Energy Profile of Rollback-Recovery Strategies in High Performance Computing

Esteban Meneses, Osman Sarood and Laxmikant V. Kalé

Parallel Programming Laboratory
Department of Computer Science
University of Illinois at Urbana-Champaign

Abstract

Extreme-scale computing is set to provide the infrastructure for the advances and breakthroughs that will solve some of the hardest problems in science and engineering. However, resilience and energy concerns loom as two of the major challenges for machines at that scale. The number of components that will be assembled in the supercomputers plays a fundamental role in these challenges. First, a large number of parts will substantially increase the failure rate of the system compared to the failure frequency of current machines. Second, those components have to fit within the power envelope of the installation and keep the energy consumption within operational margins. Extreme-scale machines will have to incorporate fault tolerance mechanisms and honor the energy and power restrictions. Therefore, it is essential to understand how fault tolerance and energy consumption interplay. This paper presents a comparative evaluation and analysis of energy consumption in three different rollback-recovery protocols: checkpoint/restart, message logging and parallel recovery. Our experimental evaluation shows parallel recovery has the minimum execution time and energy consumption. Additionally, we present an analytical model that projects parallel recovery can reduce energy consumption more than 37% compared to checkpoint/restart at extreme scale.

Keywords: rollback-recovery, checkpoint/restart, message logging, parallel recovery, energy consumption

1. Introduction

The eventual arrival of extreme-scale supercomputers will help in solving some of the hardest problems in science and engineering. From high-resolution climate modeling to patient-specific drug design, many high-impact applications require a massive amount of computation that only very large systems can provide. However, there are at least two major challenges that have to be addressed to really make extreme-scale systems functional [1, 2]. The first problem is the significant increase in the failure frequency. As many components have to be assembled together to provide all the required computing power, large systems will inevitably have a high failure rate. It is estimated that exascale machines will have a failure every few minutes [2]. Therefore, a fault tolerance mechanism has to be employed to allow applications run on large systems. The second problem is power management and energy consumption. Power will be the driver in the design of architectures, systems and applications for extreme-scale computing. Installations will have strict power limits and all the layers of the system will have to meet that power budget. Energy will also be a crucial consideration, given the high cost of managing large systems. It will be fundamental to decrease the energy consumption. Reducing power consumption by one megawatt may save around \$1M/year even in a relatively inexpensive energy contract [2].

There are several promising fault-tolerance strategies to solve the resilience challenge at extreme scale. We present a set of three rollback-recovery protocols and offer a comparative evaluation and analysis in terms of their energy and power profiles. These protocols are organized as a hierarchy, where each protocol is an incremental extension of the previous. The base protocol is the traditional checkpoint/restart based on local storage [3, 4]. The tasks in the application periodically save their state and rollback to the latest global checkpoint in case of a failure. The next method is a particular version of message logging [5] that requires messages to be stored, but avoids a global rollback in case of a failure. Should a node fail, only the tasks running on that node are rolled back. The rest of tasks will re-send the messages to the failed tasks and make progress or wait idle until recovery is finished. Finally, the third approach is called parallel recovery [6] that extends message logging by allowing the migration of tasks after a failure to accelerate their recovery.

This paper extends the material presented in our previous publication [7] by refining the analytical formulation to model the energy consumption of

the different fault-tolerance protocols, extending the experimental results on new and more accurate power-measuring hardware, and improving the projections to extreme scale systems. The contributions of this paper are the following:

- An analytical model to understand and represent the energy consumption of three different rollback-recovery mechanisms (§ 3). This model incorporates the main factors that affect the power draw in each mechanism. At the same time, the model is flexible enough to be extended to more strategies and more factors.
- An experimental evaluation of the energy consumed by the three rollback-recovery techniques (§ 4). We present results using several programs on two different parallel programming models. These results were collected on a cluster enhanced to provide the power draw of various components at the millisecond level.
- Projections of the energy profile of the rollback-recovery strategies at extreme scale (§ 5). These estimations highlight the advantage of using local rollback protocols. For instance, on a machine with more than 512,000 sockets, the total energy consumed by a 24-hour job can be reduced by more than 37% using parallel recovery, compared to checkpoint/restart.

2. Rollback-Recovery

We conceive an application as a set of *tasks* Π. Each task holds a portion of the application's data and performs part of the computation. The only mechanism for the tasks to share information is through message passing. Each message is associated with a particular method at the target task. Upon reception of a message, a task executes the associated method until completion. An application is run on a parallel architecture that is represented as a set of *nodes* Σ. The number of tasks is independent of the number of nodes. The fraction $\frac{|\Pi|}{|\Sigma|}$ is called the *virtualization ratio*. All nodes are connected through a network that does not guarantee FIFO delivery between pairs of nodes. A runtime system orchestrates the execution of the application and is in charge of assigning tasks to nodes. In addition, the runtime system can migrate tasks from one node to another. Fault tolerance and load balancing are usually the main reasons for task migration. This computational model is general enough to accommodate well established parallel programming languages, such as MPI [8] and Charm++ [9].

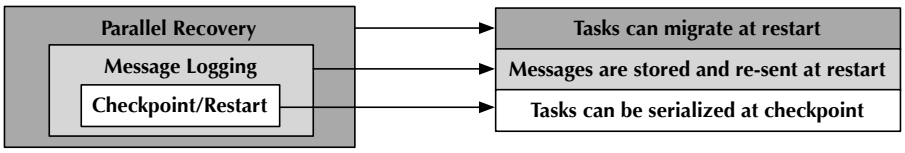


Figure 1: Organization of the three rollback-recovery protocols explored in this paper. Each protocol adds an incremental set of features that potentially reduces energy consumption in a faulty scenario.

The nodes in the system may fail according to the *fail-stop* model. That means, after a node crashes, it ceases to communicate and does not come back. Other node from a pool of spare nodes will replace the failed one. Therefore, the number of nodes dedicated to an application is constant during execution, regardless of the number of node failures. Nodes are assumed to be homogeneous and fail according to an exponentially distributed random variable. The system assembles all the nodes together and combines the individual resilience descriptors into a single value that is commonly referred as mean-time-between-failures (MTBF). This random variable represents the failure frequency of a machine. The runtime system must employ a fault-tolerance mechanism to allow an application run through failures. In this paper, we examine *rollback-recovery* techniques [10]. These protocols are based on the principle that a failure will force the system to roll back to a previously stored consistent state, and recover from that state. A typical realization of this principle is through checkpoint/restart, where the system periodically stores its state. Should a node fail, the system rolls back to the most recent valid checkpoint and restart.

This paper explores three rollback-recovery strategies: checkpoint/restart, message logging, and parallel recovery. They can be organized as a sequence of incremental additions to the fundamental checkpoint/restart scheme. Figure 1 shows the three protocols and the features each of them adds to reduce energy consumption when an application runs through failures.

2.1. Checkpoint/Restart

The most popular strategy to provide fault tolerance in HPC is checkpoint/restart. The basic tenet of this protocol is to checkpoint, or save the state of the application with certain periodicity, such that a failure will roll back all the nodes to a previous state and resume the execution from that state. The principle is simple and effective for the failure frequency of petas-

cale machines. There are several libraries implementing this protocol for HPC systems [3, 4, 11, 12].

There are at least three main approaches to determine what is included in a checkpoint [13]. In *system-level* checkpointing, the state of all the system is stored. That includes the memory pages used by the application, but also the rest of the hardware state (register, buffers, caches, etc). In contrast, *application-level* checkpointing allows the programmer to write a checkpoint method and define exactly what is saved. This approach enforces a more disciplined use of the checkpoint calls, by placing them at convenient locations in the code (for instance, after global synchronization operations). Finally, *runtime-system-based* checkpointing extends the previous approach by having the runtime system automatically store the state of some structures. The programmer has to implement the checkpoint routines and make the checkpoint calls. However, there is a major role played by the runtime system at checkpoint time. In some situations, the runtime system may decide not to checkpoint if the failure frequency is lower than a threshold.

A checkpoint is called *uncoordinated* if the tasks are allowed to checkpoint at their own pace. Such strategy does not incur any overhead in synchronizing the tasks for checkpoint, but the collection of checkpoints from all the tasks may not be consistent. To alleviate that problem, either some messages have to be stored, or there is a risk of having cascading rollbacks. This last situation means that a failure may roll back the system several checkpoints until a valid collection of checkpoints is found. A *coordinated* checkpoint, on the contrary, requires the set of tasks to collaborate in deciding the time to checkpoint or what to include in the checkpoint. Alternatively, the programmer can use globally synchronization points in the application to trigger the checkpoint. This way, the set of checkpoints form a consistent global checkpoint that can be safely used to restart from a failure.

There are several alternatives for the checkpoint storage. The traditional approach has been to use the network shared file system. However, it is clear this scheme will not scale very far and it quickly bottlenecks [1, 14, 15]. Therefore, an alternative approach is to use storage local to the nodes. One strategy is called *double-local* checkpoint/restart [3], where either memory, local disks or solid state drives (SSDs) are used to save the checkpoint. This strategy requires each node to have a checkpoint *buddy*. At checkpoint, every node saves its state in its own local storage and in the local storage of its buddy. If a node crashes, the buddy provides the latest checkpoint to the replacement node. All other nodes in the system roll back to the latest

checkpoint stored in their local storage.

In the rest of the paper we assume checkpoints are runtime-system-based, coordinated and double-local. We build the next two fault tolerance strategies on top of this checkpoint/restart scheme.

2.2. Message Logging

A node failure provokes the loss of the current state of the tasks running on the failed node. To reconstruct that state, checkpoint/restart rolls back the set of all tasks to a previous consistent checkpoint and resumes execution. Rolling back the tasks that did not fail is unnecessary, provided that the messages those tasks sent to the failed tasks can be somehow recovered. This is the spirit of message logging. In principle, all messages between tasks are stored and replayed in case of a failure. That means, only the tasks on the failed node have to roll back, the rest of the system is free to keep making progress or to wait idle. Message logging provides *local* recovery, as opposed to *global* recovery in checkpoint/restart. This ability has a high energy-saving potential.

In addition to store messages between tasks, message-logging protocols usually require more mechanisms to work properly. To recover the failed node correctly (i.e., bring its state to a consistent global state with the rest of nodes), it is often necessary that the recovering node processes the messages in the same order as it did before the crash. Even more, all non-deterministic decisions must be executed with the same output during recovery. We use the piece-wise deterministic (PWD) assumption [16] that states saving all the output of the non-deterministic events is sufficient for a correct recovery. These bits of information are called *determinants*. The only type of non-determinism we consider in this paper is message reception. Thus, every time a message is received, a determinant gets created. There are different protocols to handle determinants and how they get stored and recovered. The major message-logging protocols are called optimistic, pessimistic and causal [17]. In this paper, we will use a variant called *simple causal message logging* [5]. Additionally, we will also used the *fast message logging* protocol [15] for programs with a high-level representation that allows the suppression of determinants. The major concern in designing a message-logging protocol is to keep small the performance overhead μ associated with managing messages and determinants. The value of μ depends on several factors. However, for the applications used in this paper, the aforementioned protocols have a μ lower than 5%.

2.3. Parallel Recovery

A useful extension to message logging that aims to further reduce recovery time is called *parallel recovery* [6]. The main idea of this strategy consists in distributing the tasks on the failed node among other nodes during restart. That way, if the application is tightly coupled and most nodes are idle during recovery, the recovering tasks can recover on different nodes and in parallel. A faster recovery provides a mechanism to achieve a faster execution in a faulty environment. Therefore, the acceleration factor during recovery σ aims to offset the performance overhead of message logging μ.

The design of parallel recovery assumes tasks can only migrate during checkpoint or restart[1]. That means parallel recovery will accelerate a portion of the execution and slow down another portion. More explicitly, if the checkpoint period is τ and the failure occurs t units after the last checkpoint, then parallel recovery accelerates the recovery of the portion $\frac{t}{\tau}$, but incurs a performance penalization λ in the portion $\frac{\tau-t}{\tau}$. The factor λ appears thanks to the task migration for recovery. Some nodes will receive tasks to recover and these tasks can only go back to their origin node until the next checkpoint is reached.

Figure 2 presents a sample execution and how recovery works with the three different protocols examined in this paper. Part a) shows a system with 4 nodes and 6 tasks. This scenario depicts a failure on node Y after a few messages have been exchanged from the last checkpoint. Part b) presents the recovery using checkpoint/restart, where all the nodes roll back to the previous checkpoint and resume execution. Note that recovery may proceed differently than the original execution. Part c) shows message logging that only requires node Y to roll back. In this case, the same delivery sequence has to be reproduced. Determinants provide such guarantee. Part d) shows parallel recovery that migrates task D to node X, accelerating the recovery of tasks D and E.

3. Modeling Energy Consumption

Rollback-recovery mechanisms based on checkpoint/restart, like the three protocols described in the previous section, face an important challenge re-

[1]This restriction is used to build a more efficient implementation and does not come from the computational model.

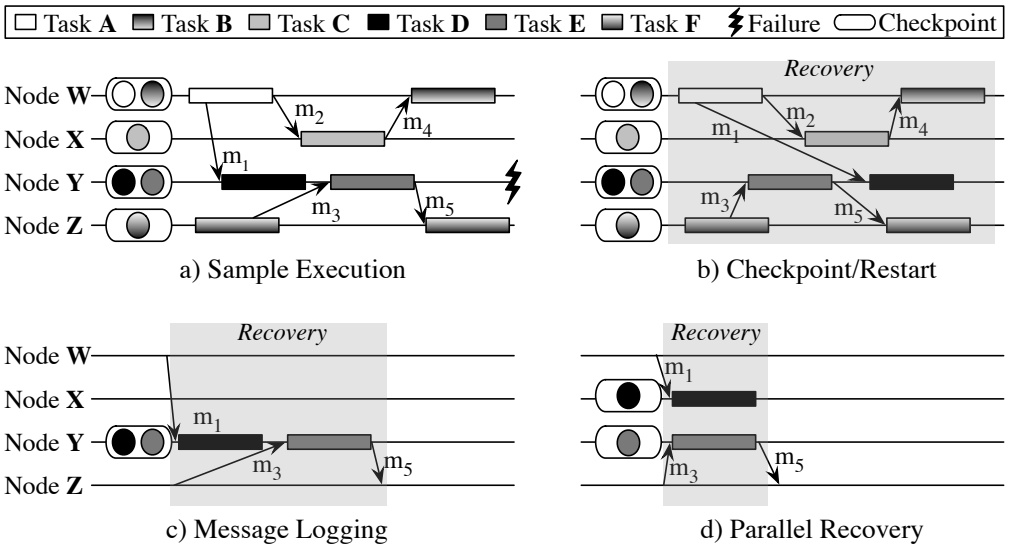


Figure 2: A sample execution and recovery with the three rollback-recovery schemes.

garding the checkpoint frequency. If checkpoints are too frequent, the overhead of saving the state of the application will negatively impact the performance. Contrarily, if checkpoints are too infrequent, the amount of work to recover after a failure may even prevent the system from making any progress. Therefore, an optimal checkpoint period must be found. The symbol τ will denote the amount of work that must be performed between two consecutive checkpoints. For failures exponentially distributed, an optimal τ is periodic [18]. The optimal value of τ depends on other factors, including checkpoint time δ, and the MTBF of the system M. A popular approximation [19] is given by the formula $\tau = \sqrt{2\delta M} - \delta$. This expression provides the optimal τ to reduce the overall execution time, considering the failures that may occur during execution.

However, in minimizing the energy consumption of rollback recovery protocols, it is important to understand how execution time and energy consumption are related and how they differ. Figure 3 shows an example run of a stencil code that checkpoints its state to memory. The figure reports the total power on one node into three components: base, CPU and memory. The power reading granularity is one tenth of a second. The application periodically checkpoints its state, and at those points there is a visible decrease

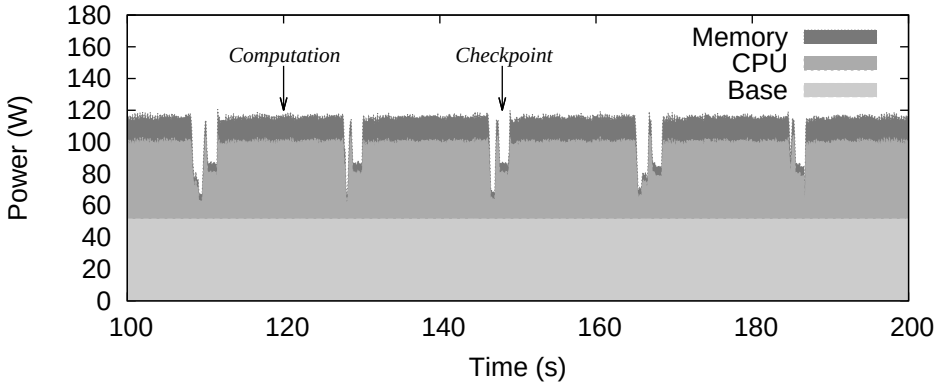


Figure 3: Different power levels in a sample execution with checkpoint/restart.

in the power of both CPU and memory. The whole execution alternates between two different power levels, one for the computation and one for the checkpoint. Therefore, the performance models to minimize execution time in rollback-recovery protocols can not directly apply to energy consumption, because of the different power levels at which computation and checkpoint happen. In addition, recovery will introduce another source for power level variation. While checkpoint/restart recovers at full power, message logging and parallel recovery dramatically decrease the power level because they only require a local recovery. Traditional performance models assume a program executes at a constant power level. They would provide a suboptimal checkpoint frequency if energy consumption has to be minimized instead of execution time. For instance, the optimum checkpoint frequency for energy consumption will be greater or equal to the optimum checkpoint frequency for execution time.

Symbol	Description	Symbol	Description
M_S	MTBF of each socket	μ	Message-logging slowdown
M	MTBF of the system	ϕ	Message-logging recovery speedup
S	Total number of sockets	P	Available parallelism during recovery
W	Time to solution	λ	Parallel recovery slowdown
δ	Checkpoint time	σ	Parallel recovery speedup
τ	Optimum checkpoint period	ψ	Migration cost
R	Restart time	H	High power of each socket
T	Total execution time	L	Low power of each socket
E	Total energy consumption		

Table 1: Parameters of the energy consumption model.

We introduce an analytical model to estimate the total energy consumption of rollback-recovery mechanisms. The model requires several parameters, which are listed in Table 1. Usually, M_S, M, S, W, δ and R are inputs to the model. For simplicity, we will assume $M = \frac{M_S}{S}$. The model determines the optimal checkpoint period τ to either minimize T or E. We will denote by τ^T and τ^E the optimum value of τ to minimize time and energy, respectively. Message logging parameters are μ and ϕ, while parallel recovery is represented by P, λ, σ, and ψ. Finally, the model considers two power levels, H and L, depending on whether the system is doing computation or checkpointing, respectively. We will assume L is also the idle power. Here are the general formulas for execution time and energy consumption:

$$T = T_{Solve} + T_{Checkpoint} + T_{Recover} + T_{Restart} \quad (1)$$

$$E = E_{Solve} + E_{Checkpoint} + E_{Recover} + E_{Restart} \quad (2)$$

These formulas are separated into four different components, according to the state in which the execution is. For each of the rollback-recovery protocols, we provide approximations for each of the components in equations 1 and 2. Table 2 shows the equations for T_C, T_M, and T_P, representing checkpoint/restart, message logging and parallel recovery, respectively. Similarly, the table lists the formulas for E_C, E_M, and E_P.

Protocol	Formulas
Checkpoint/ Restart	$T_C = W + \left(\frac{W}{\tau} - 1\right)\delta + \frac{T_C}{M}\left(\frac{\tau+\delta}{2}\right) + \frac{T_C}{M}R$ $E_C = WSH + \left(\frac{W}{\tau} - 1\right)\delta SL + \frac{T_C}{M}\Omega_C + \frac{T_C}{M}RSL$ $\Omega_C = \frac{\tau}{\tau+\delta} \cdot \frac{\tau}{2}SH + \frac{\delta}{\tau+\delta}(\tau SH + \frac{\delta}{2}SL)$
Message Logging	$T_M = W\mu + \left(\frac{W\mu}{\tau} - 1\right)\delta + \frac{T_M}{M}\left(\frac{\tau}{\tau+\delta} \cdot \frac{\tau}{2\phi} + \frac{\delta}{\tau+\delta}\left(\frac{\tau}{\phi} + \frac{\delta}{2}\right)\right) + \frac{T_M}{M}R$ $E_M = W\mu SH + \left(\frac{W\mu}{\tau} - 1\right)\delta SL + \frac{T_M}{M}\Omega_M + \frac{T_M}{M}RSL$ $\Omega_M = \frac{\tau}{\tau+\delta} \cdot \frac{\tau}{2\phi}(H + (S-1)L) + \frac{\delta}{\tau+\delta}(\frac{\tau}{\phi}(H + (S-1)L) + \frac{\delta}{2}SL)$
Parallel Recovery	$T_P = W\mu + \left(\frac{W\mu}{\tau} - 1\right)\delta + \frac{T_P}{M}\left(\frac{\tau}{\tau+\delta}\left(\frac{\tau}{2\sigma} + \frac{\tau}{2}(\lambda-1)\right) + \frac{\delta}{\tau+\delta}\left(\frac{\tau}{\sigma} + \frac{\delta}{2}\right)\right) + \frac{T_P}{M}(R+\psi)$ $E_P = W\mu SH + \left(\frac{W\mu}{\tau} - 1\right)\delta SL + \frac{T_P}{M}\Omega_P + \frac{T_P}{M}(R+\psi)SL$ $\Omega_P = \frac{\tau}{\tau+\delta}\left(\frac{\tau}{2\sigma}(PH + (S-P)L) + \frac{\tau}{2}(\lambda-1)SH\right) + \frac{\delta}{\tau+\delta}\left(\frac{\tau}{\sigma}(PH + (S-P)L) + \frac{\delta}{2}SL\right)$

Table 2: Execution time and energy consumption formulas.

The equations in Table 2 assume the number of failures in an execution can be approximated by $\frac{T}{M}$. The equations for T_C, T_M, and T_P all have the similar expressions for T_{Solve} and $T_{Checkpoint}$. The only difference comes

from the associated overhead of message logging. These formulas, though, markedly differ in the expression for $T_{Recover}$. Message logging uses ϕ to represent the speedup in recovery, while parallel recovery adds σ and λ during recovery. The migration cost of distributing tasks in parallel recovery is captured by ψ and added to the restart cost $T_{Restart}$. The equations for E_C, E_M, and E_P extend their execution time counterparts by adding appropriate power levels to each component.

4. Experimental Evaluation

4.1. Setup

The testbed used in this paper is called the Energy Cluster. It has two sections, named A and C. Section A contains 32-nodes (128 cores). Each node has a single socket with a four-core Intel Xeon X3430 processor chip running CentOS 5.7. Section C is a 20-node Dell PowerEdge R620 cluster. Each node has a single socket containing the Intel Xeon E5-2620 Sandy-bridge server with 6 physical cores @ 2GHz, 2-way SMT with 16 GB of DRAM. The Package (CPU) corresponds to the processor die that includes the cores, L1, L2, and L3 caches in addition to the memory controller. This processor supports on board power measurement and capping through Intel's Running Average Power Limit (RAPL) interface [20]. RAPL is implemented using a series of Machine Specific Registers (MSRs). These MSRs can be accesses to get power readings for each of the supported power planes at a granularity of 1 millisecond. Any power consumed other than the package and DRAM domains is referred as the base power. It includes the power consumed by all the remaining components (e.g. power supply, mother board, fans and network cards). The base power of a machine does not fluctuate a lot. For this cluster, the base power for each of the nodes came out to be between 38-44W. These base power measurements are taken from the built-in power meters on the Power Distribution Unit (PDU) that powers the cluster. Each section of the Energy Cluster is connected through a gigabit Ethernet switch.

The three fault-tolerance protocols presented in the previous sections were implemented in the Charm++ runtime system [21]. Charm++ implements a parallel programming paradigm called *migratable objects* that fits into our computational model of Section 2. In Charm++, an application is *over*-decomposed into objects. The number of objects is usually independent of the number of nodes on which the application will run. An adaptive runtime system allows Charm++ distribute the objects among the nodes and

migrate objects from one node to another. That migration is traditionally used to bring a better load balancing to the execution, reducing the total time to completion. An extension to Charm++, called Adaptive Message Passing Interface (AMPI) permits the execution of any MPI application on the Charm++ runtime. In AMPI each MPI rank becomes an object that enjoys all the benefits of Charm++, such as load balancing, fault tolerance and general adaptivity of the system.

Table 3 presents a summary of all the applications used in the experimental evaluation of the different protocols. The program *Wave2D* runs a finite difference method to compute pressure information on a two-dimensional grid. *Jacobi3D* is a 7-point stencil that computes the transmission of heat on a three-dimensional space. The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (*LULESH*) is a code for modeling hydrodynamics. This code describes the motion of materials relative to each other when subject to forces in a three-dimensional space. The MPI programs *NPB-BT* and *NPB-SP* correspond to the block-tridiagonal and scalar penta-diagonal kernels from the NAS Parallel Benchmarks (NPB). To inject a failure in a running application, we execute `kill -9 PID`, where PID represents a process running on one physical core with multiple tasks.

Application	**Wave2D**	**Jacobi3D**	**LULESH**	**NPB-BT**	**NPB-SP**
Language	Charm++	Charm++	Charm++	MPI	MPI
Domain	Physics	Physics	Physics	Linear Algebra	Linear Algebra
Virtualization Ratio	32	32	32	4	4
Max Power (C/R)	108	103	105	102	95
Max Power (ML)	103	103	105	102	96

Table 3: List of features of applications used in the experiments.

4.2. Results

We ran *Jacobi3D* with the three protocols and analyze the power and energy profile of each run. The program was run over a space of size $1500 \times 800 \times 400$, decomposed into blocks size 50^3 for a virtualization ratio of 32. The program ran for a total of 1,600 iterations with checkpoints at iterations 400 and 1,200. A failure on one core was injected at time 41 seconds. We used all the 20 nodes and 120 physical cores of the Energy Cluster Section C. Figure 4 presents the results of this experiment. On the left column we show the *progress diagram* of the execution, that shows the time at which certain iteration was completed. The diagram for checkpoint/restart illustrates a

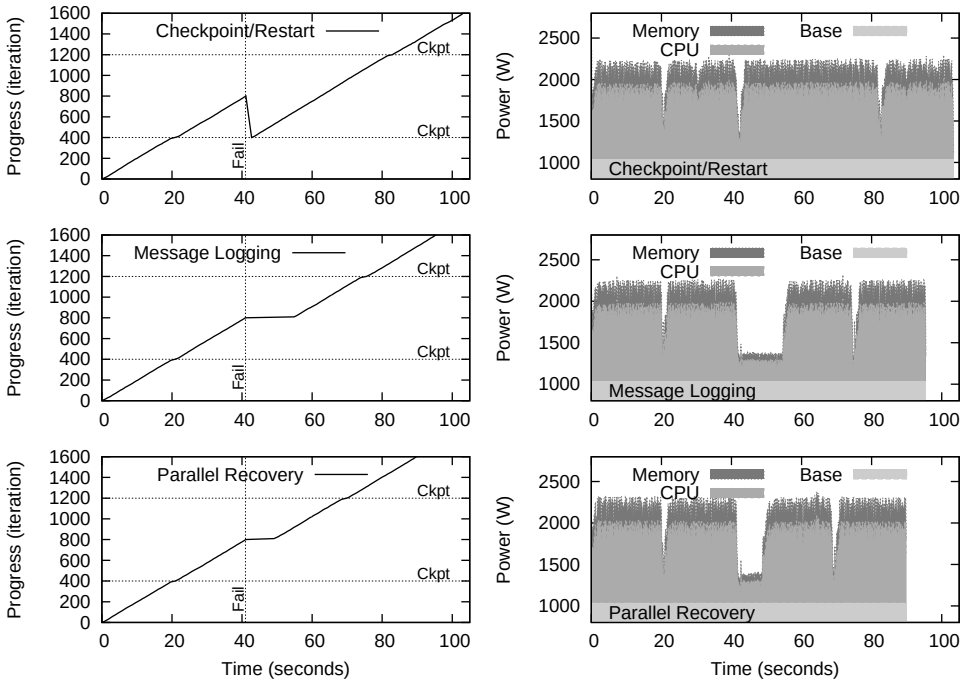


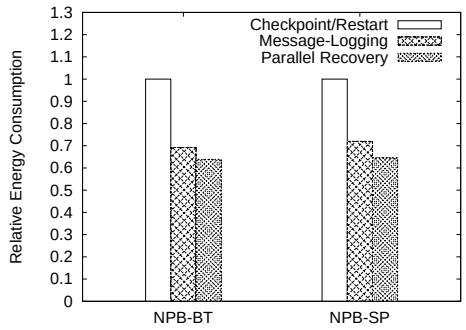
Figure 4: Progress rate of different rollback-recovery strategies in a faulty scenario.

global recovery mechanism that rolls all the tasks back to the latest checkpoint. In contrast, message-logging techniques only force a local recovery of the failed tasks. The flat section of the progress rate in message logging and parallel restart corresponds to the recovery of the work lost in a failure. In this case, parallel recovery used a value for P equals to 4. The recovery speedup achieved was 3.74. The final execution time for checkpoint/restart, message logging, and parallel recovery was 103.13 s, 95.34 s, and 89.82 s, respectively.

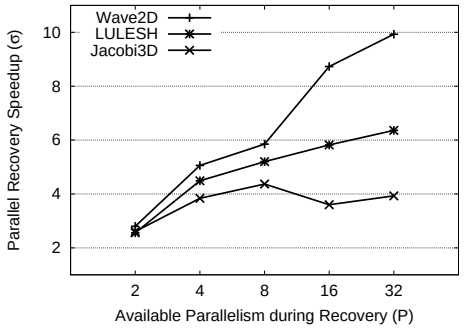
The right column in Figure 4 shows the total power of the machine for the same execution. During recovery, checkpoint/restart does not reduce the power level, because recovery is global. However, message logging and parallel recovery dramatically decrease their power consumption because the rest of tasks on other nodes are waiting idle while the failed tasks catch up. That alone decreases the total energy consumption and in the end checkpoint/restart, message logging and parallel recovery use 218.96 kJ, 193.96 kJ, and 190.97 kJ, respectively. The values predicted by the energy consumption model of Section 3 for this particular scenario are 220.69 kJ, 195.65 kJ,

and 191.45 kJ, for checkpoint/restart, message logging, and parallel recovery, correspondingly. All projected values lie within 1% error margin with respect to the experimental values. Note that message logging outperforms checkpoint/restart even considering the message logging slowdown (μ). This is the result of message logging having a speedup during recovery (ϕ) due to the immediate availability of messages while recovering.

We ran the MPI programs using the AMPI extension on 25 nodes (100 cores) of the Energy Cluster Section A. In both cases (BT and SP), the program used periodic checkpoints. A failure was inserted in the middle of one checkpoint period. Each failure point was carefully calibrated for each protocol. We measured the total energy consumption for each protocol in the faulty interval. The results are shown in Figure 5(a) and it presents the energy consumption of message logging and parallel recovery relative to checkpoint/restart. Parallel recovery manages to execute through the failure with the minimum amount of energy consumed. For this particular case, both benchmarks show similar results, with message-logging using around 70% of checkpoint/restart and parallel recovery using close to 63%.



(a) Energy consumed in a faulty checkpoint period.



(b) Parallel recovery speedup.

Figure 5: Potential for parallel recovery. The higher the virtualization ratio, the more room for acceleration during recovery and less energy consumed.

Finally, we ran the three CHARM++ programs on 120 cores of the Energy Cluster Section C to understand how much speedup is achievable with different applications. Figure 5(b) shows the speedup in recovery (σ) and the available parallelism P increases. Some programs benefit more than others from parallel recovery. *Wave2D* shows the highest speedup, reaching

almost a factor of 10. *LULESH* and *Jacobi3D* reach up a factor of 6 and 4, respectively.

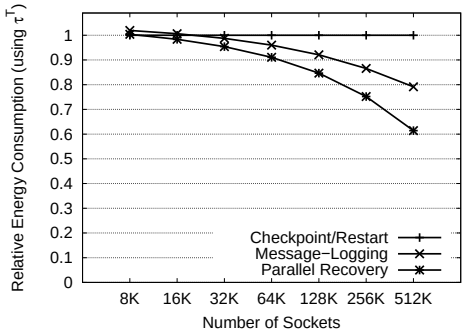
5. Extreme-Scale Projections

The previous section highlighted the potential of message logging and parallel recovery in decreasing the energy consumption of an execution. In this section we provide a set of projections of how the fault-tolerance protocols would perform in different circumstances. The most fundamental factor is scalability. Hence, we offer figures for a varying number of sockets as the measure for system size. The socket count ranges from 8,192 to 524,288, given that a large-scale machine is expected to have at least 200,000 sockets [1]. The parameters of the analytical model of Section 3 are materialized with appropriate values, shown in Table 4. We simulate a job running for 24 hours and weak scaling through the socket count range. Each socket has an MTBF of 10 years, a reasonable value according to literature estimates [1, 15, 22]. The checkpoint and restart time are based on the algorithm described in Section 2 and the match expectations at large scale [1]. The parameters for message logging and parallel restart are based on empirical evidence we have collected [5, 7, 15]. Finally, the power levels H and L are based on the experimental results of Section 4. Although the values of these last two parameters directly affect the performance of the different protocols, their relative value is what matters the most for the contrast we will present.

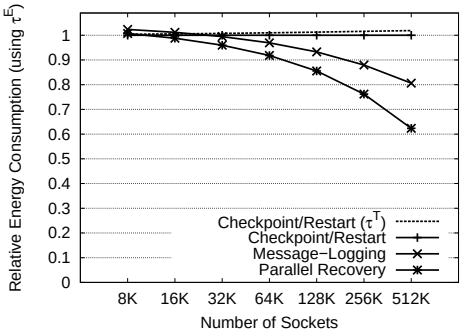
Parameter	W	M_S	δ	R	μ	ϕ	P	σ	λ	ψ	H	L
Value	24h	10 years	180s	30s	1.05	1.2	8	P	$\frac{P+1}{P}$	$\frac{\delta}{P}$	100W	50W

Table 4: Baseline values of parameters in the model.

The first comparison we present appears in Figure 6 where the relative energy consumption between the three protocols is shown. Figure 6(a) shows the energy consumption using the optimum checkpoint period to minimize execution time, τ^T. Similarly, Figure 6(b) presents the same comparison but using τ^E, the optimum checkpoint period to minimize energy consumption. Both figures show the scalability of message logging and parallel restart. A larger socket count brings higher benefits in terms of energy consumption. That means, strategies based on local recovery can tolerate higher failure frequencies. In Figure 6(a), checkpoint/restart only performs better at 8,192 sockets. At the extreme end of the scale, message logging manages to reduce



(a) Energy consumption using τ^T. Both message logging and parallel recovery perform better than checkpoint/restart after 16,384 sockets.



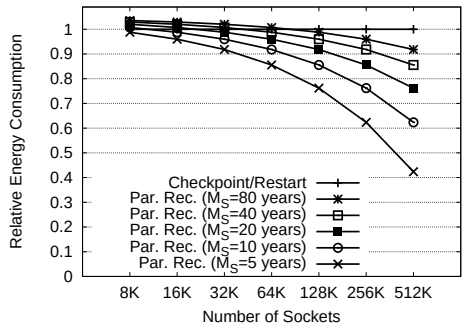
(b) Energy consumption using τ^E. Similar to the τ^T counterpart, message logging and parallel restart manage to bring big benefits compared to checkpoint/restart.

Figure 6: Comparison of energy consumption of fault-tolerance methods. Message logging and parallel recovery offer an energy efficient solution as systems scale. When compared to checkpoint/restart, message logging can reduce energy consumption more than 19%, and parallel recovery more than 37%, using τ^E.

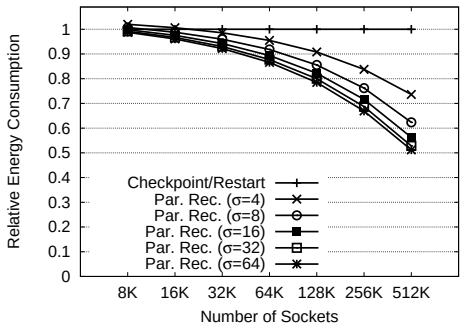
energy consumption by more than 20%, whereas parallel recovery achieves more than 38% reduction. Using τ^E does not fetch any additional significant benefit for message logging and parallel restart in this scenario (19% and 37%, respectively), as depicted in Figure 6(b). For comparison purposes, the original τ^T checkpoint/restart curve is plotted, presenting just a marginal difference with its τ^E counterpart.

One relevant aspect of the implications of failures on HPC systems comes from the reliability standards of chip manufacturers. Figure 7(a) shows the relative energy consumption of parallel recovery with different values for M_S, ranging from 5 to 80 years. A higher value of M_S decreases the benefit of parallel recovery because it reduces the failure rate and the potential for parallel recovery to consume less energy during recovery. A drop in socket reliability from 10 to 5 years causes a dramatic increase in energy consumption benefits of parallel recovery for roughly 20%. At that point, energy consumed can be reduced by more than a half avoiding traditional checkpoint/restart and using accelerated recovery.

Figure 7(b) presents the difference parallel recovery makes on energy consumption. Greater values of σ improve the energy consumption of parallel recovery, but there are diminishing returns for values greater than 16. That



(a) Effect of different values of M_S on energy consumption. The higher the failure rate, the more relative benefits of parallel recovery with respect to checkpoint/restart.

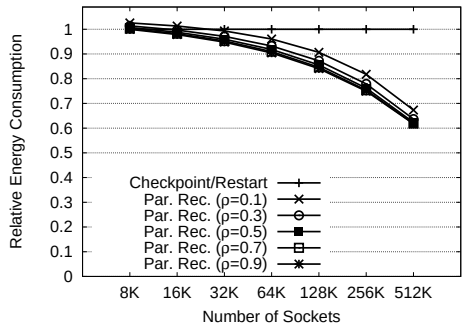


(b) Effect of different values of σ on energy consumption. The more speedup during parallel recovery, the higher the benefits.

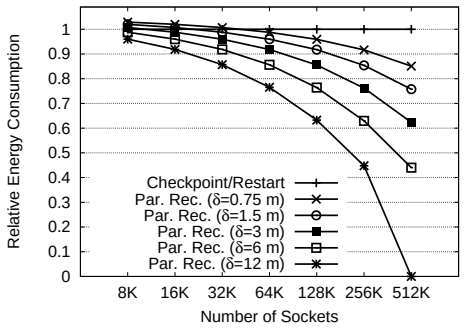
Figure 7: Factors affecting the performance of parallel recovery. Less reliable chips benefit parallel recovery. A drop of M_S from 10 to 5 years increases parallel recovery's benefits by 20%. A parallel speedup value of 16 is good enough to maintain increased benefit along the range of socket count.

means, achieving an acceleration factor of 16 in parallel recovery should be enough to provide a benefit close to the maximum possible. In addition, all the different values of σ show scalability across the socket count range. Therefore, it is not necessary to scale the parallel speedup as the system grows to keep the same benefit.

The ratio between base and maximum power, denoted by ρ, is worth exploring. Intuitively, the smaller the value of ρ, the more benefits message logging and parallel recovery should have over checkpoint/restart. That is based on the local recovery ability of these two protocols. If the difference between staying idle and executing is higher, then there is more room to reduce energy consumption. Figure 8(a) presents several values for ρ and the relative energy consumption for each value using parallel recovery. Interestingly, the difference is marginal across the different values of ρ. A smaller value of ρ simply forces checkpoint/restart to checkpoint more frequently and keep energy consumption to a minimum. However, a higher checkpoint rate implies a higher checkpoint overhead and ultimately a higher execution time. Therefore, even when checkpoint/restart manages to maintain a similar figure with different values of ρ, it sacrifices execution time for a smaller value of ρ.



(a) Effect of different values of ρ. There is little difference between the different values, but it comes at a cost of higher execution time in checkpoint/restart.



(b) Effect of different checkpoint duration. A greater checkpoint time increases the benefit of parallel recovery.

Figure 8: Understanding the effect of different parameters in the model and its overall benefit is fundamental in deciding what factors should be explored when designing fault tolerance mechanisms. The ratio of base to maximum power does not have a high impact on energy consumption, but it does on execution time. Higher values of checkpoint time may cause checkpoint/restart to collapse.

Figure 8(b) explores the relative benefit of parallel restart with different checkpoint durations, ranging from 0.75 minutes to 12 minutes. If the value of δ grows, so does the checkpoint period τ. That means, a failure will make the system recover more work. Since parallel recovery benefits of more work to recover, then it has a better contrast to checkpoint/restart. At the extreme scale, a checkpoint time of 12 minutes causes checkpoint/restart to collapse, being unable to make progress in that case. Parallel recovery can still finish execution.

6. Discussion

This paper aims to provide some insight about the interaction between fault tolerance and energy consumption. Providing effective resilience at extreme scale is imperative. However, different mechanisms show a variety of power and energy profiles. These differences help to disprove one traditional thinking about energy consumption: *minimize execution time will minimize energy consumption*. That statement is not true, at least in a faulty environment. It heavily depends on what fault tolerance mechanism is in place and under what conditions. The analytical model presented in Section 3

provides the theoretical foundations to understand why. For instance, message logging incurs performance overhead that may lead to longer execution times compared to checkpoint/restart. But, message logging recovers from a failure using a fraction of the energy checkpoint/restart uses. If the failure rate is not too high, message logging may have longer execution time but less energy consumed than checkpoint/restart [15].

Despite the fact that message logging techniques (including parallel recovery) have to deal with message storage and determinism management, these protocols do not significantly increase the power draw. Figure 4 shows on the right column the power levels of the three protocols compared in this paper and there is no evidence message logging draws more power than checkpoint/restart. The extra operations performed by message logging have an impact on performance, though. That effect is captured by parameter μ in the performance model of Section 3. This finding is important to ensure message logging and parallel restart will honor the power limitations and not push the power envelope beyond the capacity of the installation.

The ratio between idle and maximum power plays a crucial role in the relation between execution time and energy consumption. Figure 8(a) shows the effect on energy consumption of different base to maximum power ratios. Although checkpoint/restart manages to keep relatively the same difference with respect to parallel recovery, it does it by increasing the checkpoint frequency and sacrificing execution time. A smaller value of this ratio benefits message logging and parallel restart, because these two protocols exploit that ratio to decrease energy consumption. Table 5 presents a list of different architectures and an approximation of their respective ratios. For the Intel chips, we experimentally measured the power levels running *Wave2D*. The NVIDIA data can be found elsewhere [23]. The goal to decrease the base power has been a constant in the design of new chips. That leads to a smaller base to maximum power ratio that can be better incorporated by message logging and parallel recovery. Similarly, a GPU architecture has a small ratio and hybrid architectures (processors and accelerators) will be more appealing to protocols that are based on local recovery. Our performance model directly applies to hybrid architectures.

Figure 7(b) shows the effect of increasing the parallel recovery speedup. This ability comes from the fact that the application can be *over-decomposed* into small chunks of work. Parallel programming paradigms that allow this type of decomposition will be able to accelerate recovery and decrease energy consumption. What is more important is that the parallelism does not need

Architecture	Release Date	Base Power	Max Power	Base/Max Ratio
Intel Xeon E5520	Q1,09	60	125	0.48
Intel Nehalem i7 860	Q3,09	52	151	0.34
Intel Sandy Bridge i7 2600	Q1,11	21	101	0.21
NVIDIA GTX280	Q2,08	56	224	0.25

Table 5: Comparison of base and maximum power for different architectures.

to scale with the system size. In fact, a parallelism degree of 16 seems to be a good value to provide big benefits. Additional parallelism only brings a marginal benefit.

The type of failures this paper deals with is one-node failures. Although this might look a little restrictive, there is enough evidence a vast majority of failures only affect one node [4, 24]. A different type of failures that may become more relevant in the future is *correlated* failures, where several components fail in tandem given the particular architecture of the machine. For instance, nodes plugged to the same power supply will fail together given a failure of the common power supply. These type of considerations can be incorporated into a message-logging protocol to decrease the protocol's overhead and further decrease energy consumption [15].

7. Related Work

Literature on the interplay of energy consumption and fault tolerance for HPC is scarce. To the best of our knowledge, the work by Diouri *et al* [25, 26] is the only related work. In a first study [25], they presented the power draw and energy consumption of three building blocks of fault tolerance protocols: checkpointing, task coordination, and message logging. Their results show that neither of these tasks significantly increases the power draw of a node. However, message logging increases the total energy consumed by a program due to the overhead it incurs. Their initial work was later extended into ECOFIT [26], a framework for predicting the energy consumption of an application using certain fault tolerance protocol on a particular architecture. ECOFIT calibrates the power of each of the four fundamental operations (checkpointing, coordination, logging, and recovery) and then estimates the total number of these operations in an execution. They evaluated three types of protocols: coordinated, uncoordinated and hierarchical.

Our philosophy differs from theirs in that we consider coordinated protocols are the most viable way for fault tolerance in HPC. In particular, coordinated application-level checkpoint is supported by most of the fault tolerance libraries available for HPC [3, 4, 12]. The advantages of such checkpoint variant are a smaller checkpoint size (because often time it is possible to checkpoint when the state of the application is minimal), and a low cost at checkpoint (since most HPC applications have global synchronization points that hide the coordination cost). In addition, we emphasize the importance of recovery, both experimentally and analytically. The more efficient the recovery, the better the fault tolerance protocol. Parallel recovery is a good example of that.

8. Conclusions

This paper presents a comparative evaluation of three rollback-recovery mechanism according to their energy profile. We present an analytical model to describe the energy consumption of each protocol and to make projections for large-scale systems under different conditions. We also show experimental results that support our model predictions.

We conclude the following:

- The reasoning *minimize execution time will minimize energy consumption* is invalid in the context of faulty machines. Message logging is a good example of that. It incurs performance overhead due to the additional function it performs (storing messages and determinants), but recovery is significantly more efficient in energy terms.
- Neither message logging nor parallel recovery significantly increase power draw. Our empirical results support that claim. These protocols may, though, increase energy consumption in a failure-free execution due to the performance overhead of the message logging protocol.
- Parallel recovery can reduce both execution time and energy consumption in a faulty scenario. It achieves that by accelerating recovery through parallel re-execution of tasks. In an HPC environment, parallel recovery satisfies the requirements of both users (minimum execution time) and system administrators (minimum energy consumption).
- The analytical model predicts a substantial reduction in energy consumption by using parallel recovery. For a large-scale system with more than 512,000 sockets, parallel recovery will be able to reduce the total energy consumption by more than 37%, compared to checkpoint/restart.

Acknowledgments

This research was supported in part by the US Department of Energy under grant DOE DE-SC0001845. We thank Prof. Tarek F. Abdelzaher of the University of Illinois at Urbana-Champaign for granting us access to the testbed used in this paper.

References

[1] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, Exascale computing study: Technology challenges in achieving exascale systems (2008).

[2] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, D. Barkai, T. Boku, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, R. Fiore, A. Geist, R. Harrison, M. Hereld, M. Heroux, K. Hotta, Y. Ishikawa, Z. Jin, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, B. Lucas, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. Mueller, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, A. V. D. Steen, J. Vetter, P. Williams, R. Wisniewski, K. Yelick, The international exascale software project roadmap 1.

[3] G. Zheng, L. Shi, L. V. Kalé, FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI, in: 2004 IEEE Cluster, San Diego, CA, 2004, pp. 93–103.

[4] A. Moody, G. Bronevetsky, K. Mohror, B. R. de Supinski, Design, modeling, and evaluation of a scalable multi-level checkpointing system, in: SC, 2010, pp. 1–11.

[5] E. Meneses, G. Bronevetsky, L. V. Kale, Evaluation of simple causal message logging for large-scale fault tolerant HPC systems, in: 16th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems in 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011)., 2011.

[6] S. Chakravorty, L. V. Kale, A fault tolerance protocol with fast fault recovery, in: Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium, IEEE Press, 2007.

[7] E. Meneses, O. Sarood, L. V. Kale, Assessing Energy Efficiency of Fault Tolerance Protocols for HPC Systems, in: Proceedings of the 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2012), New York, USA, 2012.

[8] S. O. Marc Snir, etc, MPI: The Complete Reference, Vol. 1, The MIT Press, 1998.

[9] L. Kalé, S. Krishnan, CHARM++: A Portable Concurrent Object Oriented System Based on C++, in: A. Paepcke (Ed.), Proceedings of OOPSLA'93, ACM Press, 1993, pp. 91–108.

[10] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, D. B. Johnson, A survey of rollback-recovery protocols in message-passing systems, ACM Comput. Surv. 34 (3) (2002) 375–408.

[11] P. H. Hargrove, J. C. Duell, Berkeley lab checkpoint/restart (blcr) for linux clusters, in: SciDAC, 2006.

[12] L. Bautista-Gomez, D. Komatitsch, N. Maruyama, S. Tsuboi, F. Cappello, S. Matsuoka, FTI: High performance fault tolerance interface for hybrid systems, in: Supercomputing, 2011, pp. 1 –12.

[13] M. Schulz, Checkpointing, in: Encyclopedia of Parallel Computing, 2011, pp. 264–273.

[14] E. N. Elnozahy, R. Bianchini, T. El-Ghazawi, A. Fox, F. Godfrey, A. Hoisie, K. McKinley, R. Melhem, J. S. Plank, P. Ranganathan and J. Simons, System resilience at extreme scale, Defense Advanced Research Project Agency (DARPA), Tech. Rep. (2008).

[15] E. Meneses, Scalable message-logging techniques for effective fault tolerance in HPC applications, Ph.D. thesis, Dept. of Computer Science, University of Illinois, http://charm.cs.uiuc.edu/papers/13-17/ (2013).

[16] R. Strom, S. Yemini, Optimistic recovery in distributed systems, ACM Trans. Comput. Syst. 3 (3) (1985) 204–226. doi:http://doi.acm.org/10.1145/3959.3962.

[17] L. Alvisi, K. Marzullo, Message logging: pessimistic, optimistic, and causal, International Conference on Distributed Computing Systems (1995) 229–236.

[18] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, F. Vivien, Checkpointing strategies for parallel jobs, in: Supercomputing, SC '11, ACM, New York, NY, USA, 2011, pp. 33:1–33:11. doi:10.1145/2063384.2063428.
URL http://doi.acm.org/10.1145/2063384.2063428

[19] J. T. Daly, A higher order estimate of the optimum checkpoint interval for restart dumps, Future Generation Comp. Syst. 22 (3) (2006) 303–312.

[20] Intel, Intel-64 and IA-32 Architectures Software Developer's Manual , Volume 3A and 3B: System Programming Guide, 2011.

[21] L. Kalé, S. Krishnan, Charm++ : A Portable Concurrent Object Oriented System Based on C++, in: Proceedings of the Conference on Object Oriented Programmi ng Systems, Languages and Applications, 1993.

[22] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, D. Arnold, Evaluating the viability of process replication reliability for exascale systems, in: Supercomputing, ACM, New York, NY, USA, 2011, pp. 44:1–44:12. doi:10.1145/2063384.2063443.
URL http://doi.acm.org/10.1145/2063384.2063443

[23] S. Hong, H. Kim, An integrated gpu power and performance model, in: ISCA, 2010, pp. 280–289.

[24] E. Meneses, X. Ni, L. V. Kale, A Message-Logging Protocol for Multicore Systems, in: Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS), Boston, USA, 2012.

[25] M. Diouri, O. Gluck, L. Lefevre, F. Cappello, Energy considerations in checkpointing and fault tolerance protocols, in: 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS 2012), Boston, USA, 2012.

[26] M. el Mehdi Diouri, O. Glück, L. Lefèvre, F. Cappello, Ecofit: A framework to estimate energy consumption of fault tolerance protocols for HPC applications, in: CCGRID, 2013, pp. 522–529.