system is subject to a higher unavailable rate since one physical failure aborts the whole program.

Fig. 4 illustrates normalized costs and NET^2 of RI and SI under $\mathrm{UF} = \{100, 300\}$ and $\mathrm{PF} = \{0.125, 0.5\}$, for the system size ranging from 1 to 1000. UF is selected to equal 100 and 300 to reflect the fact that SI is far more likely to experience resource unavailability due to spot price hikes on the RaaS cloud, when compared with its RI counterpart that is subject to only rare physical failures. PF ranges are chosen by inspecting possible values from Amazon EC2 price pages [1]. Results in Fig. 4a show that SI has potentials to lower total running costs for the system size up to 10 nodes (for $\mathrm{PF} = 0.5$) and beyond (for $\mathrm{PF} = 0.125$) under $\mathrm{UF} = 100$, when compared to RI, because the smaller costs under SI outweigh higher chances of resource unavailability. If unavailability becomes 3 times more common with $\mathrm{UF} = 300$ under SI, however, a lower cost results for a small size (up to 3 under $\mathrm{PF} = 0.5$ and up to 10 under $\mathrm{PF} = 0.125$, in comparison to its RI counterpart. This is chiefly because the chance of resource unavailability under SI then jumps, likely to wipe out the gain from the lower price of SI sooner.

Meanwhile, accompanying execution time outcomes versus the system size are depicted in Fig. 4b, where PF is irrelevant under SI and thus only one single curve exists for each UF value under SI. With $\mathrm{UF} = 100$ and the system size equal to 10, SI has a larger NET^2 amount than its RI counterpart (by some 30%), as the price paid for its lower cost incurred (by some 35% reduction with $\mathrm{PF} = 0.5$ and some 84% reduction with $\mathrm{PF} = 0.125$) shown in Fig. 4a. In general, an interesting tradeoff between the cost and the execution time is observed under SI versus under RI. SI saves costs with prolonged execution times, suitable for non-real-time and cost-conscious applications. If the system size grows, SI may become inferior in both the cost and the NET^2 metrics (e.g., with some 30 nodes even under $\mathrm{UF} = 100$ and $\mathrm{PF} = 0.125$), since unavailability then grows large enough to negate any potential gain from lower SI pricing, given that each terminated run has to restart from the beginning of its execution in the absence of checkpointing. Hence, checkpointing is indispensable for moderate and large systems to address frequent unavailability. We include the results of RI and SI with known multi-level checkpointing (called Moody Model [9]) in Fig. 4 for comparison.

Fig. 4a illustrates that SI with Moody checkpointing enjoys substantial cost reduction for a wide range of system sizes (up to 100), when compared to its RI counterpart, as contrasted between the dashed curve with filled triangles and the solid curve with hollow triangles. Meanwhile, the execution time results (NET^2) of SI and of RI with Moody checkpointing stand close to one another for the system size up to 10, where SI begins to see its NET^2 values shoot up when the size rises further, as demonstrated by the dashed curve with filled triangles in Fig. 4b. Checkpointing benefits both SI and RI in terms of the two performance metrics when compared with their non-checkpointing counterparts. As can be found in Fig. 4, both cost and NET^2 amounts are lower for all system sizes with checkpointing than without checkpointing. The Moody model is of static non-compression multi-level checkpointing, with its results included in Fig. 4. Next, we illustrate motivation for our enhanced adaptive checkpointing algorithm called EAIC, which will be shown to outperform Moody in Section 5.

TABLE 1
Target PARSEC Applications

Program	Details	Number of threads created as a function of -*n* option [14]
Blackscholes (BS)	Portfolio	$1+n$
Bodytrack (BT)	Computer Vision	$1+n+1$
Canneal (CN)	Chip Design	$1+n$
Ferret (FR)	Similarity Search	$3+4n$
Fluidanimate (FA)	Interactive Animation	$1+n$
Streamcluster (SC)	Online Clustering	$1+n$
Swaptions (SW)	Portfolio	$1+n$

3.3 Adaptive Checkpointing in Multithreaded Applications

As outlined in Section 2, checkpointing to remote storage is expensive but necessary for desirable reliability and execution performance. As the file size affects the remote checkpoint latency crucially, several size reduction techniques have been considered, with incremental checkpointing and delta compression widely treated [12], [13], [21]. To further reduce checkpointing overheads, our EAIC takes checkpoints adaptively, dictated by the expected compressed file size (delta size) and compression time (delta latency), which vary dynamically in the course of job execution. While some single-threaded applications have been found to exhibit high dynamicity regarding their delta latency and delta size during execution [8], we wonder if multithreaded applications hold similar dynamicity in latency and size. To this end, we profile seven (7) target applications selected from the PARSEC 3.0 benchmark suite [5] (listed in Table 1). Details about selecting these benchmarks are given in Section 5.1. The profiling process and motivating results are outlined next.

It should be noted that since RaaS is deemed as a future cloud service unavailable yet from existing providers [4], we have established a testbed with networked multicore nodes, which resemble physical resources possibly purchased by clients from RaaS clouds, with dedicated CPU cores, memory, disk space and I/O capacity, and networked link bandwidth. The established testbed permits our investigation into EAIC on networked multicore systems, expected to be made available from RaaS clouds.

The profile starts by running benchmarks on our testbed for 30 seconds, at which the first full checkpoint is taken. After that, the corresponding metrics of delta compression are measured at every execution second during 60-second running interval. To measure the pure delta latency and delta size, we halt the job execution until each delta compression is finished. Fig. 5 illustrates the results of benchmark delta latency and delta size normalized by their corresponding means over the profiling interval, where only those of three benchmarks (i.e., BS, FR, and SC) are included for clarity. Results in Fig. 5 reveal that wide swings indeed exist for the delta latency/size over time, favoring adaptive checkpointing. For example, FR exhibits 80% (or 19%) reduction in its delta latency (or delta size) if its checkpoint is taken at the 36^{th} second, instead of the 35^{th} second. On the other hand, BS delta latency and delta size are almost constant. FR and BS are two extreme cases among our 7 target benchmarks, in terms of their latency and size fluctuations over time. Our EAIC aims to choose most desirable checkpoint times effectively based on real-time prediction of such delta latency and delta size during job execution.

4 Toward Efficient Checkpointing in Clouds

4.1 AIC Overview

Our EAIC is the enhanced version of the previous AIC [8], which consists of multiple components orchestrating an effective checkpointing mechanism. It is built upon the Berkeley Lab Checkpoint/Restart (BLCR) library [7]. Once an application compiled with the AIC library is in execution, AIC will be awaken periodically to make checkpoint decisions, based on the optimal checkpoint interval calculated using the AIC Markov model. Such a multi-level checkpointing model requires to predict current checkpoint costs involved (e.g., delta latency and delta size, if delta compression is to be executed for a fast remote checkpoint). Without profiling requirements, AIC accomplishes its cost prediction by means of *Stepwise Regression* and *Online Gradient Descent Algorithm*. Being a multi-level checkpointing scheme, AIC takes its first-level checkpoint by performing an incremental checkpoint to its local disk. AIC accelerates the higher-level checkpoint (which commonly involves sending files over a slow network) by applying special page-aligned delta compression between consecutive checkpoint files before sending them out. These operations are performed by a separate process (i.e., checkpointing process) so as to let the application process continue its execution without suspension (arriving at *concurrent/asynchronous checkpointing*). The checkpointing process may be run at a dedicated CPU core (which also accommodates checkpointing processes of all other applications running within the same node, with one such process for each application), or at any core which also executes application processes (without involving any extra CPU core to hold down the monetary cost). We evaluate those two alternatives in Section 5.3.

4.2 EAIC Adjusted Markov Model

This section describes the new adjusted Markov model (AMM) for EAIC adaptive checkpointing that incorporates both hardware failures and spot instance revocation events. As presented in Section 3.1, AMM enjoys significantly lower prediction errors than previous models.

Our EAIC model handles SI revocation and three failure levels ($\mathrm{f}_1, \mathrm{f}_2, \text{and } \mathrm{f}_3$) with two checkpoint types (c_2 and c_3), as illustrated in Fig. 6, where w is the time to do actual work, c_i is the checkpoint time of failure level i, r_i is the recovery time from c_i, and s is the average time since SI revocation until the price falls below the user bid again. The transition edge represents the possible events. In the multi-level model, the checkpoint with level i can recover all the failures with level $j \leq i$. Once recovered from the hardware failures (say, at State r_2 or r_3), the program must rerun any unsafe work (say, at State c_2 or $c_3 + c_2$). For SI revocation (at State s), we assume that no hardware failure occurs since there is no SI hardware allocated. Once the user wins bidding again, the worst case is assumed to follow where the program must be recovered from the highest-level checkpoint (i.e., c_3). In practice, the

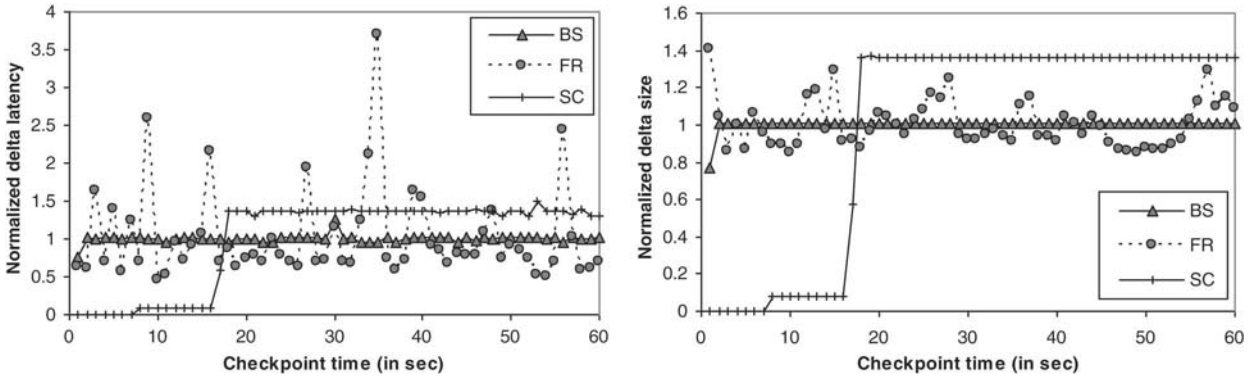

Fig. 5. Normalized delta latency and delta size of three PARSEC benchmarks obtained using our test bed when taking the next (incremental) checkpoint at different points of time over a 60-second interval. The outcomes are normalized over respective benchmark's latency/size means in the interval.

system may opportunistically employ a lower-level checkpoint. The AMM adjusting factor makes it possible to add SI states (State s) to the AIC model [8] while retaining high accuracy.

4.3 Incremental Checkpointing for Multithreaded Applications

Incremental checkpointing has been introduced for years to handle writing-only dirty pages effectively, with three implementation types, each with its benefits and drawbacks. The first type is to keep track of modified pages inside the kernel [6], [20]. It incurs low overhead but requires a patch for each kernel release, hindering its usability. The second type avoids modifying the kernel source by introducing a kernel library which clears the write bits and inspects dirty bits in page tables [19]. However, this method alters the kernel definition of write bits, possibly causing incompatibility in the future. The third type provides the library that uses `mprotect()` function to write-protect pages and to collect the dirty page by the library itself once the program tries to write the page, triggering a page fault event [12], [21]. It incurs overhead in invoking the signal handler but does not modify or expect any definition of kernel data structures. We follow this last implementation type for high compatibility and portability.

Despite its long existence, the `mprotect` method has never been made to support multithreaded applications yet. The main challenge lies in the C/C++ signal handler limitation, which allows only asynchronous-safe functions (*async-safe* for short) to be called inside. This makes it impossible to have a straightforward way for updating the shared dirty page list inside the page fault handler, because there is *no* thread synchronization function that is async-safe! We worked around this problem efficiently by introducing one EAIC *supporting thread* (ST), to handle updating dirty page list and to make checkpoint decisions. Note that the page fault event is synchronous, meaning that the event will be delivered to the thread that causes it. To update the dirty page list, we allocated one shared POSIX `pipe` for *computing thread* (CT) to inform ST of any dirty page arrival. This method is based on the fact that `read()` and `write()` functions (used for reads and writes in the shared pipe) are both async-safe and thread-safe. In addition, one barrier is added inside the checkpoint kernel module to drain all remaining dirty page information within the pipe during the checkpoint process, thereby allowing to selectively write only dirty pages to the checkpoint file. Such an extra barrier incurs minor overhead as many barriers already exist in BLCR to ensure its correctness. Low overhead of EAIC will be confirmed by measured results given in Section 5.2.

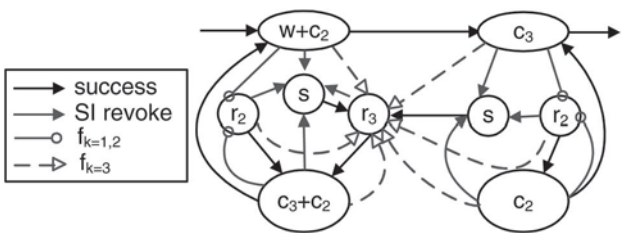

Fig. 6. EAIC adjusted Markov model for spot instance.

4.4 EAIC Enhancement for RaaS Clouds

Given hardware resources purchased from the RaaS cloud under SI pricing are expected to cause more system variability than a dedicated multi-core system, EAIC must adapt quickly to frequent system changes. We briefly outline EAIC enhancements incorporated in our implementation. First, EAIC reacts to network dynamicity by opportunistic network bandwidth measurement with *Exponential Smoothing* for better accuracy. Second, EAIC may *selectively* skip delta compression in the case where doing so increases the total checkpoint latency. Third, EAIC may send checkpoint files early during the stepwise regression period, at the beginning of execution when EAIC takes a few checkpoints (samples) to create its prediction model. In the case of high unavailable rates or a short execution time, adequate samples must be sent out even before the prediction model is established.

5 Performance Evaluation

In this section, we first evaluate the performance of EAIC on the *real* testbed by comparing it with (1) its static counterpart, (2) its basic version without cloud enhancement [8], and (3) the Moody model [9] (recent *static* multi-level checkpointing). All checkpointing methods are based on the Markov model and are implemented on our experimental testbed, aimed to resemble acquiring resources from RaaS clouds. The experimental results reveal that our EAIC outperforms its counterparts with respect to NET^2, the aggregate file size, and the monetary cost.

Next, we compare EAIC against various checkpointing policies developed recently for spot instance by Yi et al. in [23].

In contrast to EAIC, Yi's schemes are based on the probability density function. We evaluate EAIC and Yi's schemes via the instrumentation process driven by 8-month real price traces of 42 types of Amazon EC2 spot instances [23]. The evaluation results reveal significant reduction in both the execution time and monetary cost.

5.1 Experimental Setup

The testbed for EAIC performance evaluation contains a cluster of Dell PowerEdge R610 nodes, with two quad-core Xeon E5530 processors running at 2.4 GHz and having an 8-MB shared last-level cache per processor. Each testbed contains 32 GB of physical memory and one 7200-RPM SATA disk. Running CentOS 5.5 64-bit with 2.6.18 kernel, cluster nodes are connected by a Gigabit switch. The testbed permits four possible types of multi-level checkpointing implemented as libraries for comparison:

- Moody: periodically taking full checkpoints without delta compression. The checkpoint interval is calculated using the Moody multi-level model [9].
- SIC (Static Incremental checkpointing with Compression): periodically taking incremental checkpoints, with delta compression performed between two successive checkpoints. The static EAIC model is used to calculate its checkpoint interval. SIC represents the static asynchronous checkpointing [15] equipped with delta compression.
- AIC: Basic AIC without cloud enhancement [8].
- EAIC: our enhanced checkpointing mechanism.

Both Moody and SIC require average checkpoint latency information beforehand to calculate their optimal checkpoint intervals, while AIC and EAIC predicts such information online during job execution. Delta compression and remote checkpointing under SIC, AIC, and EAIC are conducted on a separate process to avoid impacting application execution. A cluster node performs level-2 (L_2) checkpoints by pushing (possibly compressed) files to the memory of its partner node, which is another cluster node pre-assigned deterministically. In addition, a Lustre file system (version 2.1.3) has been set up as level-3 (L_3) checkpoint storage with 90 MB/s aggregate write bandwidth. Our testbed reflects a leased large-scale system with fixed L_3 bandwidth per node. If the system expands (e.g., with more compute nodes), total bandwidth to the Lustre system grows accordingly; otherwise, the Lustre system soon becomes excessively congested, rendering EAIC more beneficial due to its better file size reduction.

Our target applications include seven (7) benchmarks from the PARSEC 3.0 suite, which represents workloads for multi-threaded programs with various emerging applications [5], likely to be widely use in future RaaS clouds. We select our targets among PARSEC benchmarks that have been ported to OpenSolaris previously in [14], reflecting their high usability. The `facesim` benchmark is dropped since it fails to include a large input size needed for evaluation. When running PARSEC benchmarks, $-n$ option may be set to indicate the *minimum* number of threads spawned for execution. Table 1 lists the exact number of threads created under a given $-n$ option for each benchmark, and the number is application-dependent since it is based on how the program partitions problem space [14]. The benchmarks are compiled with one of four checkpointing libraries (Moody, SIC, AIC, EAIC) separately for comparison, so that they would take checkpoints either statically or adaptively on-the-fly during benchmark execution. For each checkpoint taken, we measured incremental checkpoint latency, delta latency, delta size, and time to transfer data to partner node and to Lustre (for L_2 and L_3 checkpoints). With them, the performance metrics of interest can then be calculated.

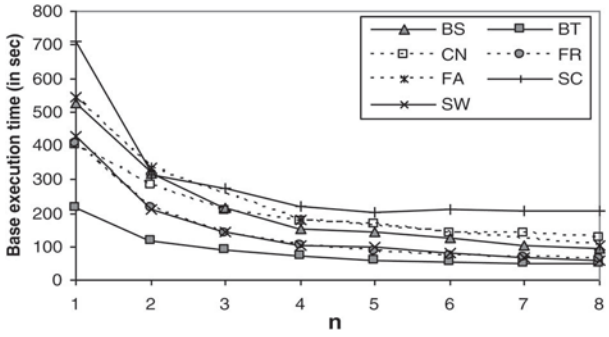

Fig. 7. Base execution time of target benchmarks under different PARSEC $-n$ options.

5.2 EAIC Failure-Free Overhead

We first set the *base execution time* as the benchmark execution time without any failure or unavailability and in the absence of checkpointing overhead (i.e., without any checkpointing). This amount reflects the best-case runtime of the application. Fig. 7 depicts the base execution times of target benchmarks under different $-n$ options when running on one node of our testbed cluster. As n increases, the benchmark spawns more threads to speed up its execution. As uncovered in Fig. 7, performance gains tend to level off for $n > 4$ under one node.

Fig. 8 plots EAIC overhead, defined as the percentage of benchmark execution time increases under EAIC in the absence of hardware failure and resource unavailability, when compared with their base execution times, for various $-n$ options. It shows that EAIC overhead is fairly low (always below 2.2%) for five benchmarks (i.e., BS, BT, FR, SC, SW). On the other hand, CN and FA exhibit relatively higher EAIC overheads (bounded by 6% and 10% respectively for CN and FA at $n = 8$). Given that no failure is present, this overhead is due to the EAIC supporting thread and its signal handler, with its amounts likely to rise gradually when n increases, as a result of intensifying thread-related overhead activities (e.g., barriers or pipes).

5.3 EAIC Performance under Experimental Testbed

In this section, we compare EAIC against previous checkpointing schemes that are also based on the Markov model. We first present the results of EAIC and Moody with and without extra dedicated CPU cores, discussing their benefits and costs on cloud systems. We then compare EAIC with other schemes.

Here, our performance metrics of interest include the *normalized expected turnaround time* (NET^2) (defined in Section 3.2), the *aggregate file size* (AFS), and the average monetary cost per application run (C). While NET^2 reflects the expected execution time, AFS indicates the mean size of the aggregate file sent to L_3 storage, consuming extra space and bandwidth which are usually charged separately under cloud systems (e.g., $\$0.095/\mathrm{GB/month}$ for S3 storage and $\$0.01/\mathrm{GB}$ for

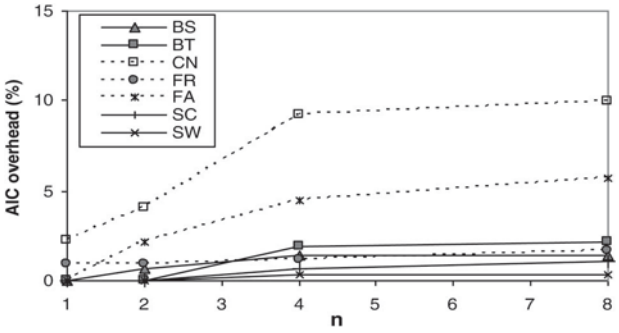

Fig. 8. EAIC overheads (presented as the percentage of base execution time) under failure-free scenario with different $-n$ options.

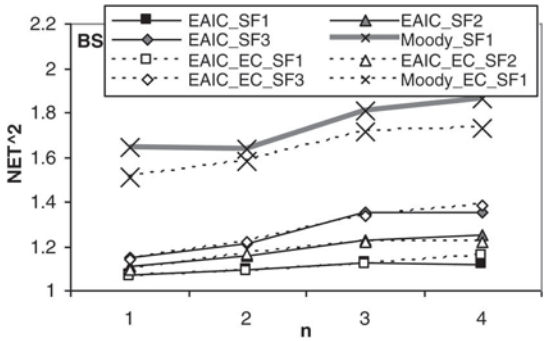

Fig. 9. NET^2 of BS benchmark under EAIC and Moody with and without extra cores (EC) for different n and SF values ($m = 3$).

transferring data to other machines across availability zones [1]). Hence, it is desirable to keep NET^2 and AFS as low as possible (with NET^2 lower-bounded by 1.0). NET^2 and AFS outcomes are calculated from corresponding checkpointing models (i.e., the AMM model given in Section 4.2 for EAIC and other models presented earlier [8], [9]). The monetary cost per application run, C, are calculated by

$$C = NET^2 \times B \times \left(\frac{p_c}{s_h} + \frac{AFS \times p_s}{s_m}\right), \tag{2}$$

where B is the application base execution time, $\mathrm{p_c}$ is the price for SI Amazon EC2, p_s is the price for S3 Storage, and s_h (or s_m) equals the number of seconds in one hour (or month). Since checkpoint files can be discarded once the application is completed, AFS extra space is needed only during the time span of application execution (and thus its cost is calculated as the pro-rated monthly charge). The term inside the parenthesis pair denotes the cost per second. When multiplying it with the actual runtime (in second), we obtain the monetary cost per application run. In this section, we match the price of SI Amazon EC2 with our testbed machine type, yielding $p_c = \$0.115/\mathrm{hour}$. The storage cost is assumed to equal that of S3 storage (which provides the similar function and reliability to our Lustre system), at $\$0.095/\mathrm{GB/month}$. Given the future RaaS cloud tends to revoke resources in fine time-granularity (e.g., in the order of seconds [3]) and to spawn more threads due to the limitation of a single core, we assume a relatively high unavailable rate of 1.0×10^{-3} to capture both RI and SI pricing scenarios. The hardware failure type breakdowns are assumed to equal those observed in the Coastal system at LLNL [8].

We evaluate EAIC under three running parameters, m, *Sharing Factor* (SF), and n, with m defined as the number of cluster nodes (allocated by RaaS) currently participating in job execution. SF is the number of application instances launched on *each* cluster node, while n is the PARSEC $-n$ option, which indicates the number of running threads of *each* application instance.

We first study how sensitive the dedicated extra core is to checkpointing performance. In networked multicore systems, idle cores usually exist and using them is encouraging for increased resource utilization [8]. However, cloud systems charge according to the number of virtual cores leased, with more cores incurring a higher cost. As a result, having dedicated (extra) cores for checkpointing may nullify potential benefits even if quicker execution completion is resulted. If the checkpointing task is light-weighted enough (like our EAIC), those cores employed for regular thread execution may also accommodate the checkpointing thread(s) without compromising regular thread performance.

NET^2 outcomes as a function of n are illustrated in Fig. 9 for EAIC and Moody with and without any extra core for checkpointing for the BS benchmark under $m = 3, \mathrm{SF} = \{1, 2, 3\}$, and $n = \{1, 2, 4, 8\}$. In Figs. 9-12, we leave out some outliers for clarity. Note that EAIC and EAIC_EC (indicating EAIC with an extra core for checkpointing) both apply concurrent checkpointing; while EAIC_EC pins all checkpointing processes to the dedicated *extra* core, EAIC lets the checkpointing processes run freely on application cores. Both EAIC and EAIC_EC have the same number of application cores (i.e., 7), where application threads run freely on those. Fig. 9 shows that the NET^2 outcomes of EAIC and EAIC_EC are closed, and in some cases, EAIC even yields lower NET^2 despite the fact that it involves fewer cores (e.g., 7 cores versus 8 cores). This counter-intuitive phenomenon is likely due to cache misses, since the checkpointing processes under EAIC have chances to run on the same cores as the computing threads, lowering cache misses, whereas EAIC_EC experiences plentiful cache misses on the dedicated core for the checkpointing thread. Given similar performance levels under EAIC and EAIC_EC, we conclude that EAIC checkpointing is light-weighted and hence, it needs no extra cores and can be accommodated by original cores provisioned for application threads without compromising their execution performance. On the other hand, Moody_EC yields noticeably lower NET^2 than Moody, signifying that Moody is not light-weighted and hence enjoys faster execution with an extra core. Next, we focus on how concurrent incremental checkpointing and delta compression applied in EAIC, AIC, and SIC improve NET^2 and AFS. Comparative discussion between EAIC, SIC, and AIC will be given later.

Figs. 10a and 10b compare NET^2 outcomes of EAIC, SIC, AIC, and Moody for BS and SC benchmarks under $m = 3$, $\mathrm{SF} = \{1, 2, 3\}$, and $n = \{1, 2, 4, 8\}$. For clarity, we include the results of Moody and AIC only for $\mathrm{SF} = 1$ (shown by curves with $\times$ marks). Among 7 target benchmarks, BS (or SC) exhibits the upper (or the lower) bound on NET^2 reduction yielded by EAIC over that of Moody (i.e., 4% to 40%). This fact reveals that EAIC benefits outweigh its overhead stated in Section 5.2 under practical circumstances. The results also show the effects of SF and n on the NET^2 value. With larger n and SF, resource contention inside computing nodes

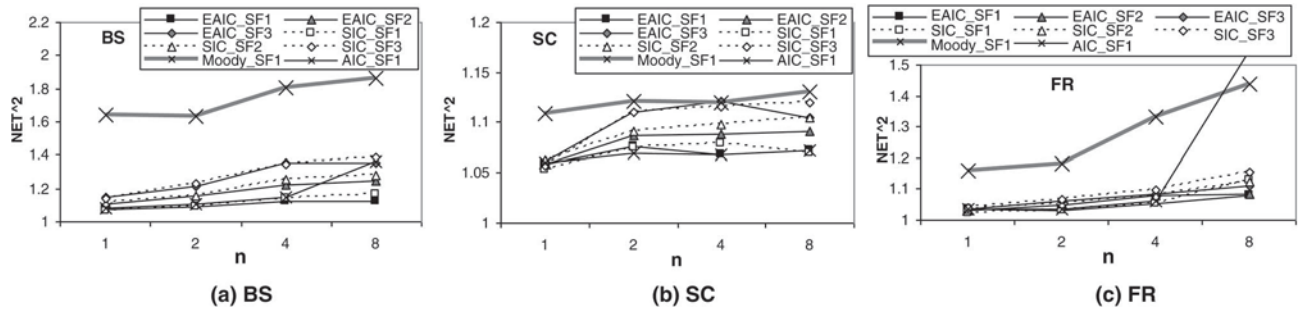

Fig. 10. NET^2 of (a) the best, (b) the worst, and (c) the average benchmarks under EAIC, SIC, AIC, and Moody for different n and SF values ($m = 3$).

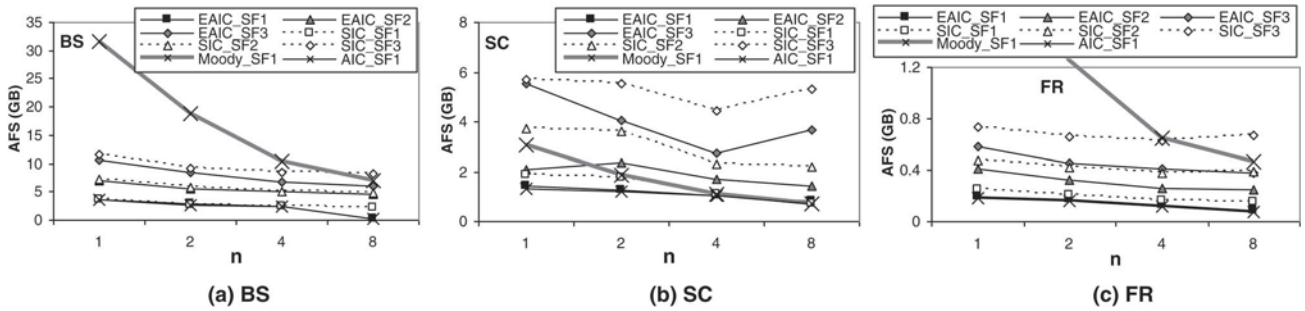

Fig. 11. Aggregate file size of (a) the best, (b) the worst, and (c) the average benchmarks under EAIC, SIC, AIC, and Moody.

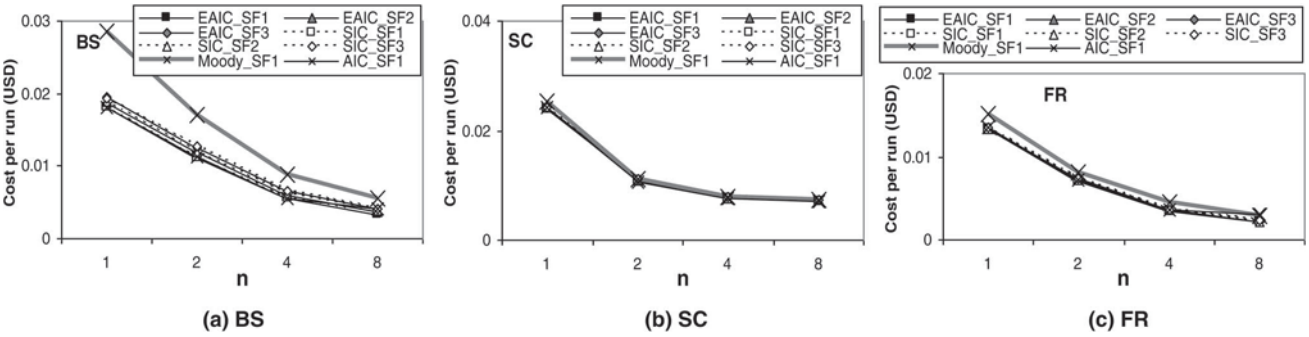

Fig. 12. Monetary costs per application run of (a) the best, (b) the worst, and (c) the average benchmarks under EAIC, SIC, AIC, and Moody.

(e.g., cache, memory, and I/O) rises, leading to bigger NET^2. However, we note that NET^2 indicates how much higher the expected execution time will be above its base execution time (without any unavailable event). NET^2 results in Fig. 10 are normalized by their base execution times with corresponding PARSEC $-n$ options (as illustrated in Fig. 7). As the base execution time of BS is $100 + \%$ longer at $n = 1$ than at $n = 8$, the expected execution time of BS is still shorter at $n = 8$ than at $n = 1$.

The AFS results are affected by n markedly, as illustrated in Fig. 11, where those under EAIC, SIC, AIC, and Moody checkpointing are included for the same range of n. Again, BS (or SC) represents the upper (or the lower) bound for AFS reduction due to EAIC when compared with Moody. BS is seen to achieve AFS reduction by up to 88% (i.e., 8.3 x reduction). A larger SF leads to increased AFS since more checkpoint files are then sent over. On the other hand, a larger n decreases the AFS of every target benchmark monotonically because of the following reasons. First, a multithreaded application generates only one file (based on working memory contents) per checkpoint, regardless of the number of threads, since all threads share the same memory address space. Second, each target application takes up a similar memory footprint, irrespective of its number of threads involved, as desired for a scalable multithreaded program. Apart from thread-specific information (such as register contents and thread-execution status information of each thread), the size of working data space is almost identical. Lastly, as the base execution time at a large n is short (see Fig. 7), the course of benchmark execution involves only a small number for checkpointing instances, requiring to send out a few files in total.

Fig. 12 illustrates the monetary cost per application run, where EAIC is seen to save the BS (or SC) execution cost by up to 40% (or by $3 \sim 5\%$). The total cost is dominated by the execution time, since the relative prices of S3 storage is currently far smaller (see Eq. 2). If the cloud starts to charge for the use of data bandwidth from EC2 to the storage

(e.g., when storage is located away from the Amazon unavailability zones [1] for more reliability), the AFS size could markedly affect the overall cost.

While not shown in this article, a larger m (the number of nodes) leads to higher NET^2, bigger AFS, and a loftier cost, caused by more resource contention at remote storage (i.e., the Lustre system). With the advantage of concurrent incremental checkpointing and delta compression clearly demonstrated, we next compare EAIC and SIC. As can be seen in Figs. 10a and 11a, limited differences exist for the results of benchmark BS under EAIC (denoted by solid curves) and under SIC (denoted by dotted curves). It can be explained using the profile depicted in Fig. 5, where the BS delta time and delta size are almost constant. As a result, adaptive checkpointing has negligible advantages over its static counterpart. Now, let us consider SC, which has a stair-like variation (see Fig. 5). EAIC makes use of such dynamicity to achieve marked AFS reduction in comparison to SIC (e.g., up to 38% at $n = 8$). Given its largest fluctuations over time among our 7 benchmarks, FR is expected to outperform under EAIC, yielding further reduction in NET^2 (or AFS) by up to 6% (or 84%) when compared with those under SIC. Since the results of FR under SIC leave limited room for improvement (with no more than 13% improvement possible at $n = 8$), EAIC thus exhibits a low NET^2 reduction level as expected. In fact, EAIC manages to reduce NET^2 by almost one half of that possible while yielding significant AFS reduction (more than $6\times$ reduction) over SIC. Given NET^2 dominates the monetary cost, EAIC yields limited cost reduction over SIC (by up to 4%), albeit being consistently the lowest.

Next, we compare EAIC with AIC, the previous adaptive mechanism aiming for the networked multicore system [8]. AIC results for $\mathrm{SF} = 1$ ($\mathrm{black\ thin\ curves\ with} \times\ \mathrm{marks}$) are presented in Figs. 10-12, revealing that EAIC yields lower NET^2 than AIC and the reduction gap grows with a larger n (up to 31% for FR). As detailed in Section 4.4, EAIC does selectively delta compression and early checkpoint transfer to enjoy big NET^2 reduction for a large n (i.e., high resource contention and short execution time). There is little difference in AFS under EAIC and AIC, indicating negligible space overhead due to AIC cloud enhancement. On the other hand, the monetary cost is reduced by AIC greatly (by up to 31% for FR), mainly due to NET^2 reduction.

In conclusion, EAIC can greatly improve NET^2 (up to 40%), AFS (up to $8.3\times$), and the monetary cost (up to 40%), over other Markov-model checkpointing schemes examined. As a result, we compare only EAIC performance with those of Yi's schemes in next evaluation driven by Amazon EC2 spot instance price traces.

5.4 Instrumentation Evaluation Setup

We evaluate our proposed checkpointing scheme for 42 spot instance types (ITs) of Amazon EC2 via an instrumentation process. The 8-month real price traces collected by Yi et al. for evaluating their checkpointing policies earlier [23] are adopted to drive our instrumentation evaluation, with four checkpointing schemes examined:

- EAIC: our adaptive checkpointing mechanism;
- A (SI adaptive checkpointing): making a checkpoint decision based on the probability density function (PDF) of SI revocation occurrence;
- C (current-price-based adaptive checkpointing): making a checkpoint decision based on PDF of SI revocation occurrence of the current SI price;
- IDEAL: the ideal checkpointing scheme, which checkpoints right before each SI revocation with the size reduction technique incorporated. Apart from information about revocation times, IDEAL is assumed to know the dynamic delta file size beforehand. While not attainable in practice, IDEAL serves as the baseline.

While other checkpointing schemes were considered in Yi's article [23], they are inferior to the two schemes (i.e., A and C) examined here for most ITs. Specifically, A prevails in 13 ITs, and C in 26 ITs among 42 ITs [23]. Similarly, Yi's other adaptive schemes mentioned in [22] were outperformed by A and C under the seven benchmarks listed in Table 1 and, thus, they are not included for comparison either. Hereinafter, the term of "Yi schemes" refers to Schemes A and C described in [23]. Every trace keeps one SI price per minute, as the price may change in each minute, yielding the SI price granularity in minute.

Performance metrics of interest are (1) job turnaround time (defined as the elapsed time since the job starts until it finishes, including all SI revocations and checkpoint-restart overheads), and (2) the monetary cost per application run. Target benchmarks employed for evaluation are those seven PARSEC applications (see Table 1). Without loss of generality in utilizing the 8-month real SI price traces to acquire resources for benchmark execution, we assume *one single user's bid price per instance type* throughout benchmark execution, set it to be the "median of the SI price range" recorded in the traces (like [23]). More details about price selection can be found in [2]. In order to cover SI price fluctuation over traces, ten initial time points were chosen randomly for a given trace, with each point specifying one subtrace to start at the initial time point for deciding resource availability during benchmark execution by means of an instrumentation evaluation process. Our instrumentation evaluation is detailed next.

For EAIC, instrumentation evaluation starts by obtaining λ (SI revocation rate), α (AMM adjusted parameter), and R (the average time duration when SI price falls below the bid price after revocation) per subtrace (specified by one randomly chosen initial time point) from a 1-day learning period. These parameters are employed by EAIC to determine resource availability during actual benchmark execution on our testbed based on the associated subtrace for collecting its checkpointing data (i.e., the execution turnaround time and the monetary cost).

By contrast, Yi schemes require the probability distribution of SI revocation, obtained through one 28-day to 112-day learning period per subtrace. Note that probability distribution information of SI revocation calls for a much longer learning period than EAIC (which needs just 1-day learning). Yi schemes also require the average checkpoint latency beforehand, and such latency information is measured on our testbed. Once the probability distribution and checkpoint latency are determined per subtrace, Yi schemes are instrumented by the subtrace for resource availability over benchmark execution. As the baseline case, IDEAL employs both dynamic checkpointing latencies (obtained by EAIC) and the future prices (given in the associated subtrace) beforehand, enabling it to checkpoint right before SI revocation with the lowest checkpoint latency ideally.

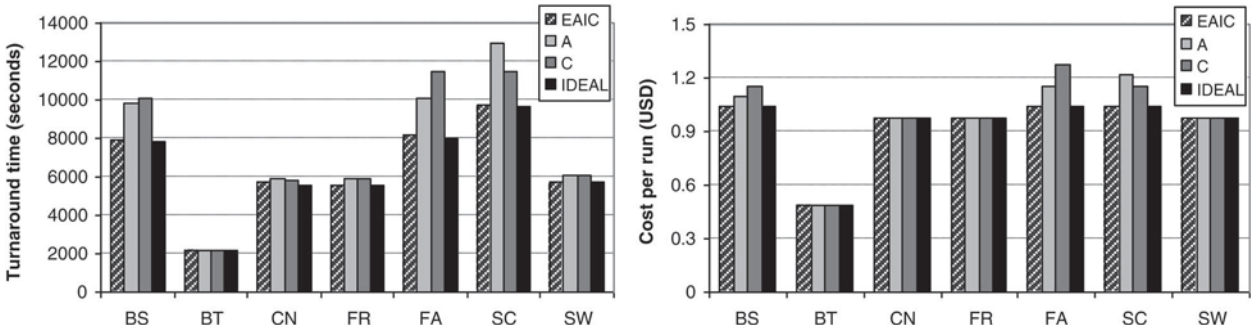

Fig. 13. Turnaround time and monetary cost per application run of seven benchmarks under EAIC, Yi schemes (A and C), and ideal checkpointing (IDEAL) for Instance Type (IT) 34.

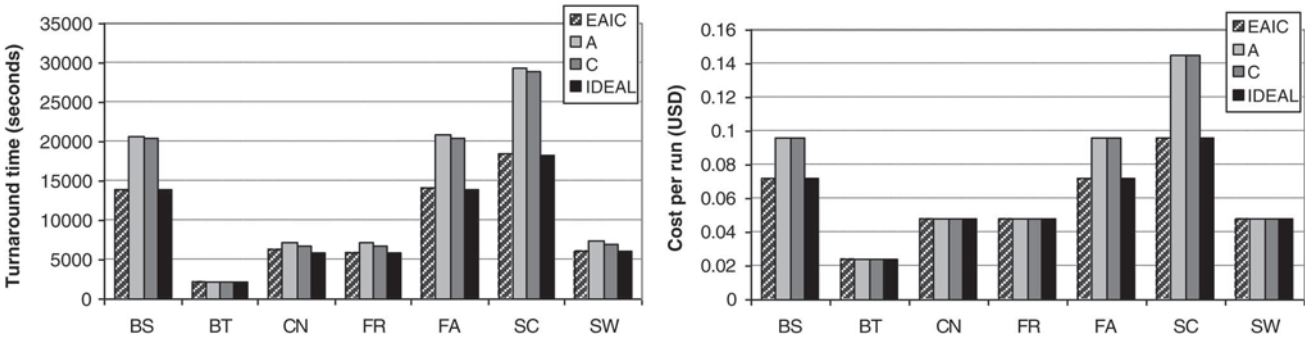

Fig. 14. Turnaround time and monetary cost per application run of seven benchmarks on Instance Type (IT) 25.

Given that PARSEC benchmarks complete their execution in a few hundred seconds (shown in Fig. 7), our instrumentation process repeatedly executes every benchmark many times (say 10 times) in order to make the total execution time of each benchmark in hours necessary for Yi schemes (which aimed at long-execution applications) and for adequate use of its associated subtrace to determine resource availability. It should be noted that hardware failure information is not included in the traces, and hence, our instrumentation evaluation did not consider hardware failures (which occur rarely). Under current practice, Amazon spot instance charges the user by its determined SI price (not user bid price) hourly. In the case of revocation, Amazon does not charge the last partial hour. We adopt this pricing practice when calculating the monetary cost. For each instance type and its corresponding price trace, the collected turnaround time and the monetary cost per application run are the averaged amounts of those obtained following 10 subtraces defined by the 10 trace initial time points chosen randomly.

5.5 Evaluation Results

Instrumentation evaluation has been conducted for all 42 Amazon EC2 instance types, with a representative result set demonstrated in Fig. 13, where mean turnaround times and monetary costs of EAIC, Yi schemes (A and C), and IDEAL for all target benchmarks under Amazon EC2 Instance Type 34 (IT34 for short) are included. EAIC is seen to enjoy shorter turnaround times(with up to 25% reduction) and smaller monetary costs (lowered by up to 20%) than Yi schemes. Furthermore, EAIC performs close to IDEAL (within 3.2%) for all benchmarks, attaining nearly the best outcomes possible.

As might be expected, EAIC exhibits various degrees of performance gains (in the mean turnaround time and the monetary cost) over its Yi counterparts under different ITs. While Fig. 13 shows a representative result set (under IT 25), the performance gain potential of EAIC can be noticeably higher, as depicted in Fig. 14, where EAIC is observed to enjoy up to 37% (or 34%) reduction in the turnaround time (or monetary cost) when compared with Yi counterparts for FA benchmark under IT 28. At the same time, EAIC maintains its performance results within 5% of those of IDEAL under such an IT.

Fig. 15 depicts the reduction amounts of performance metrics for EAIC over Scheme C (the most prevailing scheme known so far [23]) under each benchmark on every IT. The results are ordered along the y-axis by the growing turnaround time reduction amount for Benchmark FA. For such a benchmark, EAIC has the 3rd smallest (or the largest) reduction in the turnaround time under IT 36 (or IT 28), enjoying some 5% (or 58%) reduction in the turnaround time. Correspondingly, EAIC yields a decrease in the monetary cost by some 5% (or 33%) under IT 36 (or IT 28) for the benchmark. Overall, EAIC achieves turnaround time (or monetary cost) reduction in the range of 0% to 58% (or 0% to 59%). EAIC is seen to attain reduction more commonly and by larger amounts in the turnaround time (for nearly 95% of the total 294 cases, which involve 42 ITs and 7 benchmarks) than in the monetary cost. Such smaller monetary gains and fewer cases for cost reduction are due directly to the fact that Amazon does not charge for the last partial hour upon revoking SI execution. Therefore, the cost under EAIC may be identical to that under Yi Scheme C, even though EAIC shortens the application turnaround times. When compared with Scheme A (not shown in the figure), EAIC also reduces the turnaround time and the monetary cost markedly (by up to 45% and 34%, respectively).

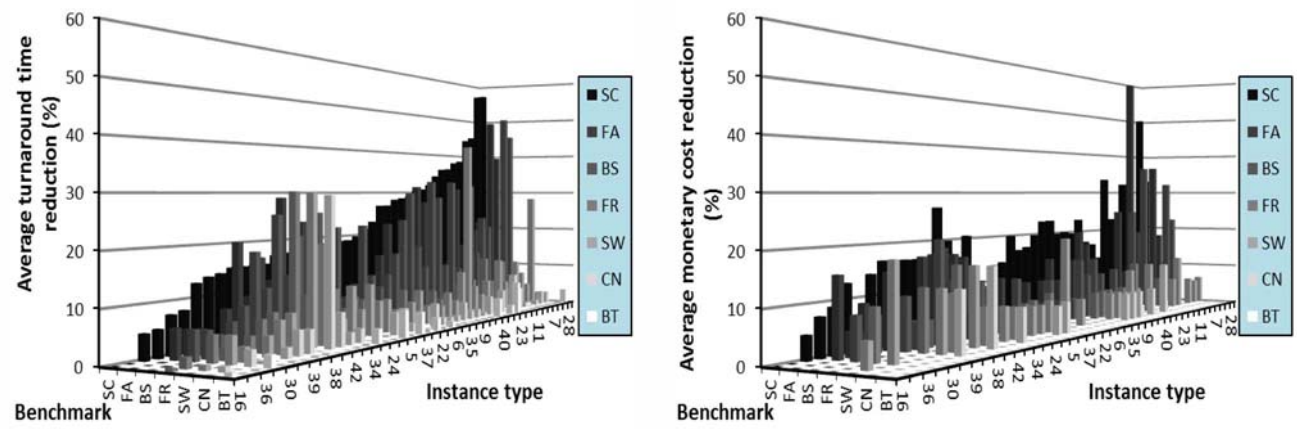

Fig. 15. Reduction percentage of turnaround time and monetary cost of 42 instance types obtained from the log, with their results illustrated in the increasing order of the FA turnaround time reduction amount along the y-axis.

In general, EAIC exhibits larger metric reduction on benchmarks with a longer base execution time (e.g., FA, BS, and SC) due to their higher chance of SI revocation to better benefit from effective checkpointing. A larger reduction gap is observed for the IT with more fluctuation in pricing. Under such a circumstance, EAIC indeed may reduce the application turnaround time and the monetary cost significantly when compared with previous schemes.

6 Conclusion

This paper has presented an efficient adaptive multi-level checkpointing scheme (dubbed EAIC, enhanced adaptive incremental checkpointing) for multithreaded applications run on networked multicore systems acquired from the envisioned future RaaS (Resource-as-a-Service) cloud [4] under spot instance (SI) pricing. We have developed the adjusted Markov model (AMM), which accommodates SI revocations and hardware failures to yield higher accuracy with lower complexity in space and time than the previous well-known model. According to our feasibility analysis, multi-level checkpointing may reduce cost and expected runtime of multithreaded applications under both fixed and bidden cloud pricing (e.g., Reserved Instances and Spot Instances). The nature of fluctuations in checkpointing overhead metrics commonly existing in the course of execution, as observed in our target multithreaded applications, reveals rich opportunities for adaptive checkpointing to lower overhead, as exploited by our EAIC. The design and implementation details of EAIC are provided, with highlights on cloud enhancement for adaptive checkpointing. With EAIC support, multithreaded applications enjoy shorter execution turnaround times, substantially reduced aggregate checkpoint file sizes, and lower monetary costs than under previous Markov-model checkpointing schemes, according to our experimental results. In addition, our instrumentation evaluation driven by real price traces of Amazon EC2 spot instances confirms that EAIC can substantially reduce application turnaround times and monetary costs under spot instances, exhibiting near-ideal outcomes. EAIC is thus particularly beneficial for future cloud systems. It will also benefit other cloud paradigms (like IaaS) facing frequent resource revocation (e.g., SI pricing).

Acknowledgment

This work was supported in part by the U.S. National Science Foundation under Award Number: CCF-0916451.

References

[1] Amazon Web Services, "Amazon EC2 Pricing and Instance Purchase Options," http://aws.amazon.com/ec2/pricing/, accessed in May 2013.

[2] A. Andrzejak, D. Kondo, and S. Yi, "Decision Model for Cloud Computing under SLA Constraints," *Proc. 3rd IEEE Int'l Symp. Modeling, Analysis & Simulation of Computer and Telecomm. Systems (MASCOTS)*, pp. 257-266, 2010.

[3] O.A. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "Deconstructing Amazon EC2 Spot Instance Pricing," *Proc. 3rd IEEE Int'l Conf. Cloud Computing Technology and Science (CloudCom)*, pp. 236-243, July 2010.

[4] O.A. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "The Resource-as-a-Service (RaaS) Cloud," *Proc. 4th USENIX Workshop Hot Topics in Cloud Computing (HotCloud'12)*, pp. 12-12, June 2012.

[5] C. Bienia, S. Kumar, J.P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," *Proc. 17th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT)*, pp. 72-81, Oct. 2008.

[6] R. Gioiosa, J.C. Sancho, S. Jiang, and F. Petrini, "Transparent, Incremental Checkpointing at Kernel Level: A Foundation for Fault Tolerance for Parallel Computers," *Proc. IEEE/ACM Int'l Conf. High Performance Computing, Networking, Storage, and Analysis (SC'05)*, pp. 9-23, Nov. 2005.

[7] P.H. Hargrove and J.C. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters," *J. Physics: Conf. Series*, vol. 46, pp. 494-499, June 2006.

[8] I. Jangjaimon and N.-F. Tzeng, "Adaptive Incremental Checkpointing via Delta Compression for Networked Multicore Systems," *Proc. IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS)*, pp. 7-18, May 2013.

[9] A. Moody, G. Bronevetsky, K. Mohror, and B.R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-Level Checkpointing System," *Proc. IEEE/ACM Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1-11, Nov. 2010.

[10] B. Nicolae and F. Cappello, "BlobCR: Efficient Checkpoint-Restart for HPC Applications on IaaS Clouds Using Virtual Disk Image Snapshots," *Proc. IEEE/ACM Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1-12, Nov. 2011.

[11] I. Petri, O.F. Rana, Y. Regzui, and G.C. Silaghi, "Risk Assessment in Service Provider Communities," *LNCS: Economics of Grids, Clouds, Systems, and Services*, vol. 7150, pp. 135-147, 2012.

[12] J.S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under Unix," *Proc. USENIX Ann. Technical Conf.*, pp. 213-224, Jan. 1995.

[13] J.S. Plank, J. Xu, and R.H.B. Netzer, "Compressed Differences: An Algorithm for Fast Incremental Checkpointing," Univ. of Tennessee, Technical Report CS-95-302, Aug. 1995.
[14] K.K. Pusukuri, R. Gupta, and L.N. Bhuyan, "Thread Reinforcer: Dynamically Determining Number of Threads via OS Level Monitoring," *Proc. IEEE Intl' Symp. Workload Characterization (IISWC)*, pp. 116-125, Nov. 2011.
[15] K. Sato et al., "Design and Modeling of a Non-Blocking Checkpointing System," *Proc. IEEE/ACM Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1-10, Nov. 2012.
[16] B. Schroeder and G.A. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," *IEEE Trans. Dependable and Secure Computing*, vol. 7, pp. 337-350, Oct. 2010.
[17] C.P. Tang, T.Y. Wong, and P.P.C. Lee, "CloudVS: Enabling Version Control for Virtual Machines in an Open-Source Cloud under Commodity Settings," *Proc. IEEE Network Operations and Management Symp. (NOMS)*, pp. 188-195, Apr. 2012.
[18] N.H. Vaidya, "A Case for Two-Level Recovery Schemes," *IEEE Trans. Computers*, vol. 47, no. 6, pp. 656-666, June 1998.
[19] M. Vasavada et al., "Comparing Different Approaches for Incremental Checkpointing: The Showdown," *Proc. Linux Symp.*, pp. 69-80, June 2011.
[20] C. Wang, F. Mueller, C. Engelmann, and S.L. Scott, "Hybrid Checkpointing for MPI Jobs in HPC Environments," *Proc. 16th IEEE Int'l Conf. Parallel and Distributed Systems*, pp. 524-533, Dec. 2010.
[21] S. Yi, J. Heo, Y. Cho, and J. Hong, "Adaptive Page-Level Incremental Checkpointing Based on Expected Recovery Time," *Proc. ACM Symp. Applied Computing (SAC)*, pp. 1472-1476, Apr. 2006.
[22] S. Yi, D. Kondo, and A. Andrzejak, "Reducing Costs of Spot Instances via Checkpointing in the Amazon Elastic Compute Cloud," *Proc. 3rd IEEE Int'l Conf. Cloud Computing (CLOUD)*, pp. 236-243, July 2010.
[23] S. Yi, D. Kondo, and A. Andrzejak, "Monetary Cost-Aware Checkpointing and Migration on Amazon Cloud Spot Instances," *IEEE Trans. Service Computing*, vol. 5, pp. 512-524, Nov. 2012.

Itthichok Jangjaimon received the BE degree in computer engineering from Kasetsart University, Bangkok, Thailand, in 2004, and the MS degree in computer science from the University of Louisiana, Lafayette, in 2008. From 2004 to 2006, he worked on Grid infrastructure establishment at Thai National Grid Center (TNGC), Thailand. He was also a visiting researcher at National Institute of Advanced Industrial Science and Technology (AIST), Japan, in 2004. Currently, he is a PhD candidate at Center for Advanced Computer Studies (CACS), the University of Louisiana, Lafayette. His research interests include peer-to-peer systems, cloud computing, and distributed systems.

Nain-Feng Tzeng (M'86-SM'92-F'10) has been with Center for Advanced Computer Studies, the University of Louisiana, Lafayette, since 1987. His current research interest is in the areas of computer communications and networks, high-performance computer systems, and parallel and distributed processing. He was on the editorial board of the *IEEE Transactions on Computers*, 1994-1998, and on the editorial board of the *IEEE Transactions on Parallel and Distributed Systems*, 1998-2001, and was elected to chair the Technical Committee on Distributed Processing, the IEEE Computer Society, from 1999 to 2002. He is the recipient of the outstanding paper award of the 10th International Conference on Distributed Computing Systems, May 1990, and received the University Foundation Distinguished Professor Award in 1997.