Leaping Shadows: Fault Tolerance with Forward Progress

Xiaolong Cui, Taieb Znati, Rami Melhem
Computer Science Department
University of Pittsburgh
Pittsburgh, USA
Email: {mclarencui, znati, melhem}@cs.pitt.edu

***Abstract*—**

***Keywords*-Leaping; Lazy Shadowing; extreme-scale computing; forward progress; reliability;**

I. Introduction

Computing power of supercomputers has become a vital factor in determing a country's competiveness in research, technology, and even national defence. As the demand for massive computing power continues to increase, the HPC community is striving to develop larger computing systems in pursuit of the power of scale. Nowadays, a race among various countries is underway to build the world's first exascale supercomputer to accelerate scientific discoveries, big data analytics, etc. It is expected that the first exascale supercomputer will enter service by 2020. However, making the transition to extreme-scale poses numerous unavoidable scientific and technological challenges.

As today's HPC and Cloud Computing systems grow to meet tomorrow's compute power demand, two of the biggest challenges will be system resilience and power consumption, both being direct consequences of the dramatic increase in the number of system components [1], [2]. Regardless of the reliability of individual component, the system level failure rate will continue to increase as the number of components increases, possibly by several orders of magnitude. It is projected that the Mean Time Between Failures (MTBF) of future extreme-scale systems will be at the order of hours or even minutes, meaning that many failures will occur every day [3]. Without an efficient fault tolerance mechanism, faults will be so frequent that the applications running on the systems will be continuously interrupted, requiring the execution to be restarted from a previous point for every failure.

Also thanks to the continuous growth in system components, there has been a steady rise in power consumption in large-scale distributed systems. In 2005, the peak power consumption of a single supercomputer reached 3.2 Megawatts. This number was doubled only after 5 years, and reached 17.8 Megawatts with a machine of 3,120,000 cores in 2013. Recognizing this rapid upward trend, the U.S. Department of Energy has set 20 megawatts as the power limit for future exascale systems, turning power from an optimzation goal to a leading system design constraint.

Today, two classic approaches to fault tolerance are dominant. The first approach is rollback recovery, which rolls back and restarts the execution every time there is a failure. This approach is often equipped with checkpointing to periodically save the execution state to a stable storage so that execution can be restarted from a recent checkpoint in the case of a failure [4], [5], [6]. On the other hand, the second approach, referred to as process replication, exploits hardware redundancy and executes multiple instances of the same task in parallel to overcome failure [7], [8], [9].

However, neither of the above two approaches applies well to future extreme-scale systems. Checkpointing/restart lacks of forward progress, meaning that its efficiency drops as failure rate increases. Given the anticipated increase in system level failure rates and the time to checkpoint large-scale compute-intensive and data-intensive applications, the time required to periodically checkpoint an application and restart its execution will approach the system's MTBF [10]. On the other hand, process replication has a low system efficiency (no more than 50%) by default because of its need for dedicated resources for replica processes. Therefore, should an exascale system be built in the next few years with any of the two fault tolerance approaches, a large portion of its capacity would be wasted due to fault tolerance. Furthermore, neither of them addresses the power cap issue.

To address the above shortcomings, we proposed Lazy Shadowing as an adaptive and power-aware approach to achieve high-levels of resilience in extreme-scale, failure-prone computing environments. Lazy Shadowing is a novel computational model that goes beyond adapting or optimizing well known and proven techniques, and explores radically different methodologies to fault tolerance [11]. The basic tenet of Lazy Shadowing is to associate with each main process a suite of shadows whose size depends on the "criticality" of the application and its performance requirements. Each shadow process is an exact replica of the original main process, and a consistency protocol is used to assure that the main and the shadow are consistent. When possible, the shadow executes at a reduced rate to save power. Lazy Shadowing achieves QoS along with power awareness by dynamically responding to failures. Experimental results demonstrate that Lazy Shadowing achieves higher performance and significant energy savings compared

to existing approaches in most cases.

Recently, however, several limitations of Lazy Shadowing have been identified. Firstly, Lazy Shadowing can only tolerate one failure per shadowed set. As a result, failures, as they occur, reduce the vulnerability of the system. Secondly, shadows are substitutes for mains, increasing the implementation complexity to deal with failures. Thirdly, the assumsion of crash failures limits the efficiency of Lazy Shadowing.

In this paper, we present our latest work on addressing the above limitations. Firstly, we combine dynamic process creation with shadow leaping to efficiently keep the system from becoming vulnerable. Then, we deviate from the concept of shadow as a replica, where a shadow is promoted to a new main when the original main fails, and use shadow as an "assistant" to a main whereby a shadow helps a main to overcome failures until the main completes execution. Last but not least, for different types of failures, we study different schemes to optimize Lazy Shadowing. The main contributions of this paper are as follows:

- An enhanced scheme of Lazy Shaodwing that incorporates rejuvenation techniques for consistent reliability and improved performance.
- A full-feature implementation for Message Passing Interface
- A through evaluation of the overhead and performance of the implementation with various benchmarks and real applications.

The rest of the paper is organized as follows. We begin with a survey on related work in Section II. Section III introduces system design and fault model, followed by discussion on implementation details in Section IV. Section V presents empirical evaluation results. Section VI concludes this work and points out future directions.

II. RELATED WORK

III. SYSTEM DESIGN

The computational model of Lazy Shadowing has been continuously optimized to improve its scalability and efficiency, to eliminate vulnerability, and to reduce implementation complexity and overhead. This section presents the system design of the comprehensive Lazy Shadowing model.

A. Fault model

As demonstrated in [11], Lazy Shadowing is able to tolerate both hardware and software failures under the fail-stop fault model. In this work, we continue with the assumption of fail-stop model. In order to further improve resilience and efficiency, however, we differentiate between temporary failures and permanent failures. Temporary failures include memory bit flips, kernel panic, etc., and can be recoverd by rebooting the machine, while permanent failures, such as failure in power supply and network switch, needs the device to be replaced in order to recover. Later in this paper we will show that Lazy Shadowing maximizes the resource utilization for resilience by adopting different schemes for different types of failures.

B. Shadowing

Shadowing is the essential concept in Lazy Shadowing. With shadowing, each original process (referred to as main) is associated with a replica process (referred to as shadow) that executes at a potentailly lower rate to save power. Then fault tolerance comes from the property that if one process fails, its associated process can continue to complete the task. For example, if a main process fails, its shadow is promoted to a new main and continue to carry out the assigned task. In this way, shadow processes are substitutes for mains in the case of failure. In this work, we deviate from the original concept of shadow as a replica and use a shadow as "assitant" to a main to complete computation in the presence of failures. Specifically, if a main fails, a new main process will be rejuvenated from its associated shadow by our leaping technique (discussed below), while the shadow remains a shadow.

C. Leaping

Leaping is initially proposed as a technique to boost performance in [11]. As the shadows execute slower than mains, failure recovery will introduce delay to the execution. Leaping opportunistically takes advantage of the recovery time and copies state from healthy mains to their associated shadows. As a result, shadows achieve forward progress with minimal overhead, and recovery time for future potential failures is minimized.

Recently, we have identified an empirical problem to which leaping is a solution. When Lazy Shadowing is used to execute an MPI application, application messages are generated at the rate of the mains, but consumed by the shadows at a lower rate because the shadows are slower. As a result, messages accumulate on the shadow side and could possibly result in a buffer overflow. Leaping is naturally a solution to this problem as it can move the shadows forward and synchronize their execution states with those of the mains. After the synchronization, accumulated messages at the shadows become obsolete and thus can be safely discarded. To differentiate the two cases where leaping is used, leaping during failure recovery is referred to as failure induced leaping while leaping to avoid buffer overflow is referred to as forced leaping.

D. Rejuvenation

It has been discussed in [11] that with shadow collocation each shadowed set can only tolerate one failure. After the first failure, all main processes in the shadowed set would lose their shadows and become vulnerable. Although quantitative study show that a second failure in a shadowed set is unlikely to occur even with over one million processes, in

practice the system will become more and more vulnerable as failures occur. In many cases, it would be too costly to take such risk, especially for long-running, large-scale, and mission-critical applications. Therefore, it is preferable to maintain the same level of resilience across failures.

Not surprisingly, vulnerability could be avoided by creating a new process for every failed process (either main or shadow). In this way, every main is always guaranteed to have an associated shadow and shadow does not need to substitute a main. The problem, however, is that the newly created process will start from the beginning and may lag far behind the other processes in the system. Later on when the new processs needs to participate in a synchronization point or when it need to perform a failure recovery, significant delay will incur as a result of its lag. Fortunately, leaping can be used to deal with this issue. After a new process is created, we can use leaping to synchronize the new process' state with the existing one. Consider a pair of main process, M, and shadow process, S. If M fails, a new main process will be created to replace M, and then a leaping from S will advance the new process to the state of S. On the other hand, if S fails, a new shadow process will be created and immediately advanced to the state of M by leaping.

Depending on the type of failure, the new process will be placed at different locations. If it is a temporary failure, the node where failure occurs will be rebooted and then used to host the new process, whether it is a main or shadow. Although there is a delay from the rebooting, it is usually acceptable and can be accounted part of the recovery. For permanent failures, the node cannot be used and we have to migrate and collocate some processes. If the new process is a main, its existing shadow will be migrated to another node where shadow process(es) reside, and make room for the new main. Otherwise, if the new process is a shadow, it will be directely created on a shadow node.

IV. IMPLEMENTATION

Implementation issues have been briefly discussed in [11]. In this paper we present the details of our full-feature implementation of lsMPI, which is a library for Message Passing Inteface. Instead of a building a complete MPI implementation from scratch, lsMPI is designed as a separate layer between MPI and user application, and uses the MPI profiling hooks to intercept every MPI call. In this way, we not only can take advantage of existing MPI performance optimization that numerous researches have spent years on, but also achieve portability across all MPI implementations that comply with the MPI standard. When used, lsMPI transparanetly spawns the shadow processes during the initialization phase, manage the coordination between main and shadow processes during execution, and guarantee order and consistency for messages and non-deterministic events.

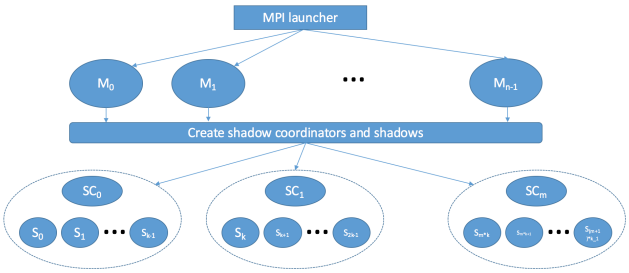

Figure 1: Logical organization of a MPI world with Lazy Shadowing.

A. *Execution rate control*

While the mains always execute at maximum rate for the consideration of HPC's throughput goal, the shadows can be configured to executing slower by collocation [11]. When the user specifies N processes, lsMPI will translate it into $2N+K$ processes where K is the number of shadowed sets. The user has the flexibility to specify how the main processes are organized into shadowed sets through a rankfile. During initialization lsMPI will spawn N main processes, N shadow processes, and K shadow coordinator processes for K shadowed sets. The logical organization is depicted in Figure 1. Each shadow coordinator is collocated with all the shadow processes in a shadowed set, and it manages the coordination between main and shadow processes the shadowed set. When a main process finishes, it will notify its corresponding shadow coordinator, which then terminates the associated shadow process to save power. When a main process fails, the shadow coordinator will send a signal to initiates a leaping from the associated shadow.

B. *Consistency*

State consistency between mains and shadows is required both during normal execution and following a failure. We design a consistency protocol, as shown in Figure 2, to assure that the shadows see the same message order and MPI operation results as the mains. In this figure, A and B represent two mains, and A' and B' are their shadows. For each message, the main of the sender sends a copy of the message to each of the main and shadow of the receiver, and the shadow of the sender is suppressed from sending out messages. We assume that two copies of the same message are sent in an atomic manner, as this multicast functionality can be implemented within NIC.

We assume that only MPI operations can introduce non-determinism. MPI_ANY_SOURCE receives may result in different message orders between the main and shadow. To deal with this, we always let the main receive a message ahead of the shadow and then forward the message source to its shadow using a SYNC message (Figure 2). The shadow then issues a receive with the specific source. Other operations, such as MPI_Wtime() and MPI_Probe(), can be

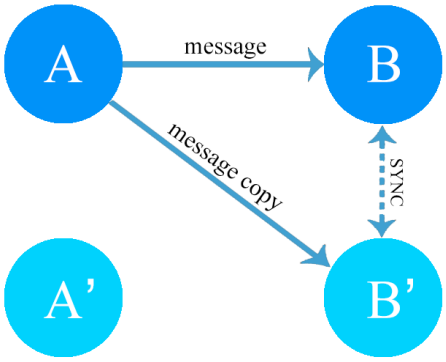

Figure 2: Consistency protocol for lsMPI.

dealt with by always forwarding the result from the main to the shadow.

C. Leaping

Checkpointing/restart requires each process to pause and save its execution state, which can be used by the same or a different process to update its state. Leaping is similar to Checkpointing/restart in saving a process' execution state, but the state is always transferred between a pair of main and shadow processes. To reduce the size of data involved in saving a process' execution state, we choose to implement leaping in the same way as application-level checkpointing. lsMPI provides a routine for users to register data as process state. Application user could use application specific knowledge to register only necessary variables, or use compiler techniques to automate this [12].

The function header for process state registration is as follows:

void leap_register_state(void *addr, int count, MPI_Datatype dt);

For each piece of data to be registered, three parameters are needed: a pointer to the address of the data, the number of data items, and the datatype. Internally, lsMPI uses a linked list to keep track of all registered data. After each call of "leap_register_state()", lsMPI will add a node to its internal linked list to record the three parameters. During leaping, the linked list is traversed to retrieve all registered data as the process state.

Coordination of leaping is easier than coordination of checkpointing, since leaping is always between a pair of main and shadow processes. To synchronize the leaping between a main and a shadow, shadow coordinator in the corresponding shadowed set is involved. For example, when a main process detects failure of another main process and decides to initiate a leaping, it will send a specil message to its shadow coordinator, which then uses a signal to notify the associated shadow to participate in the leaping.

Different from Checkpointing where the process state is saved to storage, leaping directly transfers process state between a main and its shadow. Since lsMPI keeps track of the mapping between each main and its shadow and MPI provides natural support for message passing between processes, lsMPI uses MPI messages to transfer process state. Although multiple pieces of data can be registered as a process' state, only a single message is needed to transfer the process state, as MPI supports derived datatypes. To prevent the messages carrying process state from mixing with application messages, lsMPI uses a separate communicator for process state messages. With the synchronization of leaping by shadow coordinator and the fast transfer of process state via MPI messages, the overhead of leaping is minimized.

A challenge in leaping lies in the need for maintainning state consistency across leaping. To make sure a pair of main and shadow stay consistent after a leaping, not only user-defined states should be transfered correctly, but also lower level states, such as program counter, need to be updated correspondingly. Specifically, the process which updates its process state during leaping needs to satisfy two requirements. Firstly, after updating its state, the process should resume execution at the same point as the process providing the state. Secondly, the process should discard obsolete message that previously stored in its buffer before resuming normal execution. To address this challenge, first we assume that the application's main body consists of a loop, which is true in most cases.

To satisfy the first requirement, we restrict leaping to always occur at certain possible points, and uses internal counter to make sure that both main and shadow start leaping from the same point. For example, when a main process triggers a leaping and askes shadow coordinator to notify its associated shadow, the shadow coordinator will triggers the shadow's specific signal handler. The signal handler does not carry out leaping, but sets a flag for leaping and receives a counter value that indicates the leaping point from its main process. Then, the shadow will checks the flag and compare the counter value at every possible leaping point. Only when both the flag is set and counter value matches will the shadow start leaping. In this way, it is guaranteed that after leaping the main and shadow will resume execution from the same point. To balance the trade-off between the overhead and the flexibility of checking for when to perform leaping, we choose MPI receive operations as the possile points for leaping.

There is no straighforward solution for the second problem as the message buffer is maintained by MPI runtime and not visible to lsMPI. Alternatively, we choose to remove obsolete message from message buffer by having the process execute all the skipped MPI communication routines after it finishes leaping. To achieve this, we require the user to define a function for the MPI communication functions used in the application's main body loop. The function should have two parameters to specify the starting and ending index for skipped iterations. In addition, the user needs to register

the function with lsMPI with the following library call:

void leap_register_func(void (*func)(int, int));

To discard all obsolete messages after leaping, the process that updates its process state will call the registered function, for which the two parameters will be automatically specified by lsMPI. Essentially, it executes all the MPI communication functions from the skipped iterations and consumes all the useless messages.

V. EVALUATION

VI. CONCLUSION AND FUTURE WORK

ACKNOWLEDGMENT

This research is based in part upon work supported by the Department of Energy under contract DE-SC0014376.

REFERENCES

[1] S. Ashby, B. Pete, C. Jackie, and C. Phil, "The opportunities and challenges of exascale computing," 2010.

[2] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson *et al.*, "Addressing failures in exascale computing," *International Journal of High Performance Computing Applications*, p. 1094342014522573, 2014.

[3] 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, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead," 2008.

[4] E. Elnozahy and et. al., "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.

[5] S. Kalaiselvi and V. Rajaraman, "A survey of checkpointing algorithms for parallel and distributed computers," *Sadhana*, vol. 25, no. 5, pp. 489–510, 2000. [Online]. Available: http://dx.doi.org/10.1007/BF02703630

[6] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, Feb. 1985. [Online]. Available: http://doi.acm.org/10.1145/214451.214456

[7] J. F. Bartlett, "A nonstop kernel," in *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, ser. SOSP '81. New York, NY, USA: ACM, 1981, pp. 22–29. [Online]. Available: http://doi.acm.org/10.1145/800216.806587

[8] W.-T. Tsai, P. Zhong, J. Elston, X. Bai, and Y. Chen, "Service replication strategies with mapreduce in clouds," in *Autonomous Decentralized Systems,* 10^{th} *Int. Symp. on*, 2011, pp. 381–388.

[9] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, 2011, pp. 44:1–44:12.

[10] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward exascale resilience," *Int. J. High Perform. Comput. Appl.*, vol. 23, no. 4, pp. 374–388, Nov. 2009. [Online]. Available: http://dx.doi.org/10.1177/1094342009347767

[11] X. Cui, T. Znati, and R. Melhem, "Adaptive and power-aware resilience for extreme-scale computing," in *16th IEEE International Conference on Scalable Computing and Communications*, July 18-21 2016.

[12] G. Bronevetsky, D. Marques, K. Pingali, S. McKee, and R. Rugina, "Compiler-enhanced incremental checkpointing for openmp applications," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–12.