Future Generation Computer Systems 30 (2014) 66–77

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Accelerating incremental checkpointing for extreme-scale computing

Kurt B. Ferreira [a,*], Rolf Riesen [b], Patrick Bridges [c], Dorian Arnold [c], Ron Brightwell [b]

[a] Scalable System Software Department, Sandia National Laboratories, Albuquerque, NM 87185-1319, United States
[b] IBM Research, Ireland
[c] Department of Computer Science, University of New Mexico, Albuquerque, NM, United States

HIGHLIGHTS

- We describe a novel incremental checkpointing solution using hashing.
- We examine the performance of this approach with real HPC workloads.
- We model the benefits of this incremental approach for future systems.
- We show that GPUs can dramatically increase hashing speeds.
- However, this increase in speed has little impact on efficiency.

ARTICLE INFO

Article history:
Received 14 July 2012
Received in revised form
15 February 2013
Accepted 13 April 2013
Available online 6 May 2013

Keywords:
Fault-tolerance
Checkpointing
Incremental checkpointing
Graphics processing units

ABSTRACT

Concern is beginning to grow in the high-performance computing (HPC) community regarding the reliability of future large-scale systems. Disk-based coordinated checkpoint/restart has been the dominant fault tolerance mechanism in HPC systems for the past 30 years. Checkpoint performance is so fundamental to scalability that nearly all capability applications have custom checkpoint strategies to minimize state and reduce checkpoint time. One well-known optimization to traditional checkpoint/restart is incremental checkpointing, which has a number of known limitations. To address these limitations, we describe `libhashckpt`, a hybrid incremental checkpointing solution that uses both page protection and hashing on GPUs to determine changes in application data with very low overhead. Using real capability workloads and a model outlining the viability and application efficiency increase of this technique, we show that hash-based incremental checkpointing can have significantly lower overheads and increased efficiency than traditional coordinated checkpointing approaches at the scales expected for future extreme-class systems.

1. Introduction

Disk-based coordinated checkpoint/restart has been the dominant fault tolerance mechanism in high performance computing (HPC) systems for at least the past 30 years. In current large distributed-memory HPC systems, this approach generally works as follows: periodically, all nodes quiesce activity, write all application and system state to stable storage, and then continue with computation. In the event of a failure, the stored checkpoints are read from stable storage to return the application to a previous known-good state.

Checkpoint performance impacts scalability of large-scale applications to such a degree that many capability applications have their own optimized *application-specific* checkpoint mechanism to minimize the saved checkpoint state and therefore the time to write the checkpoint to stable storage (this time is also referred to as the checkpoint commit time). While this approach minimizes the application state that must be written to disk, it requires intimate knowledge of the application's computation and data structures, and is typically difficult to generalize to other applications.

One well-known and generalized optimization of traditional checkpoint/restart is *incremental checkpointing*. Incremental checkpointing [1–3] attempts to reduce the size of a checkpoint, and therefore the checkpoint commit time, by saving only differences (or deltas) in state from the last checkpoint. The underlying assumption being that the mechanism used to determine the differences in state has significantly lower overhead than the time to save the additional data to stable storage.

Current incremental methods have failed to achieve dramatic decreases in checkpoint size because of a reliance on page protection mechanisms to determine which address ranges have been written, or *dirtied*, during the checkpoint interval [2]. Relying

* Corresponding author.
E-mail addresses: kbferre@sandia.gov (K.B. Ferreira), rolf.riesen@ie.ibm.com (R. Riesen), bridges@cs.unm.edu (P. Bridges), darnold@cs.unm.edu (D. Arnold), rbbrigh@sandia.gov (R. Brightwell).

0167-739X/$ – see front matter © 2013 Published by Elsevier B.V.
http://dx.doi.org/10.1016/j.future.2013.04.017

solely on page-based mechanisms forces such an approach to work at a granularity of the operating systems page size. Therefore, even if only one byte in a page is written, the entire page is marked as dirty and must be saved. Furthermore, if identical values are written to a location, that page is still marked as dirty. These problems are compounded by the increasing maximum page sizes of modern processors and the increased performance for HPC applications on these larger page sizes.

To address these limitations, we describe a hybrid incremental checkpointing approach that uses page protection mechanisms, a hashing mechanism that can be optionally be offloaded to GPUs if available and idle. GPUs reduce the overhead and power consumption of the hash calculation. Using real HPC workloads, this work compares the performance of this technique against a page protection-based incremental systems and a highly optimized, application-specific checkpoint technique. Our results show that our approach is able to dramatically reduce system checkpoint sizes compared to previous incremental checkpointing systems; in some cases approaching the checkpoint sizes of hand-tuned application-specific checkpointing systems. Our results also show that this technique can significantly improve application efficiency, with the key performance factor being the amount of memory compression from the technique (i.e. the size of the checkpoint file), rather than the speed of the incremental approach.

This paper is organized as follow. First in Section 2, we define a model to illustrate when this hash-based approach will pay off both in comparison to a page-based incremental checkpointing approach as well as a more traditional, disk-based checkpointing approach. In Section 3, we describe the design and implementation of `libhashckpt`, a previously published [4] incremental checkpointing library. We show the resulting checkpoint state compression from this technique using a number of real-world HPC capability workloads in Section 4. In addition, we compare the compression results against an optimal application-based checkpointing mechanism. In Section 5, using a number of hash algorithms, we show the costs of performing this hashing on a CPU versus the speedup seen using a GPU. Section 6 uses the aforementioned model and measured results to present the viability of this technique using a GPU and CPU for possible systems in the exascale design space thereby defining under which situations we would use this approach. In Section 7 we outline the increase in application efficiency in those scenarios where the approach is viable. Related checkpoint optimization work is discussed in Section 8. We conclude with a discussion of the implications of this work as well as ongoing research in Section 9.

2. A model for the viability of hash-based incremental checkpointing

To evaluate the viability of this method we compare the performance of this hash-based mechanism first against that of a strictly page-based approach. This hash-based approach outperforms a page-based approach when the reduction in the checkpoint size for the hash method outweighs the cost of computing the hashes of the modified pages. More specifically, this approach is viable when the sum of the time to hash modified memory (T_{hash}), plus the time to write the application blocks that have been determined changed ($T_{write\ hash}$), is less than the time to write the memory that hash been determined changed using a strictly page-based approach ($T_{write\ whole}$). This model was first introduced by Plank et al. in [5]. For clarity we provide it here. In more detail we have:

$$T_{hash} + T_{write\ hash} < T_{write\ whole} \tag{1}$$

$$\left(\frac{|c|}{\beta_{hash}}\right) + \left(\frac{(1-\alpha)\times|c|}{\beta_{ckpt}}\right) < \frac{|c|}{\beta_{ckpt}} \tag{2}$$

where:

- $|c|$ is the size of page-based checkpoint
- α is the percent reduction of hash-based approach in comparison to the page-based method
- β_{hash} is the per-process hash rate
- β_{ckpt} is the per-process checkpoint commit rate.

This equation can be reduced to:

$$\frac{\beta_{ckpt}}{\beta_{hash}} < \alpha. \tag{3}$$

The maximum per-process checkpoint commit rate (β_{ckpt}) is generally known for many HPC platforms. Therefore, we must measure the hashing rate (β_{hash}), which is specific to both a specific platform and hashing algorithm; and the compression percentage (α), which will be specific to a particular application. In the next section, we use the `libhashckpt` library to measure these quantities.

2.1. Viability against coordinated checkpointing

In this section we outline the viability of this hash-based technique in comparison to traditional checkpoint restart. Similar to above, this approach is viable when the sum of the time to hash modified memory (T_{hash}), plus the time to write the application blocks that have been determined changed ($T_{write\ hash}$), plus the time to mark dirty pages during a checkpoint interval (T_{dirty}), is less than the time to write the memory that hash been determined changed using a strictly page-based approach ($T_{write\ whole}$). Again we have:

$$T_{hash} + T_{write\ hash} + T_{dirty} < T_{write\ whole} \tag{4}$$

$$\left(\frac{|c|}{\beta_{hash}}\right) + \left(\frac{(1-\alpha)\times|c|}{\beta_{ckpt}}\right) + T_{dirty} < \frac{|c|}{\beta_{ckpt}}. \tag{5}$$

As we will show in later sections, for checkpoint intervals expected on future systems, T_{dirty} is equal to 0. Therefore, this equation again reduces down to:

$$\frac{\beta_{ckpt}}{\beta_{hash}} < \alpha. \tag{6}$$

The same as Eq. (3). Therefore, the viability of this approach is the same in comparison to both a page-based incremental approach and a traditional checkpoint/restart mechanism.

3. `Libhashckpt`: hash-based incremental checkpointing

3.1. Overview

This work uses a previously published incremental checkpoint library called `libhashckpt` [4]. The hash-based incremental checkpointing mechanisms in `libhashckpt` works as follows. While the application is running, the library uses the page-protection mechanism (i.e. `mprotect()`) to mark those virtual memory pages that have been written in the checkpoint interval as potentially dirty. To support MPI applications, the library must also intercept all receive calls, including those implicit to collective operations which use buffers internal to the MPI library. `libhashckpt` marks all receive message buffers as dirty, identifying them as candidates to be checked by the hashing mechanism. These message buffers require marking as changes in memory from high-performance, user-level network hardware, such as those found on leadership-class HPC systems, are not subject to the processor's page protection mechanisms. Therefore,