Hindawi Publishing Corporation
Scientific Programming
Volume 2016, Article ID 9315493, 11 pages
http://dx.doi.org/10.1155/2016/9315493

Research Article

Optimizing Checkpoint Restart with Data Deduplication

Zhengyu Chen, Jianhua Sun, and Hao Chen

College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China

Correspondence should be addressed to Jianhua Sun; jhsun@hnu.edu.cn

Received 1 March 2016; Accepted 5 May 2016

Academic Editor: Laurence T. Yang

Copyright © 2016 Zhengyu Chen et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The increasing scale, such as the size and complexity, of computer systems brings more frequent occurrences of hardware or software faults; thus fault-tolerant techniques become an essential component in high-performance computing systems. In order to achieve the goal of tolerating runtime faults, checkpoint restart is a typical and widely used method. However, the exploding sizes of checkpoint files that need to be saved to external storage pose a major scalability challenge, necessitating the design of efficient approaches to reducing the amount of checkpointing data. In this paper, we first motivate the need of redundancy elimination with a detailed analysis of checkpoint data from real scenarios. Based on the analysis, we apply inline data deduplication to achieve the objective of reducing checkpoint size. We use DMTCP, an open-source checkpoint restart package, to validate our method. Our experiment shows that, by using our method, single-computer programs can reduce the size of checkpoint file by 20% and distributed programs can reduce the size of checkpoint file by 47%.

1. Introduction

Infrastructure as a Service (IaaS) [1] is a form of cloud computing that provides virtualized computing resources over the Internet. IaaS allows customers to rent computing resources from large data centers rather than buy and maintain dedicated hardware. More and more software production and services are directly deployed and run on IaaS clouds. With the increase of computing nodes, the probability of failures increases. Hardware failure fail-over or any unexpected reason makes node inactive. For distributed and high-performance computing (HPC) applications, a failure in one node causes the whole system to fail.

Fault tolerance is an indispensable component of cloud computing. However, due to the embarrassingly parallel nature of mainstream cloud applications, most approaches are designed to deal with fault tolerance of individual processes and virtual machines. This either involves restarting failed tasks from scratch (e.g., MapReduce) or leveraging live migration to replicate the state of virtual machines on-the-fly in order to be able to switch to a backup virtual machine in case the primary instance has failed [2, 3].

Checkpoint is an important method to provide fault tolerance for distributed and HPC applications [4]. Through the use of checkpoint, program's in-memory states can be written to persistent storage. If a crash occurs, the previously saved checkpoint can be used to recover program to the latest state. In IaaS cloud environments, checkpoint technique can be more beneficial. For example, when resource prices become expensive or the budget is tight, we can suspend the running program and recover it later, or we can transfer the program to a new cloud provider without losing progress.

However, with the increasing scale of computation, the checkpoint size also increases. So the efficiency of the checkpoint operation is becoming more and more important. Not only will checkpoints incur performance cost, but the checkpoint files consume large storage space, which causes I/O bottlenecks and generates extra operational costs. For example, saving 1 GB RAM for each 1,000 processes consumes 1 TB of space. Although there have been attempts to reduce the IO overhead using hardware technology (i.e., SSDs), with the increasing of the number of computational nodes, this method is still not an ideal solution.

In this paper, we focus on the reduction of the checkpoint file size and ultimately achieve the purpose of reducing the I/O and storage overhead. In the storage space of the checkpoint file is mainly consumed by the content of the process's address space. We analyzed the content of the process's address space in order to identify different segments

of redundancy. The results of analysis show that there exist a lot of duplicate data in the stack section. For distributed applications, in heap segment between different processes, there are a lot of duplicate data, and the contents in their dynamic link library and code segments are the same. But the amount of duplicate data in dynamic link library is less. According to these characteristics, we propose a method that uses inline deduplication to identify and eliminate duplicate memory contents at the page level to reduce the checkpoint file size. DMTCP (Distributed Multithreaded Checkpointing) is a transparent user-level checkpoint package for distributed applications. We implement our method based on DMTCP. Experiments show that our method can greatly reduce the checkpoint size.

The contributions of our work are as follows:

(i) We conduct a detailed analysis of the contents of the checkpoint file, which reveals some characteristics of data redundancy and the potential of integrating deduplication into existing checkpoint systems.

(ii) Motivated by our experimental analysis, we propose a method to reduce the size of the checkpoint file, which works for both single-node and distributed high-performance computing systems.

(iii) We present the design and implementation of our approach in DMTCP and perform extensive evaluation on a set of representative applications.

This paper is organized as follows. Section 2 presents related work. Section 3 describes our motivation and related background. Section 4 contains the design and implementation of our approach. In Section 5, we provide an experimental evaluation of our design. Section 6 analyzes the experimental results, and the conclusions and future work are given in Section 7.

2. Related Work

2.1. Checkpoint Restart. There are roughly four major directions of research on reducing the size of checkpoint file. Incremental checkpointing reduces the checkpoint size by saving only the changes made by the application from the last checkpoint [5–7]. Usually, a fixed number of incremental checkpoints is created in between two full ones. During a restart, the state is restored by using the most recent full checkpoint file and applying in an ordered manner all the differences before resuming the execution.

Memory exclusion [8] skips temporary or unused buffers to reduce the size of the checkpoint file size. This is done by providing an interface to the application to specify regions of memory that can be safely excluded from the checkpoint.

Checkpoint compression is a method for the reduction of the checkpoint file size by reducing the size of process images before writing them to stable storage [9]. Besides compression, another possible solution is deduplication that is the mainstream storage technology. It can effectively optimize storage capacity. This technology can reduce the requirement on physical storage space and can meet the data storage need which grows day by day. Our work focuses on inline deduplication to identify and eliminate duplicate data in the process.

2.2. Data Deduplication. Data deduplication is a specialized data compression technique for eliminating duplicate copies of repeating data [10, 11]. Data deduplication splits the input file into a set of data blocks and calculates a fingerprint for each of them. If there exists a block sharing the same fingerprint, it indicates that block is duplicated, and we only need to store the index number for the duplicated block. Otherwise, it means that data block is unique, and its content needs to be stored. As can be seen from the above process, the key technology of deduplication mainly includes data block segmentation, fingerprint computation, and data block retrieval. Among these techniques, data block segmentation is the most crucial.

Block segmentation algorithm is divided into three types: the fixed-size partition, content-defined chunking, and sliding block. Fixed-size partition algorithm first splits the input data into fixed-size chunks. Then, chunks are compared between each other in order to identify and eliminate duplicates. While simple and fast, fixed-size partition algorithm is very sensitive to data insertion and deletion and not adaptive to the changes of content.

To deal with such issues, content-defined approaches [12] are proposed. Essentially, they involve a sliding window over the data and that hashes the window content at each step using Rabin's fingerprinting method [13]. When a particular condition is fulfilled, a new chunk border is introduced and the process is repeated until all input data was processed, leading to a collection of variable-sized chunks. Content-defined chunking algorithm is not sensitive to changes in content, data insertion and deletion only affects a few blocks, and the remaining data blocks are not affected. But it also has disadvantages, the size of the data block is difficult to determine; coarse granularity results in nonoptimal effect, and fine-grained granularity often leads to higher cost. Thus, the most difficult part of this algorithm lies in how to choose a suitable granularity.

Sliding block algorithm [14] combines the advantages of fixed-size partition algorithm and content-defined chunking algorithm. This algorithm uses Rabin fingerprinting to subdivide byte-streams into chunks with a high probability of matching other chunks generated likewise. If a signature of the chunck/block matches one of the precomputed or prespecified delimiters, the algorithm designates the end of this window as a chunk boundary. Once the boundary has been identified, all bytes starting from the previous known chunk boundary to the end of the current window is designated a chunk. A hash of this new chunk is computed and compared against the signatures of all preexisting chunks in the system. In practice, this method makes use of four tuning parameters, namely, the minimum chunk size, the maximum chunk size, the average chunk size, and the window size. The sliding block algorithm deals with data insertion and data deletion process efficiently and can detect more redundant data than content-defined approaches; its drawback is that it is prone to data fragmentation. In order to obtain more redundant information, we use the sliding

block algorithm in our approach, which will be described later.

At present, data deduplication technology is widely used in the storage system and network system; by using deduplication it can effectively reduce the data storage and system overhead. For example, Srinivasan et al. [15] can achieve 60–70% of the maximum deduplication with less than a 5% CPU overhead and a 2–4% latency impact through the use of deduplication. Agarwal et al. [16] designed EndRE; the system uses the technology of data deduplication to eliminate the redundancy of network data and reduce the cost of WAN access. The experiment results show that EndRE can save an average of 26% of the bandwidth and reduce the end-to-end delay of 30%.

3. Background and Motivation

3.1. Background. Checkpoint restart is a mechanism that periodically saves the state of an application to persistent storage and offers the possibility to resume the application from such intermediate states.

Depending on the transparency with regard to the application program, single-process checkpoint techniques can be classified as application-level, user-level, or system-level. User-level checkpoint services are implemented in user space but are transparent to the user application. This is achieved by virtualizing all system calls to the kernel, without being tied to a particular kernel [17]. System-level checkpointing services are either implemented inside the kernel or as a kernel module [18]. The checkpoint images in system-level checkpointing are not portable across different kernels.

DMTCP (Distributed Multithreaded Checkpointing) is a transparent user-level checkpoint package for distributed applications [19]. DMTCP is very convenient for applications to set checkpoint and restart. It works completely in user space and does not require any changes to the application or operating system. DMTCP automatically tracks all local and remote child processes and their relationships. DMTCP implements a coordinator process because DMTCP can also checkpoint distributed computations across many computers. The user can issue a command to the coordinator, which will then relay the command to each of the user processes of the distributed computation.

Like the principles of a checkpoint, DMTCP also copies the program information in-memory to the checkpoint file. Program's in-memory information includes the process id, the process's address space, opened file information, and signal state. In the checkpoint file, the content of the process's address space occupies the main storage space. The process's address space information is mainly read from */proc/self/maps*, from which we can obtain the contents of the address space. The program's address space consists of heap, stack, shared libraries, *mmap* memory area, data, and text segment.

3.2. Motivation. In this paper, we focus on reducing the size of checkpoint file based on deduplication. In this section, we analyze the content redundancy in regular checkpoint files. The content needed to be stored by the checkpoint files generated by DMTCP can be classified into 5 categories: heap, stack, code, dynamic link library, and mmap. We ignore other types of content in checkpoint files due to the limited space they occupy. The target programs used in our experiment can be divided into two types: single-node program and distributed program. Next, we conduct analysis on program address spaces of these five categories on the two kinds of programs, respectively, in order to characterize data duplication in prospective applications. After retrieving the content of their progress address space, three methods can be used to perform the analysis: gzip compression, deduplication, and the hybrid of compression and deduplication. Sliding block algorithm is used in deduplication, the minimum chunk size is 512 B, the maximum chunk size is 32768 (32 K), average chunk size is 4096 B (4 KB), and the window size is 48 B.

Table 1: Single-node program heap (KB).

Type	Original	gzip	Deduplication	Deduplication + gzip
BaseLine	132	0.49	68.18	0.51
Python	868	128.24	600	130.20
Vim	2220	283.14	2107.46	289.68
BC	264	8.8	232.23	8.83
Perl	264	23.63	264	23.98

3.2.1. Single-Node Program Analysis. We perform experiments using the following applications: BaseLine, a simple C program, whose functionality is to print numbers consistently; Python(2.7.6), an interpreted, interactive, object-oriented programming language; Vim(7.4), an advanced text editor; BC(1.06.95), an arbitrary precision calculator language; and Perl(5.18.2), Practical Extraction and Report Language interpreter.

Table 1 displays the experimental results on heap. The compression rate of BaseLine, Vim, BC, and Perl is 99.6%, 87%, 96.6%, and 91%, respectively, showing that gzip based compression has good effect on heap. But the results become unsatisfying when redundant data deletion technique is used. For example, in BaseLine, the original size of heap is 132 KB; after deduplication it still occupies 68.18 KB, with a deduplication rate of 48.3%. For some applications the results are even worse. For example, the rate for Python and Vim is 30.8% and 5.1%, respectively. The results become even worse if we try to use gzip directly succeeding the deduplication procedure. As shown in Table 1, after the deduplication + gzip operations, the resulting heap size is even bigger than when only the gzip compression approach is used. Of course, the redundant data comes from adding unnecessary index information.

Table 2 illustrates that applying deduplication and deduplication + gzip to stack is much more effective as compared to heap. In BaseLine, the original stack size is 8072 KB; after compression it shrinks to 11.69 KB, and after deduplication it is 105.1 KB. With deduplication + gzip, the size is only 4.15 KB, which means the compression rate is 99.86% and the deduplication rate is 98.7%.

In single-node program experiments, we can conclude that the duplication of heap is quite limited as compared to its

Table 2: Single-node program stack (KB).

Type	Original	gzip	Deduplication	Deduplication + gzip
BaseLine	8072	11.69	105.16	4.15
Python	8072	13.19	137.24	5.75
Vim	8068	13.03	101.36	5.57
BC	8072	13.32	105.18	5.8
Perl	8072	12.37	105.28	4.86

Table 3: Distributed program heap (KB).

Type	Original	Deduplication	Redundancy rate
BaseLine	264	101.59	61.52%
CG	264	165.14	37.54%
EP	264	149.07	43.53%
LU	264	153.09	42%
MG	264	161.13	38.98%
IS	264	129	51.13%

relatively high compression efficiency. On the other hand, the compression and duplication rates are both high in the case of stack, and using gzip + deduplication can achieve much better effect. During the analysis of code, dynamic link library, and mmap, we find that the duplication rate is not high, and gzip compression is more suitable for reducing the size of these components.

3.2.2. Distribute Program Analysis. We conduct experiments on distributed programs using the following applications: BaseLine and NAS NPB3.3. The BaseLine is a simple MPI program, whose function is to calculate Pi. The NAS Parallel Benchmarks (NPB) are a small set of programs designed to help evaluate the performance of parallel supercomputers. NPB 3.3-MPI was used in our experiments. The benchmarks run under MPICH2 are CG (Conjugate Gradient, irregular memory access and communication, level C), EP (Embarrassingly Parallel, level C), LU (Lower-Upper Gauss-Seidel Solver, level C), MG (Multigrid on a sequence of meshes, long- and short-distance communication, memory intensive, level C), and IS (Integer Sort, random memory access, level C). For convenience, we adopt two computers to build the experiment cluster. A single process is run on each node. The content of various segments, that is, heap, stack, code, DLL, and mmap of each process, is collected. "Original" in Tables 3 and 4 represents the total size of each segment of the two nodes.

Table 3 shows the results of the heap of distributed programs. The original size of BaseLine is 264 KB. After deduplication, the size becomes 201.59 KB, obtaining a duplication rate of 61.52%. Table 3 also shows that the duplication rates of CG, EP, LU, MG, and IS are 37.54%, 43.53%, 42%, 38.98%, and 51.13%, respectively. The rates increase significantly as compared to the single-program counterparts, inferring that there exists a lot of duplicated information on the heap of each process between the two different nodes. We can apply gzip + deduplication method to reduce the heap size of distributed programs.

Table 4: Distributed program stack (KB).

Type	Original	Deduplication	Redundancy rate
BaseLine	16140	80.21	99.5%
CG	16144	88.24	99.45%
EP	16140	80.24	99.5%
LU	16136	88.27	99.45%
MG	16144	88.27	99.45%
IS	16136	76.2	99.53%

Table 4 displays the result of the stack of distributed programs, which implies that there is a lot of redundant data in stacks. For example, the original size of BaseLine is 16140 KB, and the size becomes 80.21 KB after deduplication, obtaining a duplication rate of 99.5%. Other testing applications also reveal a very high duplication rate. The rates of CG, EP, LU, MG, and IS are 99.45%, 99.5%, 99.45%, 99.45%, and 99.53%, respectively. In conclusion, not only do stacks have a lot of redundant data themselves, but also the duplication rates of the stack between different processes are even higher.

Other segments, like code, DLL, and mmap, do not have too much redundant data on the same node, but their contents are the same between different nodes in distributed programs. But DMTCP stores every code, mmap, and DLL of each node when setting checkpoints. Thus we only need one checkpoint file to store the code and mmap of multiple processes on the same node and DLL, while others only need to keep the index.

In this section, we analyze the duplication rate of heap, stack, DLL, and mmap in single program and distributed program, respectively. In order to reduce the checkpoint file size efficiently, we implement different strategies on different segments, which can be summarized as follows:

Single-node program:

(i) for heap, code, DLL, mmap, and others: we continue to use gzip for compression;
(ii) for stack: gzip + deduplication method is adopted.

Distributed program:

(i) for heap and stack between different processes: gzip + deduplication method is adopted;
(ii) for code, DLL, and mmap of multiple processes on the same node: only one copy is stored in the checkpoint file; others just keep the corresponding index;
(iii) other: we continue to use gzip compression.

4. Design and Implementation

4.1. Overview. In this paper, we focus on how to reduce the checkpoint file size. When setting a checkpoint, we examine each memory page in the program and apply techniques tailored to the page type as detailed in the following.

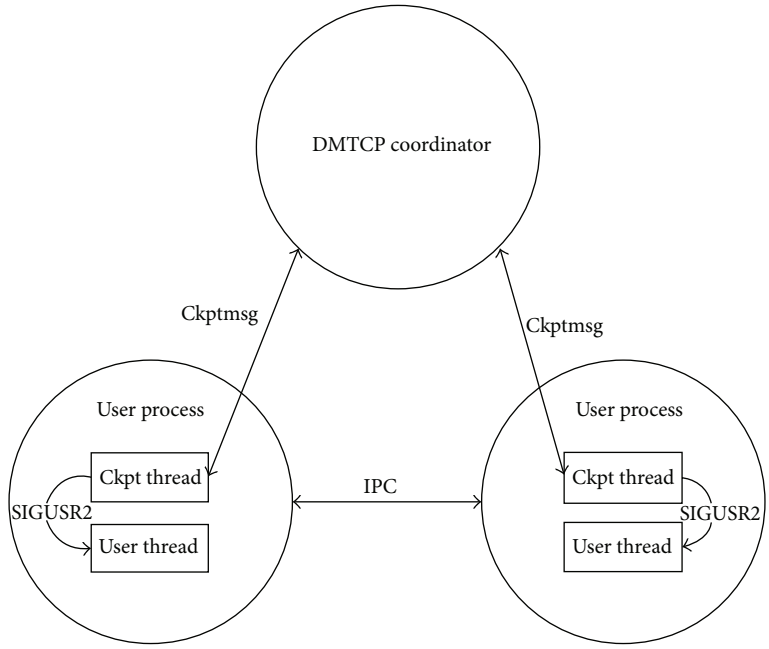


Figure 1: Architecture of DMTCP.

(i) Heap: in single-node programs, redundancy in heap is not high, so gzip compression is used. In distributed programs, we employ a combination of deduplication and gzip to identify and eliminate redundancy within heap pages.

(ii) Stack: for both single-node programs and distributed programs, we employ a combination of deduplication and gzip to identify and eliminate redundancy within stack pages.

(iii) DLL and mmap of multiple processes on the same node and code: in single programs, we only use gzip compression. In distributed programs, for DLL, code, and mmap of multiple processes on the same node, only one copy is stored in the checkpoint file, while a corresponding index is kept.

(iv) Other: for both single-node programs and distributed programs, the remaining pages are compressed using gzip.

Figure 1 shows the architecture of DMTCP. DMTCP uses a stateless centralized process, the coordinator, to synchronize checkpoint and restart between distributed processes. In each user process, DMTCP creates a checkpoint thread. The checkpoint thread is used to receive commands sent from the coordinator, such as setting up checkpoints. In addition, the checkpoint thread contacts user thread through the signal (SIGUSR2). Our approach is implemented based on DMTCP. Next, we discuss in detail the specific implementation.

4.2. Heap and Stack. The application can request a new checkpoint at any time by using the command: *dmtcp_command-c*. When this command is issued and received by the coordinator, each node will start to set checkpoint. From the previous analysis, we can know that heap segment redundancy in single-node program is not high, and stack segments have higher duplicate data. In a distributed application, heap and stack segment between different processes have a high possibility of containing duplication. We need to identify the single-node program and distributed programs. So, first we need to check the type of the program. The process of each node sends a request to the coordinator; the coordinator determines whether the program is a distributed application through collecting process information of each node and returns the results to them. If the program is not a distributed application, we only need to identify and eliminate duplicate memory contents at the heap and stack page. Otherwise, we need to rely on the information provided by the coordinator to eliminate the redundancy between different process of nodes.

Now we describe the heap and stack data deduplication algorithm in distributed application. First, we need to get the hash value of each memory page by hashing the content of each page. If the hash value exists in the local hash table, we can regard the page as a duplicate and obtain the page index from the hash table and copy it to the checkpoint file. Otherwise, we need to send the information to the coordinator to query whether the hash value exists in other processes.

If the coordinator returns true, we will get the page index from the coordinator and copy it to the checkpoint file. Otherwise, we need to store the page content to the checkpoint file and generate a page index and send it to the coordinator. The index contains the checkpoint file name, the offset of the page in checkpoint file, and the page length. The steps of this algorithm are depicted graphically in Algorithm 1.

```
(1)  function DoDedup(Memorydata, type)
(2)    LocalHashes ← nil
(3)    Mpiflag ← GetProgramType()
(4)    UniqueChunkNum ← 0
(5)    ChunkNum ← 0
(6)    if Mpiflag = false then
(7)      return SingleDodedup(Memorydata, type)
(8)    end if
(9)    while p_i ∈ Memorydata do
(10)     h_i ← Hash(p_i)
(11)     if h_i ∉ LocalHashes then
(12)       UniqueChunkNum + +
(13)       if InquireFromCoor(h_i, type) = true
         then
(14)         DownloadIndexInfo(indexinfo)
(15)         StoreUniqueChunk(indexinfo)
(16)        else
(17)         StoreUniqueChunk(p_i)
(18)         GenerateIndex(indexinfo)
(19)         UploadIndexInfo(h_i, type, indexinfo)
(20)        end if
(21)       MetaData[ChunkNum]          ⟵
     UniqueChunkNum
(22)     ChunkNum ← ChunkNum + 1
(23)     Hashes ← LocalHashes ∪ {h_i}
(24)   else
(25)     MetaData[ChunkNum]          ⟵
     UniqueChunkNum
(26)     ChunkNum ← ChunkNum + 1
(27)    end if
(28)  end while
(29) end function
```

Algorithm 1: Heap and stack data depulication algorthim.

4.3. Dynamic Link Library, Shared Memory, and Code. In a distributed application, all of the dynamic link library, code, and mmap of multiple processes on the same node are the same. For all the running processes, this content only retains one copy. In DMTCP, however, these contents are copied to the local checkpoint file. Therefore, when setting a checkpoint, we just need to copy the pages of the dynamic link library, shared memory, and code to a checkpoint file, and the other checkpoint files only need to save the corresponding index information. However, for single-node applications, the effects of data deduplication are not prominent. So this algorithm does not apply to single-node applications. DMTCP invokes gzip compression by default.

Like Algorithm 1, we first need to query the program type. For a distributed program, we continue to the next step. Otherwise, the algorithm is over. Next, we just have to send local IP address to the coordinator to query whether there are multiple processes running on the local node. If the coordinator receives multiple IP addresses, it means the result is true. The coordinator will send the result to all processes. For shared memory segments, our algorithm is only suitable for multiple processes on a node. In order to eliminate redundancy between all processes dynamic link library and code segment, we send a query to the coordinator to know whether there is an index for the segment on the coordinator. If the index does not exist, we copy the dynamic link library, code, and DLL content to the checkpoint files and then build an index and upload it to the coordinator. Otherwise, the index information is obtained from the coordinator and saved to the checkpoint file. The steps of this algorithm are depicted in Algorithm 2.

4.4. Restart. Let us look at the operation of the stack segment during the restart. For single-node applications, according to the normal process of restart, the difference is that if the data read from the checkpoint file is index, we need to locate and read the real content from the memory according to the index. For distributed applications, the restart is more complicated. Each node containing unique block information needs to create a listener thread, which is used to monitor requests from other processes. After receipt, the requested content will be sent to the requesting process. Before creating the listener, we need to send initialization information to the coordinator for registration. Registration information includes the checkpoint file name, IP address, and port number. When reading the stack and heap information from the checkpoint file, we first read the metadata of each page and locate each page from the meta information. If the page

```
(1)  function DoDedup1(LibInf, type)
(2)    Mpiflag ← GetProgramType()
(3)    pos ← 0
(4)    Ckptname ← Getckptname()
(5)    if Mpiflag = true then
(6)      return SingleDodedup(LibInf, type)
(7)    end if
(8)    if InquireFromCoor(LibInf.name, type) = true then
(9)      Index ← nil
(10)      LibInf.flag ← 2
(11)      GetInfFromCoord(Index)
(12)      LibInf.pos ← Index.pos
(13)      LibInf.pos ← Index.ckptname
(14)      StoreToFile(LibInf)
(15)    else
(16)      LibInf.flag ← 1
(17)      StoreToFile(LibInf, pos)
(18)      StoreToFile(LibInf.data)
(19)      UpLoadToCoord(LibInf, Ckptname, pos)
(20)    end if
(21)  end function
```

Algorithm 2: Dynmic link library, code, and mamap data deduplication algorthim.

exists locally, then we can read the corresponding content to restart. Otherwise, we need to get the content from other nodes. Before sending the request message to other nodes, we need to send a request to the coordinator to get the connection information to other nodes. The steps of this algorithm are depicted in Algorithm 3.

At the checkpoint recovery phase, we need to remap program states into memory. We take the dynamic link library as an example to introduce the recovery phase. When reading the dynamic link library from the checkpoint file, we first obtain the header information of the dynamic link library. We get the checkpoint file name of the dynamic link library and check its existence on the local node. If true, we continue to read the content from the checkpoint file. Otherwise, we need to read the corresponding content from other nodes. Like the previous recovery operation, we need to request the coordinator to obtain information about the checkpoint file and finally retrieve its content to restart. When the content of the checkpoint file is read and remapped into memory, the program can be restored to a correct state to run.

5. Experiment

In this section, we evaluate our approach on QingCloud, which is a public cloud service, providing on-demand, flexible, and scalable computing power. In particular, QingCloud is a popular cloud platform, and, due to its per-second charge method, our experiment will be conducted on QingCloud. For simple comparisons, we use two nodes on QingCloud to conduct our experiments. The configuration of the cloud node is dual-core processors and 2 GB of RAM. The system ran 64-bit ubuntu 14.04. The two nodes are connected by LAN. Experimental programs are divided into two categories: single-node programs and distributed programs across the nodes of a cluster.

For single-node programs experiments, we use the following applications: Python (2.7.6) an interpreted, interactive, object-oriented programming language; Vim (7.4) interactively examining a C program; BaseLine, a simple C application, whose function is to print a 32-bit integer value every one second; BC (1.06.95) an arbitrary precision calculator language; Emacs (2.25) a well known text editor; and Perl (5.18.2) Practical Extraction and Report Language interpreter. To show the breadth, we present checkpoint times, restart times, and checkpoint sizes on a wide variety of commonly used applications. These results are presented in Figures 2, 3, and 4.

For distributed programs, the checkpoint will be written to local disk. We use the MPI package for our experiments. We use the following programs to implement our experiments: NAS NPB3.3 and BaseLine. The BaseLine is a simple MPI program that calculates Pi. The NAS Parallel Benchmarks (NPB) are a small set of programs designed to help evaluate the performance of parallel supercomputers. Problem sizes in NPB are predefined and indicated as different classes. The benchmarks run under MPICH2 are BT (Block Tridiagonal, level C), SP (Scalar Pentadiagonal, level C), EP (Embarrassingly Parallel, level C), LU (Lower-Upper Symmetric Gauss-Seidel, level C), MG (Multigrid, level C), and IS (Integer Sort, Level C).

```
(1)  function UndedupHeapAndStack(fd)
(2)    Mpiflag ← GetProgramType()
(3)    AreaInf ← nil
(4)    NodesInf ← nil
(5)    NodesFd ← nil
(6)    if Mpiflag = true then
(7)      CreateServerThread()
(8)      Register()
(9)      GetNodesInfFromCoor(NodesInf)
(10)   else
(11)     return SingleUnDodedup(fd)
(12)   end if
(13)   while ReadArea(fd, AreaInf) do
(14)     if AreaInf.ckptname = MyCkptName() then
(15)       ReadAndRemap(fd, AreaInf)
(16)     else
(17)       if NodeFd[AreaInf.ckptname] ≠ nil then
(18)         ConnFd ← 0
(19)         ConnFd ←
       ConnectNode(NodesInf, AreaInf.ckptname)
(20)         NodesFd[AreaInf.ckptname] ←
       ConnFd
(21)       RemapFromOtherNode(NodesFd, AreaInf)
(22)     end if
(23)   end if
(24)  end while
(25)  end function
```

Algorithm 3: Restart.

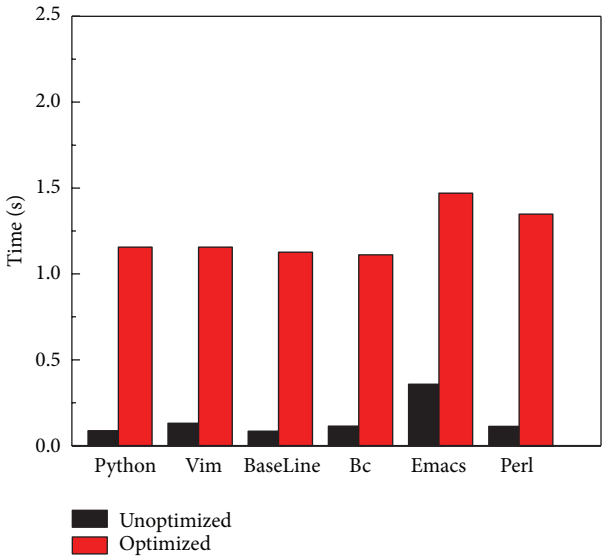


Figure 2: Checkpoint time of single-node programs.

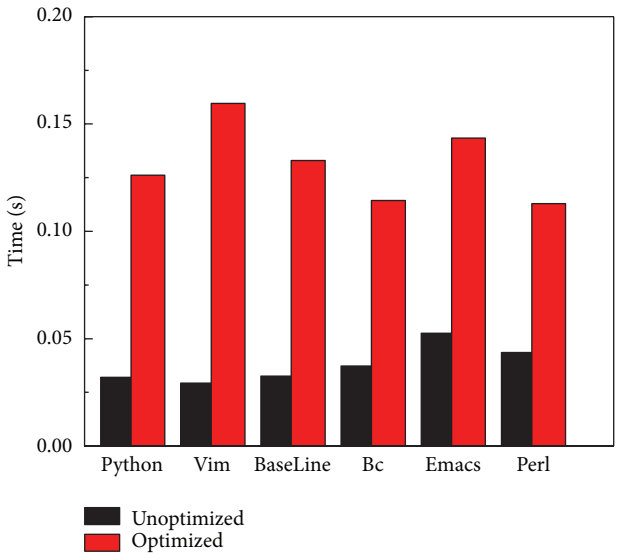


Figure 3: Single-node programs restart time.

We report checkpoint times, restart times, and checkpoint file sizes for a suite of distributed applications. In each case, we report the time and file size when no compression is involved. The experiment was repeated five times. In Figure 7, the checkpoint file size includes the sum of the checkpoints on the two nodes.

6. Experimental Analysis

The graphs in Figure 2 show that, as compared to the original DMTCP, the optimized version increases the checkpointing time. Using the data deduplication algorithm and checking the application category are two main reasons for the increase of time consumption. The checkpointing times under the

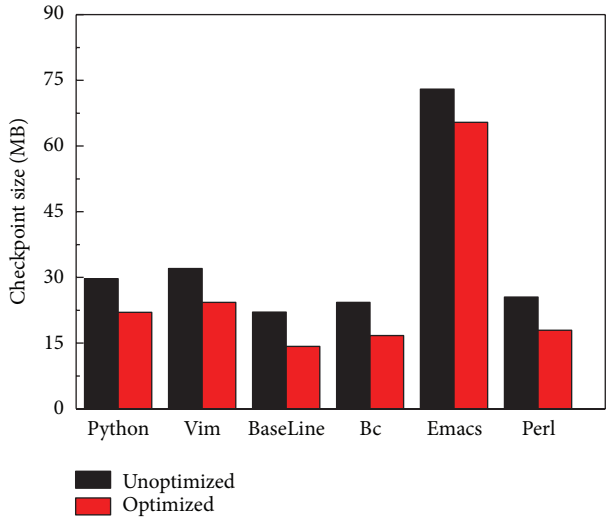


Figure 4: Single-node programs checkpoint file size.

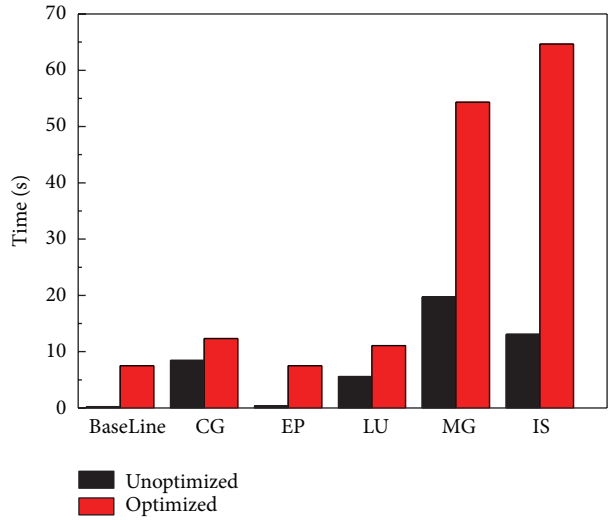


Figure 5: Distributed programs checkpoint time.

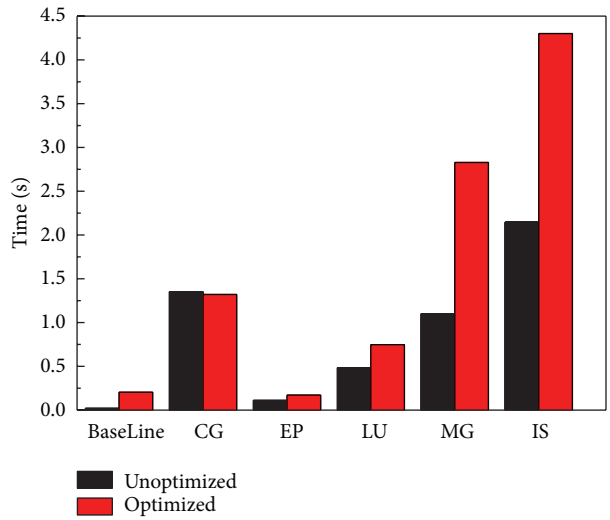


Figure 6: Distributed programs checkpoint restart time.

original DMTCP are 0.08, 0.13, 0.08, 0.11, 0.35, and 0.11 seconds for Python, Vim, BaseLine, Bc, Emacs, and Perl, respectively. As a comparison, for the optimized DMTCP, the checkpointing time increases, such as 1.15 seconds for Python, 1.56 seconds for vim, 1.12 seconds for BaseLine, 1.11 seconds for Bc, 2.9 seconds for Emacs, and 1.34 seconds for Perl.

Figure 3 presents the restart time. In the restart time, For the original DMTCP, the time consumption for the restart operation is 0.03, 0.02, 0.03, 0.03, 0.05, and 0.04 seconds for Python, Vim, BaseLine, Bc, Emacs, and Perl, respectively. As in the case of the checkpoint phase, the restart time also increases. The restart time for each application is increased by about 0.1 seconds, in order to check the type of application to communicate with the coordinator, which resulted in additional costs.

In Figure 4, we can see that the checkpoint file size of all applications is different between the original DMTCP and optimized DMTCP. For single-node programs, we only remove redundant data in the stack, but the final effect is still good. Python's checkpoint file size using the original DMTCP is 29 M and the file size for the optimized version is only 21, indicating about 25% saving. Similarly, the file sizes of other programs in optimized DMTCP are all reduced nontrivially. For example, Vim is reduced by 24%; BaseLine is reduced by 35%; Bc is reduced by 31%; Emacs is reduced by 10%; Perl is reduced by 29%.

Figures 5, 6, and 7 show the results for distributed applications. Similar to the single-node programs, the checkpoint time when using the optimized DMTCP increases for all of the applications. For example, for the case of CG, the checkpoint time is 8 seconds versus 12 seconds. For other test programs, the time is also increased. For example, BaseLine increases 7.2 seconds, EP increased 7 seconds, and LU increased 5.5 seconds. The cost is mainly incurred by the operation of data deduplication. Restart time also increases, and the quantity depends on the duplicate data volume of the various applications.

Figure 7 shows all the checkpoint file sizes of the distributed program. In the original DMTCP, the checkpoint sizes for the testing programs range from 65 MB to 1607 MB. Correspondingly, in the optimized DMTCP, the checkpoint size for BaseLine, CG, EP, LU, MG, and IS is 34 M, 564 M, 36 M, 396 M, 47 M, and 800 M, respectively, indicating a checkpoint size reduction by 47.7%, 49.8%, 47.7%, 49%, 50.4%, and 50%.

Through the above experiments, we can conclude that, by using our method, despite the increase of checkpoint and restart time, the checkpoint file size can be greatly reduced. For the distributed applications, the effect is much more prominent. In the experiment, single-node program mainly reduced the stack segment of redundant information. But, for distributed applications, we reduce the redundant information in the heap and stack segments between different processes. Moreover, the code, dynamic link libraries, and the

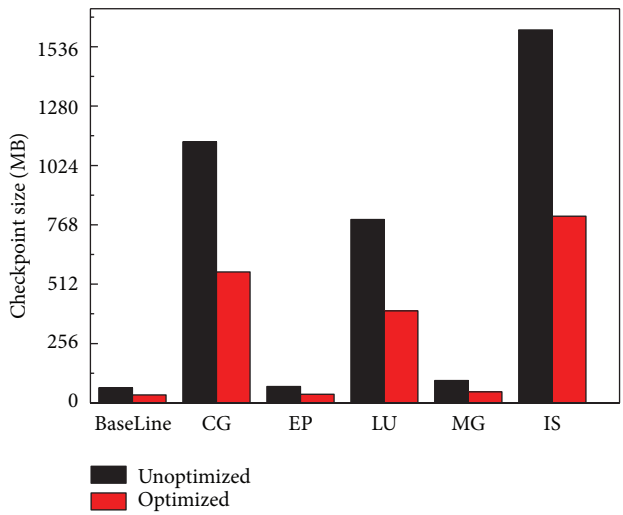


Figure 7: Distributed programs checkpoint file size.

contents of shared memory in the same node are stored in the checkpoint file of a process, while the remaining checkpoint files are only stored in the content index. This is the effect of the experimental program distributing better the single-node program.

In this paper, the experimental program will be divided into two types: single-node program and distributed program. When checking the type of the program, process of each node needs to wait until the coordinator collects information of all processes, which will lead to some of the processes being blocked. When the checkpoint is set, the heap and stack segment uses data deduplication technology, which will be segmented data block. In the distributed application, the data deduplication technology will communicate with the coordinator to query block information. In the checkpoint restart phases, each node containing unique block information needs to create a listener thread, which is used to monitor requests from other processes. Above all, these operations will make extra time overhead; of course, there are some other operations that will also have time overhead. We plan to improve our approach to reduce the time overhead in the future.

7. Conclusions and Future Work

In this paper, we conduct a detailed analysis about the data redundancy in checkpoint files and the potential of utilizing this finding to optimize checkpointing systems. Based on the findings, we propose the design and implementation of a system, which leverages inline data deduplication to achieve the goal of reducing the size of checkpoint file. We perform extensive experiments on a wide range of single-node and distributed applications, and the results demonstrate the effectiveness of our system that is more prominent for distributed applications. However, the results also indicate that there are rooms for improvement in time consumption, which we plan to address in future work.

Competing Interests

The authors declare that they have no competing interests.

Acknowledgments

This research was supported in part by the National Science Foundation of China under Grants 61272190 and 61572179, the Program for New Century Excellent Talents in University, and the Fundamental Research Funds for the Central Universities of China.

References

[1] M. Armbrust, A. Fox, R. Griffith et al., "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[2] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: high availability via asynchronous virtual machine replication," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pp. 161–174, San Francisco, Calif, USA, 2008.

[3] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: current technology and future trends," *Computer*, vol. 38, no. 5, pp. 39–47, 2005.

[4] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.

[5] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems," in *Proceedings of the 18th Annual International Conference on Supercomputing*, pp. 277–286, ACM, July 2004.

[6] N. Naksinehaboon, Y. Liu, C. Leangsuksun, R. Nassar, M. Paun, and S. L. Scott, "Reliability-aware approach: an incremental checkpoint/restart model in HPC environments," in *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGRID '08)*, pp. 783–788, IEEE, Lyon, France, May 2008.

[7] K. B. Ferreira, R. Riesen, P. Bridges, D. Arnold, and R. Brightwell, "Accelerating incremental checkpointing for extreme-scale computing," *Future Generation Computer Systems*, vol. 30, no. 1, pp. 66–77, 2014.

[8] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley, "Memory exclusion: optimizing the performance of checkpointing systems," *Software—Practice and Experience*, vol. 29, no. 2, pp. 125–142, 1999.

[9] D. Ibtesham, D. Arnold, K. B. Ferreira, and P. G. Bridges, "On the viability of checkpoint compression for extreme scale fault tolerance," in *Euro-Par 2011: Parallel Processing Workshops*, pp. 302–311, Springer, 2012.

[10] X. Lin, G. Lu, F. Douglis, P. Shilane, and G. Wallace, "Migratory compression: coarse-grained data reordering to improve compressibility," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14)*, pp. 257–271, USENIX Association, 2014.

[11] M. Lillibridge, K. Eshghi, and D. Bhagwat, "Improving restore speed for backup systems that use inline chunk-based deduplication," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13)*, pp. 183–198, San Jose, Calif, USA, February 2013.

[12] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, article 18, USENIX Association, 2008.

[13] L. Valiant, *Center for Research in Computing Technology*, Harvard University, Cambridge, Mass, USA, 1994.

[14] N. Mandagere, P. Zhou, M. A. Smith, and S. Uttamchandani, "Demystifying data deduplication," in *Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion*, pp. 12–17, ACM, Leuven, Belgium, 2008.

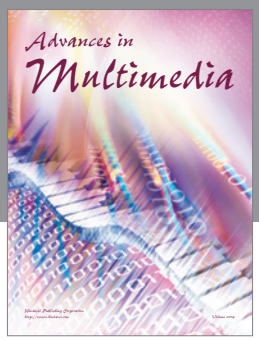
[15] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti, "iDedup: latency-aware, inline data deduplication for primary storage," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*, vol. 12, pp. 1–14, San Jose, Calif, USA, February 2012.

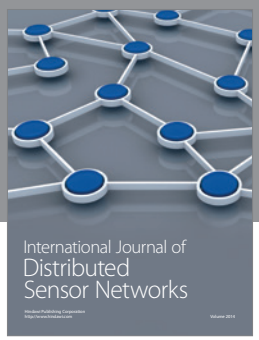
[16] B. Agarwal, A. Akella, A. Anand et al., "Endre: an end-system redundancy elimination service for enterprises," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI '10)*, pp. 419–432, 2010.

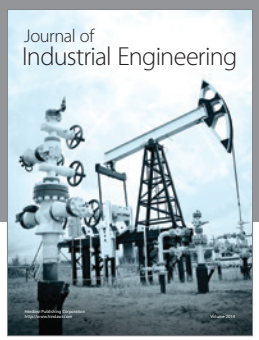
[17] M. Litzkow and M. Solomon, *Supporting Checkpointing and Process Migration Outside the Unix Kernel*, 1992.

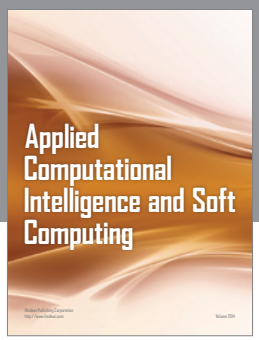
[18] J. Duell, *The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart*, Lawrence Berkeley National Laboratory, Berkeley, Calif, USA, 2005.

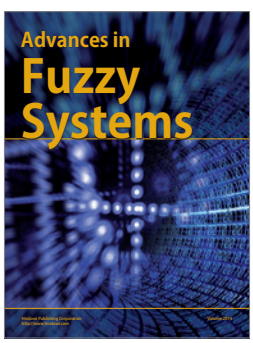
[19] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: transparent checkpointing for cluster computations and the desktop," in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS '09)*, pp. 1–12, IEEE, Rome, Italy, May 2009.

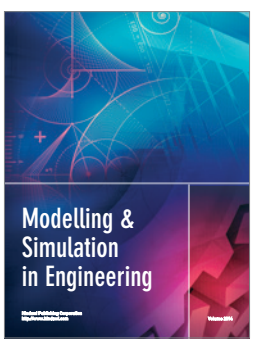


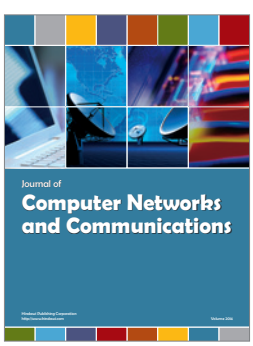












Hindawi
Submit your manuscripts at
http://www.hindawi.com