2009

Extending Scojo-PECT by migration based on application level checkpointing

Jiaying Shi
University of Windsor

Extending Scojo-PECT by migration based on application level checkpointing

By

Jiaying Shi

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2009

Your file Votre référence
ISBN: 978-0-494-57636-6
Our file Notre référence
ISBN: 978-0-494-57636-6

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

In parallel computing, jobs have different runtimes and required computation resources. With runtimes correlated with resources, scheduling these jobs would be a packing problem getting the utilization and total execution time varies. Sometimes, resources are idle while jobs are preempted or have resource conflict with no chance to take use of them. This greatly wastes system resource at certain degree.

Here we propose an approach which takes periodic checkpoints of running jobs with the chance to take advantage of migration to optimize our scheduler during long term scheduling. We improve our original Scojo-PECT preemptive scheduler which does not have checkpoint support before. We evaluate the gained execution time minus overhead of checkpointing/migration, to make comparison with original execution time.

To my wife, Xiaoguang Li

And my dear parents

Acknowledgements

I would like to take this opportunity to express my sincere gratitude to my supervisor Dr. Angela C. Sodan. Without her extensive guidance and constant encouragement, I could not finish this thesis. I would also like to thank all committee members, Dr. Roberto Muscedere, Dr. Xiaobu Yuan and Dr. Dan Wu for their valuable time and comments.

My special thanks go to my wife, Xiaoguang Li, for her support and good advices during the two and half years' graduate study.

I would like to thank Lichun Zhu for his great work on modifying and debugging programs; Peiyu Cai for his idea and help on assuming algorithm; Xijie Zeng, Wei Jin and Xiaorong Cao for the happy experience during the time I worked with them.

Last, but not least, I would like to thank all of my friends for all the help and support during the completion of this thesis.

Table of Contents

List of Tables

List of Figures

1. Introduction

In high performance computing area, a job has one or more processes, like serial job and parallel job. People cared about how to schedule the jobs that would bring more benefit. There are three metrics to evaluate a schedule strategy. At users' view, *response time* defines how fast the job has been processed. It is the time period between job submission and termination. Usually we use average response time instead of each individual one. As some situations did exist when average response time got good result, while several specific jobs were severely delayed. So, another metric *fairness* would be counted in. It is the factor to check whether the job is fairly or unfairly treated. It sets up a threshold to compare a job's response time and its estimated response time as it's been submitted. At systems' view, *utilization* defines how efficient the resources have been taken advantage of. It could be counted as a relation between used and total resource during a time scale. Better utilization would lead to shorter response time but not guarantee in every case. Thus, sometimes there is a tradeoff between response time and utilization.

There are two basic types of job scheduling approaches: time sharing and space sharing. For *time sharing*, multiple jobs are allocated to run on the same set of processors. Multiple processes share using a processor with time slices. The advantage of time sharing is that jobs can start to run sooner (maybe result as a shorter response time). Regarding *space sharing*, processors are divided into groups and each group exclusively allocated to a job, by means of this way, job could run until finish without been suspended. Space sharing is easy to implement without any context switch overhead. Unfortunately, inefficient packing schemes usually generate fragmentations. Some scheduling methods are proposed to improve the performance of time and space sharing, e.g. backfilling, by which a job is scheduled to run out of its original FCFS (First Come First Serve) order to fill the "holes".

Checkpointing mechanism is a term we often used in database and high-performance computing area. It keeps the current program's computation state into stable storages for the purpose of recovery once occur failure and get restart with minimum loss of computing work. We extended the original scheduler to take use of checkpoint mechanism not for fault tolerance but for the resource flexibility by storing the computation state periodically. As the original scheduler does not support the checkpoint and it only does preemption (suspend jobs when slice ends and resume on the same resource in its own type slice), it greatly restricts the usage of computation resource even if there is free resource.

A checkpoint of a single process contains the processor's address space and states of its registers. And a global state is required in additional in multi-processor and distributed systems. It has the information to describe the relationship between checkpoints of each processor. To restart from a checkpoint, it just needs to initialize the address space from the checkpoint file and reset the registers.

Two basic types of checkpointing schemes are full checkpointing and incremental checkpointing. With full checkpointing, the system just saves the whole image of current process state into stable storage. And incremental checkpointing, it maintains a list of all the dirty pages of memory which has been modified during the computation since the last checkpoint. When reaching the checkpoint, only those pages on the list will be stored since the remaining pages are just read only variables. The latter approach would save larger space in checkpoint file size and thus reduce the checkpoint overhead greatly.

There is another classification method based on platforms, several major categories of checkpointing are: *Hardware level*, additional hardware incorporates with processor to save state; *System/Kernel level*, operation system will be responsible for taking checkpoint; *Application level*, the checkpoint code is directly inserted into application by programmer or preprocessor. Application level checkpoint (ALC): for this definition, it is a checkpointing technique in application level. It highly depends on programmer who is required to have sufficient knowledge on specific applications. In the abstraction level, it selects relevant core of data in order to reduce the amount of checkpoint file size and achieve more efficient operation by pre-compiler, which we will discuss in background issue for more detail.

In our thesis, we extended original scheduler by adding application level checkpoint. With checkpoint support, the scheduler could take use of migration and get improvement in utilization and average relative response time.

The rest of the thesis is organized as follows. Background issues are discussed in Chapter 2. Chapter 3 introduces original Scojo-PECT. And extended job scheduling algorithm is described in detail in Chapter 4. Chapter 5 presents the simulation, experiments and results analysis. Finally, the conclusion of the thesis is made in Chapter 6 and we discuss a little about future work in it.

2. Background Issues:

Job scheduling in parallel computing area has been studied for a long period. Around the scheduling approaches, there is generally not only the time sharing and the space sharing, but also other mechanisms involved such as preemption and backfill. However, these approaches could partially solve the fragmentation created during arranging jobs phase and make better use of idle resource avoiding making nodes unfairly working(the situation some nodes running busy, some keep idle).

There is now another option to improve the performance of total job scheduling using migration based on checkpointing, which we discussed above. One type is system level checkpointing, and the other is application level checkpointing. For system level checkpointing, University of Wisconsin introduced Condor system [14] which provides important features that: Source code does not need to be modified to take advantage of these benefits. Code that can be re-linked with the Condor libraries then the jobs can produce checkpoints and they can perform remote system calls [14], and Lawrence Berkeley National Laboratory introduced BLCR (Berkeley Lab Checkpoint/Restart) [16], a system-level checkpoint/restart implementation for Linux clusters that targets the space of typical High Performance Computing applications, including MPI. As a kernel module, it has a “callback” interface to allow any library or application code to cooperate in the taking of and restoring from a checkpoint.

For application level checkpointing, Cornell University introduced C3 (Cornell Checkpoint Compiler) [15] system, it executes almost all source code and instruments them to perform application level state saving by insert a potential checkpoint at locations in the application where checkpoints might be taken. And the pre-compiler co-work with the native compiler of hardware platform to interpret the calls from the instrument application.

As those two kinds of checkpointing mechanisms present, which one would be better for high performance computation. What is the exact difference between them?

The differences between System level vs. Application level checkpointing techniques:
One is the checkpoint file size. As we know, the system level checkpointing approach barely has the knowledge of what application it is. It just stores the whole image of process at the time it takes the checkpoint. For application level checkpointing approach, it is been assumed the programmer has sufficient knowledge on applications and could tell what kind of variables are dead, which part of image could be compressed. In this manner, the size of checkpoint file get smaller compare to system one. In [15], authors made comparison between C3 and Condor, used five kinds of applications for testing, it proved ALC could have smaller size compare to SLC, though results varies.

Another difference is the interval setting of application level checkpointing. As the system level, the operation system will handle the checkpoint and decide when to generate a checkpoint, what is more, that kind of checkpoint in system level could be done at arbitrary time as the processes has been preempted with no risk of lacking records. The application level, the application at the beginning based on its own code will have no idea when and how to do the checkpoint operation. So inserting the intervals for application level checkpointing would be better choice for the simulation. In [1][2][3], the authors studied on determining the optimal interval for setting checkpoints in application and made proof. [2][3], are continuous study and the authors in latter made great efforts modifying the formula to let it suits more cases and made more accurate to predict the interval.(the former failed to make prediction on small size of checkpoint file) And in [1], author tried to verify the impacts over the system overall quality of interval setting through many aspects, as they put interval as adjustable parameter and they also pointed out the optimal checkpoint intervals heavily depend on some system parameters, such as the time to place a checkpoint, the time to recover from a fault, the fault arrival rate, which has the similarity with [2][3]. And for this reason, we decided to take advantage of their existing parameter and formula to make it suitable to our research.

As the checkpointing mechanism usually is applied in HPC (High Performance Computing) field for the purpose of fault tolerance, we imported the idea into our work as a solution to let the application have more flexibility choosing free computation resource and reach the goal of improving system performance.

3. Scojo-PECT

The extending work is all based on the Scojo-PÉCT. In this chapter, we are going to discuss the detailed original scheduler including how it works, what kind of scheduling techniques have been adopted and also some priorities of jobs.

3.1 Scojo-PECT Preemptive Scheduler

Scojo-PECT [8] is a job scheduler framework. Scojo-PECT provides coarse grain time slices to avoid the excessive waiting time when jobs getting access to their resource. All running jobs are preempted and divided into three types (short, medium, long) based on their runtimes. This could make the preemption in the disk affordable. They are scheduled in different time slices of their own types.

Scojo-PECT does not require checkpointing [10] support and therefore impose the constraint that preemption jobs without checkpoint could just restart on the same resource instead of migrating to different resources. One slice time of each type was scheduler per time interval which was set based on the resource time share which is predefined. Resource shares [9] can be defined based on specific job mixes, administrator's policies. E.g. it can be set different for different times during the day.

Scojo-PECT can either use EASY or conservative backfilling. Backfilling is an approach to move job ahead in the queue if it does not delay other jobs. EASY backfilling [10] means jobs could be move ahead by not delay the first waiting job. Conservative backfilling [10] requires none jobs in queue to be delayed compare to their schedule position upon submission. Within the scheduler, the job will not dynamically change and jobs per type were typically scheduled in FCFS order with backfilling applied.

Scojo-PECT also provides safe non-type backfilling [8] because the manual division of job into several types may generate fragmentations since job runtimes and job sizes are correlated. Non-type backfilling means preempted or waiting jobs of different slices can be backfill to a slice if they do not delay any jobs of the slice type or their own types.

Scojo-PECT implemented an event-driven simulator to have it co-work with scheduler. The event-based simulator defines four kinds of events with job submission and termination mean new job come into waiting queue and job finish running and release the occupied resources. Slice begin events describe new time slice starts and specific jobs resumed/scheduled to run and slice end events means one slice run out of its time share and all running jobs are preempted.

3.2 Core Scheduling Algorithm

The detailed original algorithm of Scojo-PECT [8] is:

At first, at the end of slice, jobs were preempted at the end of corresponding time slice and preempted job could be resumed execution on the same resource first;

Second, scheduler attempts to schedule jobs of slice type from waiting queue which first fit the free resource;

Then, scheduler tries to backfill (EASY or Conservative) of slice type;

Last, scheduler will try to non-type backfill (e.g. using the medium and long type job in their waiting queue to backfill the fragmentation left in small job type slice).

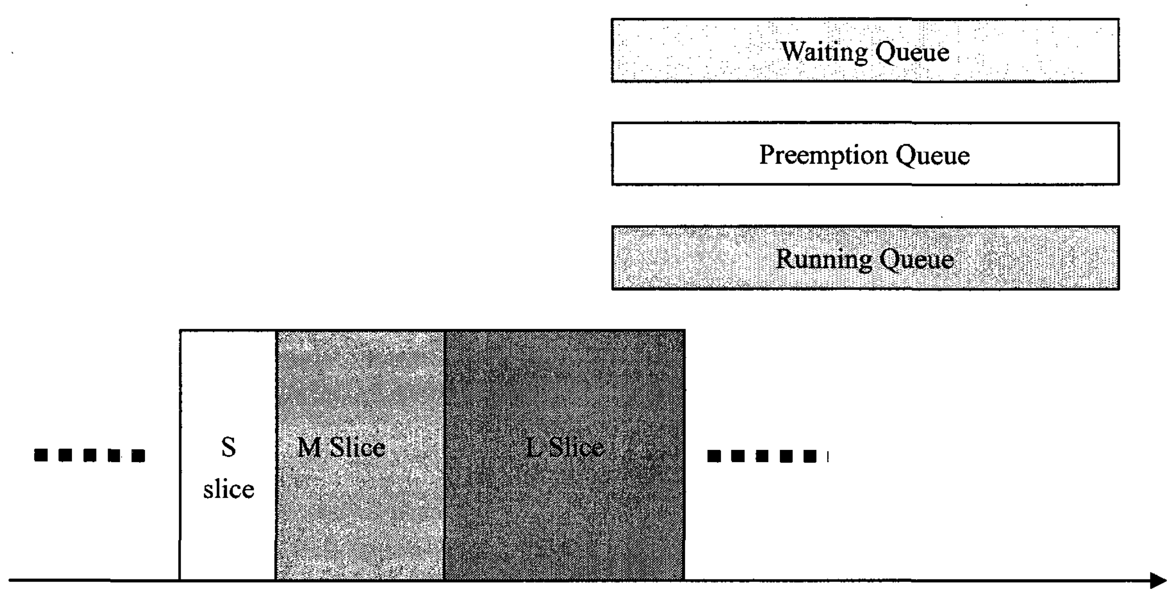


Figure 1. Scojo-PECT Scheduler sketch map

4. Extended Job Scheduling Algorithm

In this chapter, we are going to discuss how we extended the original scheduler. We start talking about the motivation and objective. And we explain the cost model (interval setting, overhead distribution calculation) we have applied in our experiment in detail. At last, we will discuss the extended job scheduling algorithm such as how it works, what extra benefit it could get compared to original scheduler. After that, we will analyze the potential benefit of a simple case by adopting our extending job scheduling algorithm.

4.1 Assumption, Motivation, Scheduling Objective

The extending of applying application level checkpoint is based on the following assumptions:

1. We introduce checkpoint in our simulator as an approach for flexibly using free resource, not for the aim of failure recovery
2. All the jobs randomly generated by Lublin-Feitelson model are independent

As we discussed the original Scojo-PECT, it has feature like preemption while it will restrict scheduler to make decision on using idle resources even there are available. And originally the Scojo-PECT did not have checkpoint support, for this reason, we imported checkpoint/migration mechanism to let the scheduler to better utilize resource much more flexible.

The objective of the scheduler one is to get better average response time based on different workload model. The response time means the time cost between jobs been submit and return back. It generally represents the efficiency of scheduler processing operation. Another is to find out the relation between the average response time and relative factors like average job size, randomly job generation and also job type ratio.

4.2 Cost Model

As we discussed in background issue, there are two major differences between system and application level checkpoint. One is the file size, and the other checkpoint time. For the first, we involved the gamma distribution [11] and we will discuss it in detail at below, and the latter, we borrowed their formula for setting up optimal intervals for application level checkpoint [3].

4.2.1 Trend of Checkpoint File Sizes

The background of this function is our investigation of other peoples' experiment results between system level and application level checkpoint sizes using different categories of program. The most common situation is the application level will reduce the full size of checkpoint files in system level to the average of 50%, which in another way means nearly half space of checkpoint file could be saved by applying the modification in application level [5].While, there are still some specific application required more on the availability of data, in that case, the reduction of checkpoint size between these two levels will be less, vice versa. Two extreme cases are the application level checkpoint files nearly be the same as the system level one; and the application level checkpoint file shrink tremendously and close to 20%, shows in Table1. So we manually set the statistic bound ranges from 20% to 95% of the full size of system level checkpoint file [5][6]. With the trend, as it closes to the low bound 20%, the probability of density is higher, which means there is much higher possibility to reduce the checkpoint file size for at least 20%, and to the other side, high bound 95%, the probability of density is almost none means there is little chance for application level checkpoint reduce the file size for 95%.

Applications	System-Level Checkpoints	User-Defined Checkpoints	Size Reduction
ISING 256	3176	269	91.5 %
ISING 512	3962	1049	73.5 %
ISING 768	5268	2341	55.5 %
ISING 1024	7082	4145	41.4 %
ISING 1280	9408	6461	31.3 %
ISING 1536	12251	9289	24.1 %
ISING 1792	15601	12629	19.0 %
SOR 256	3409	540	84.1 %
SOR 512	4999	2104	57.9 %
SOR 768	7613	4692	38.3 %
SOR 1024	11251	8304	26.1 %
SOR 1280	15994	12940	19.1 %
GAUSS 512	5312	2052	61.3 %
GAUSS 1024	11806	8200	30.5 %
ASP 512	3990	1024	74.3 %
ASP 1024	7230	4096	43.3 %
NBODY 4000	3540	312	91.1 %

Table 1. Relation Chart of Application & System Level Checkpoint [5]

Although we have the trend to represent the probability density, but we could not predict the reduction of file size, so based on the table 1[5], we analyzed the data and formed the chart. To better present the relation and make it precisely, we chose the gamma distribution with the two bounds fixed at 0.2 and 0.95, which totally express and cover the specific case. The trend chart we have created partly from those data [5] and is shown below to indicate the relation between application level checkpoiont file and system level checkpoint file when the file size goes up, see Figure 2.

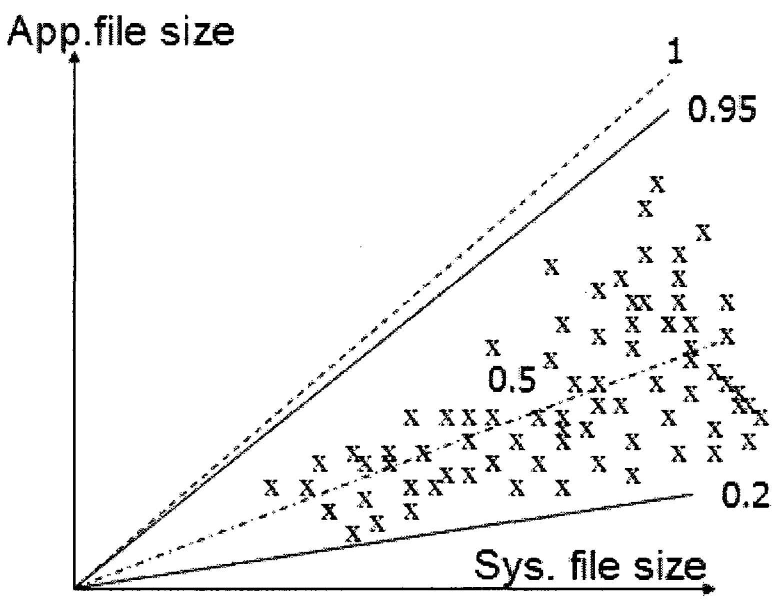


Figure 2 Chart of relation between application and system file sizes

The gamma function is used to let it control the randomly generated figures which represent the checkpoint file size of system level to display the trend which would be close to the gamma distribution, with its parameter fixed. As for gamma distribution [11], in probability theory and statistics, is a two-parameter family of continuous probability distributions. One is the shape parameter k and the other is scale parameter θ. So a random variable x is gamma-distributed with these two parameters could be denoted as

$\alpha \sim \Gamma$ (k, θ). For its probability density function, both k, θ would be positive, it could be expressed as

$$F(x; k, \theta) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \; \Gamma(k)} \quad \text{for } x>0 \text{ and } k, \; \theta>0. \; [11]$$

If k is a positive integer, then Γ (k) = (k-1)!

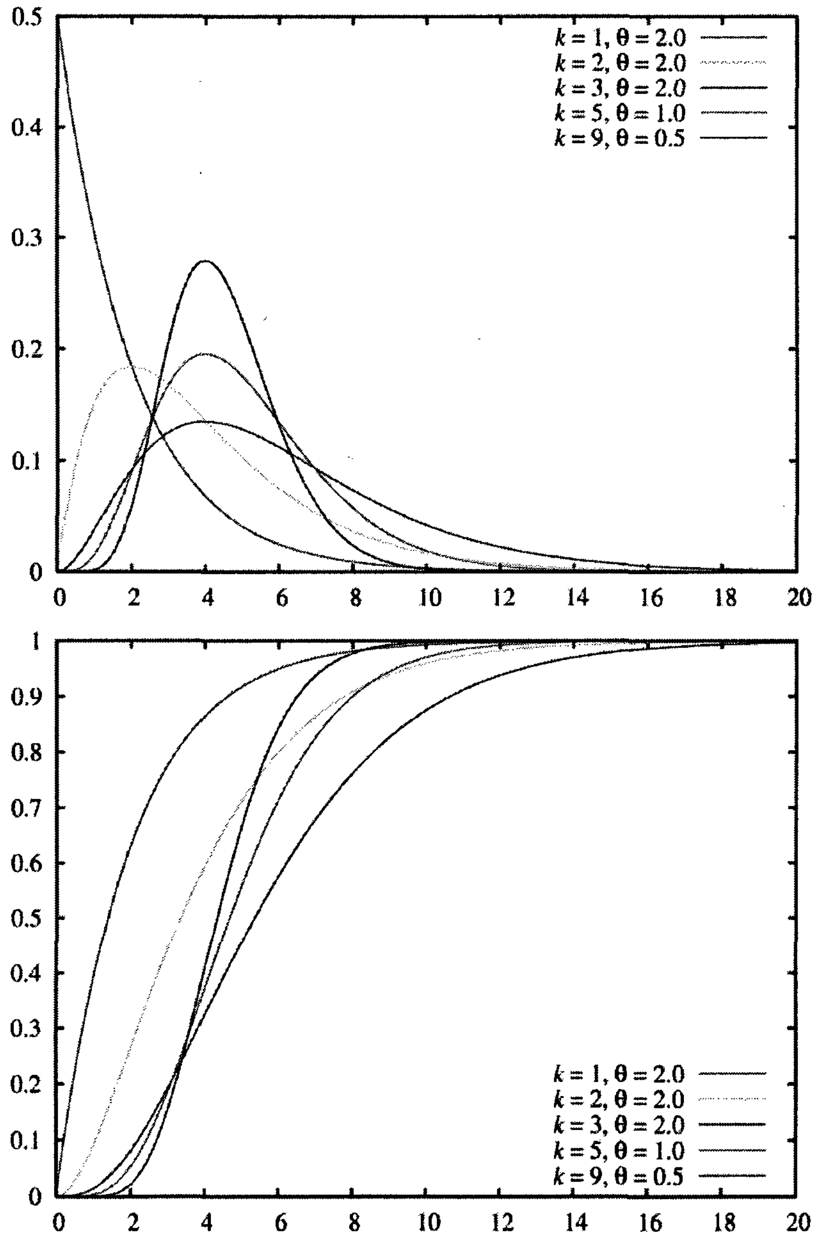


Figure 3. Probability density function [11] & Cumulative distribution function [11]

So back to our prediction of the application level checkpoint file size, based on the function given above, we let S_{app} be the application level checkpoint (ALC) file size and S_{sys} be the system level checkpoint(SLC) file size, combined with the upper and lower bound restriction, we formed our formula:

$$S_{app} = \alpha \times S_{sys} \text{ with } \alpha \sim \Gamma\ (k, \theta)$$

This also could be expressed as:

$$S_{app} = F(x; k, \theta) \times S_{sys} = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \ \Gamma(k)} \times S_{sys} \quad \text{for } x>0 \text{ and } k=2,\ \theta = 1 \qquad (1)$$

Gamma_Low for Cumulative distribution 0.2, which is 0.8274 (2)

Gamma_High for Cumulative distribution 0.95, which is 4.743 (3)

While x = gammaRandom.getnextdouble (), one function in colt.jar (provides a set of Open Source Libraries for High Performance Scientific and Technical Computing in Java.), between Gamma_Low and Gamma_High, based on formulas above, S_{app} would be

$$\left(\frac{\text{x + Gamma\_Low - Gamma\_Low}}{\text{Gamma\_High - Gamma\_Low}} \right) \times Ssys = S_{app} \qquad (4)$$

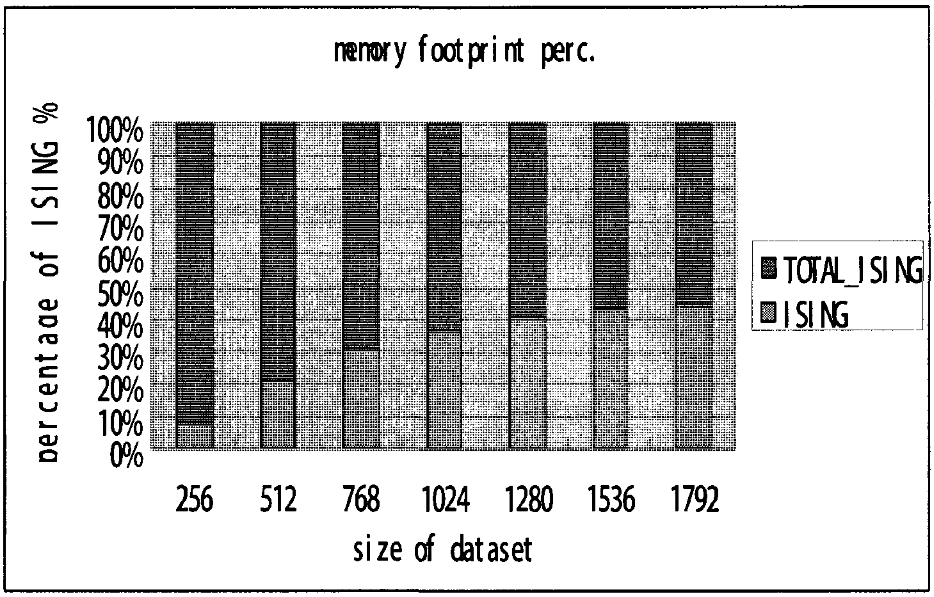
So the checkpoint overhead of ALC would be

$$\textbf{ALC cost} = \frac{Sapp}{IB} + \text{Tcoor},$$

where IB is integrated bandwidth of reliable storage, we put the 0.5 to be the coordination time of processes before the checkpointing starts and for current technology we pick 70MB/s for the write out speed as recent technology. Thus, the formula be

$$\textbf{ALC cost} = \frac{x - 0.8314}{2.9266 \times 70} \times S_{sys} + 0.5,$$

The reason why we pick why we pick the shape parameter k as 2 and the scale parameter θ as 1, we concluded the two specific application and judged different situations while the size changed from Figure 4.



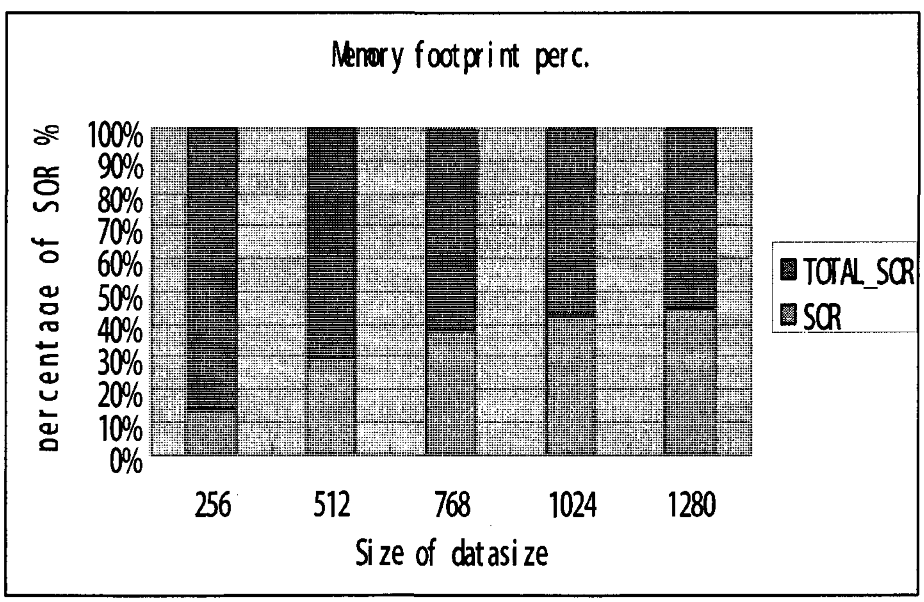


Figure 4. Relation of reduction of file size in ISING and SOR applications

For calculating the low bound and high bound, we used the assistant tool MATLAB, to draw the graph and get the figures at two extreme bounds. Like the Figure 6 below, we settle the cursor on the cumulative distribution function chart and move it to get the right figure at 0.2 and 0.95, then we map the x to the probability density function chart. Let them cover the 20% and 95% of the total area.

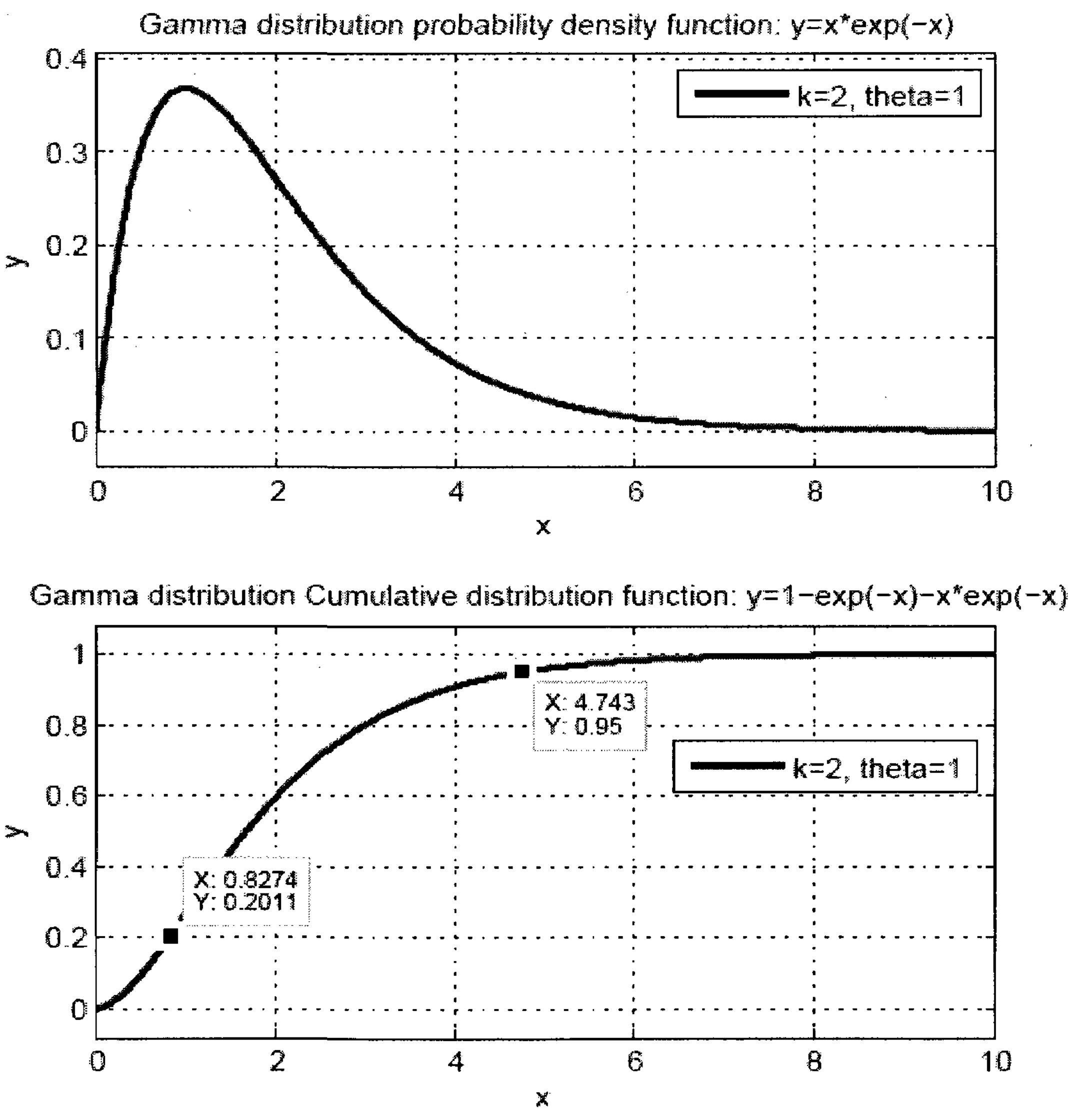


Figure 5. The (PDF) probability density function and (CDF) cumulative distribution function of gamma distribution

4.2.2 Interval Setting

For deciding the interval of checkpointing, several papers [1][2][3] discussed optimal intervals in the context of failure tolerance, the first was brought up in early 1974 given an explicit formula $T_{opt} = \sqrt{2\delta M}$ [1] where δ is the time to write a checkpoint file, M is the mean time between system failures (MTBF, it involves the whole aspects of system such as CPU, memory, power cable etc.), and T_{opt} is the optimum compute time between writing checkpoint files. But its restriction is this model completely fails to predict simulation results for small M. As for that reason, the second paper brought up in 2003, by Young and claimed that $T_{opt} = \sqrt{2\delta(M+R)} - \delta$ [3] to be an excellent estimator of the optimum compute interval between restart dumps for values of $(T+\delta)/M < \frac{1}{2}$ at the end, where R be the restart time. In 2009, *chen et al* [1] pointed out the optimal checkpoint intervals heavily depend on some system parameters, such as the time to place a checkpoint, the time to recover from a fault, the fault arrival rate, and the user specified parameters, like user defined timing constraints. Setting the intervals could influent the system availability and task execution time.

We picked up the first formula to help us setting interval for our simulator. The reason is we need to roughly predict the interval, although the model did not count the failure probability. The MTBF as its definition is the mean time between failure for individual components (e.g., processors, disks, memories, power supplies and networks). A large number of components in system with physical connection between them, this inevitably leads to frequent individual failure. For components mentioned above, they all have an operational lifetime measured in years. There are three different component reliability levels: MTBFs of $10^4, 10^5, 10^6$ hours. In particular, the IBM BlueGene/L system with 65536 nodes is expected to have an MTBF of less than 24 hours [4]. We followed the graph and select the MTBF under the reliability level of 10^5 hours, with 128 nodes involved in our simulation system around 200 hours. So in our case, the optimal checkpoint interval would be set following the formula:

$$T_{opt} = \sqrt{2\delta M} = 20\sqrt{\delta} = 20\sqrt{\frac{Ssys}{70} + 0.5}$$

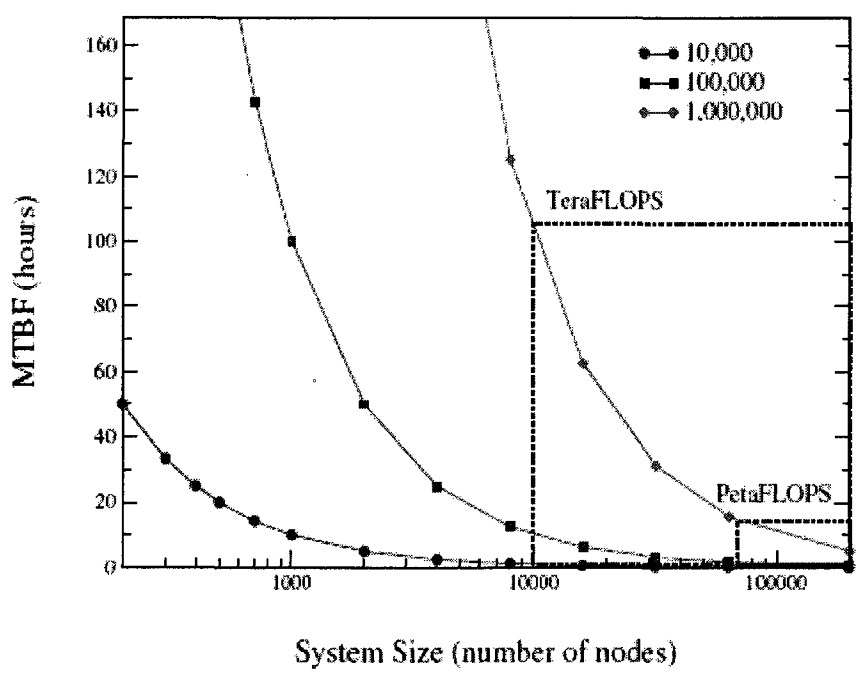


Figure 6. MTBF chart under different reliability level. [4]

For the reason that, the optimal interval is too theoretical and might not be able to suit every cases, thus we try to make our simulation more close to the real situation by adding the normal distribution (Gaussian distribution) on the base interval setting and fix the confidence limit σ^2 to let the probability density located between the confidence limits.

The normal distribution is $T_{\text{int}} \sim N(\mu, \delta^2)$ and after expanded it is the formula with variable x related with μ and σ^2, with μ be the median value of the bell curve and σ^2 be the variance, here we define the σ^2 be around two to let the probability density would reach 99.5% between the variances.

$$p(x) = \frac{1}{\delta\sqrt{2\pi}} \exp\left(\frac{-(x-\mu)^2}{2\delta^2}\right) \qquad \langle\ \mu = 0, \delta^2 = 1\ \rangle$$

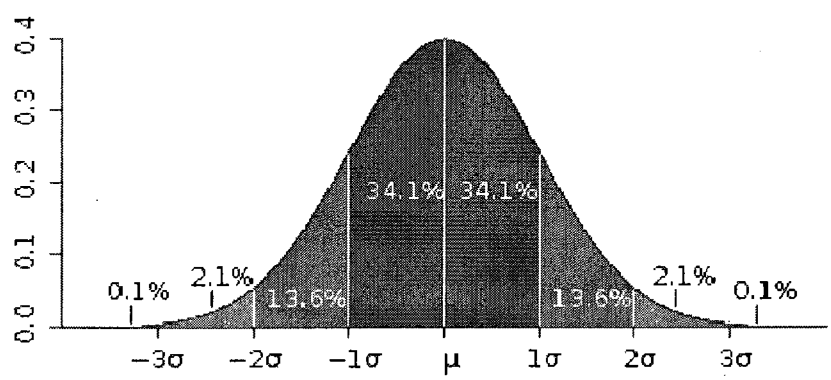


Figure 7. Probability density function of normal distribution [12]

So far, we have done checkpoint overhead part and for migration and restart time we need to clarify them. The migration time is not depending on the number of nodes, but linearly to checkpoint file size. Its cost mainly relies on the time transfer the image from stable storage to target nodes.

$$\textbf{Migration cost} = \frac{Sapp}{W} + T_{min} \qquad [21]$$

This formula was developed by Peiyu Cai [21]. (W is the write speed of stable storage during the migration and for current technique we take 30MB/s, and T_{min} is the base time which represents the basic overhead.) And for restart cost, that time on local node is constant. Its cost relies on putting images to the node memory, once the migration done transfer, the time to restart just short within one second.

4.3 Checkpointing Scheme by Migration with Checkpointing at Application Level

As the original Scojo-PECT does not have checkpoint/migration support, the original idea to introduce migration based on checkpointing is to fill fragmentations which were created during the jobs being scheduled and find out the improvement of the scheduler performance like average relative response time. The original scheduler handles the slice by a series of operations such as release preempted jobs' memory, calculate free resources, search preemption queue, waiting queue and apply backfill strategy if possible.

As Peiyu Cai discovered mainly four cases on system level before [21], we did some investigations on the application level on his basis. After investigating the scheduler, we designed an algorithm and added some policies to take advantage of migration based on checkpointing. We reached a conclusion that to take better use of the free resources and fill up the fragmentations, we could achieve it through backfilling (get jobs in the rear position ahead) and migration (put jobs run on other free resource at the same time or at a later time).

Generally, the application level checkpoint extension has three main stages:

1. Job initialization stage, see Figure 8. We figure out the places where we need to insert our checkpoint intervals (create intervals based on the runtime) and generate an interval list to hold them in time order and meanwhile calculate checkpoint overhead based on the cost model.

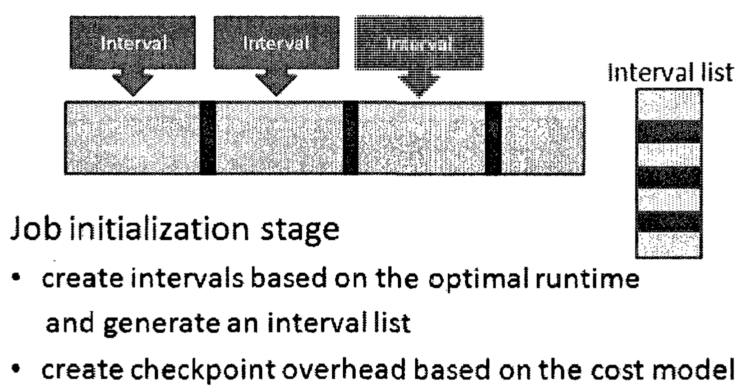


Figure 8 Job initialization stage

2. Create next checkpoint event stage. As the scheduler processed jobs slice by slice and generally jobs can be started either at the beginning or the middle of slice. So we summarized two places we create the checkpoint event. 1) The beginning of slice, at that time all the jobs we prepared to run in the particular slice, each has their own runtime and job size with its starting time. 2) At the start time of jobs, as not all the

jobs start from the beginning of slice.

3. Handle checkpoint event stage, see Figure 10. In this stage, we are going to discover the chance to take advantage of migration based on checkpointing. We have two kinds of queues here dealing with checkpointed jobs and selecting jobs for migration. One is the reserve queue (It holds the checkpointed jobs) and the other is the candidate queue (It holds the jobs which are able to run on the free resources at the checkpoint time).

   When reaching the checkpoint, a couple of things will be done. 1) Stop job and add checkpoint overhead to simulate the checkpoint process. 2) Move the checkpointed job into reserve queue. In reserve queue, a comparison function will pick out the candidate jobs (could run on free resources) for migration and move them from reserve queue into candidate queue. 3) The original scheduler do schedule first (try to find out any non-type backfill), and right away migration operation will be done to let jobs in candidate queue run on the free resources again.

The whole process of the simulation will contain such terms:

Reserve queue, it will hold the checkpointed jobs.

Candidate queue, it will hold the jobs which are able to run on the free resources at the checkpoint time.

Application level interval list, it will hold the interval event based on jobs and sort it by time order.

4.4 Extended Scheduling Algorithm

We only gain benefits based on the migration stage. In this chapter, we will focus on talking about migration phase of handling the checkpoint event stage.

During us dealing with the checkpoint event, we can classify them into several steps, see Figure 9:

The first step: get jobs from running queue and add them to the reserve queue, which is used to hold the checkpointed jobs. And then let the original scheduler do schedule first (The schedule function normally checks preemption queue, waiting queue of own job type and other job types). We only extend the original scheduler and will not affect the original decision.)

The second step: in the reserve queue, we compare the nodes of jobs with the total free nodes in the cluster at the checkpoint time and pick out jobs which are able to run to the candidate queue.

The third step: try to assign free nodes to candidate jobs of candidate queue. For each job, if free resources are not the same as the original ones, do migration; if it is, run the candidate jobs on the original nodes.

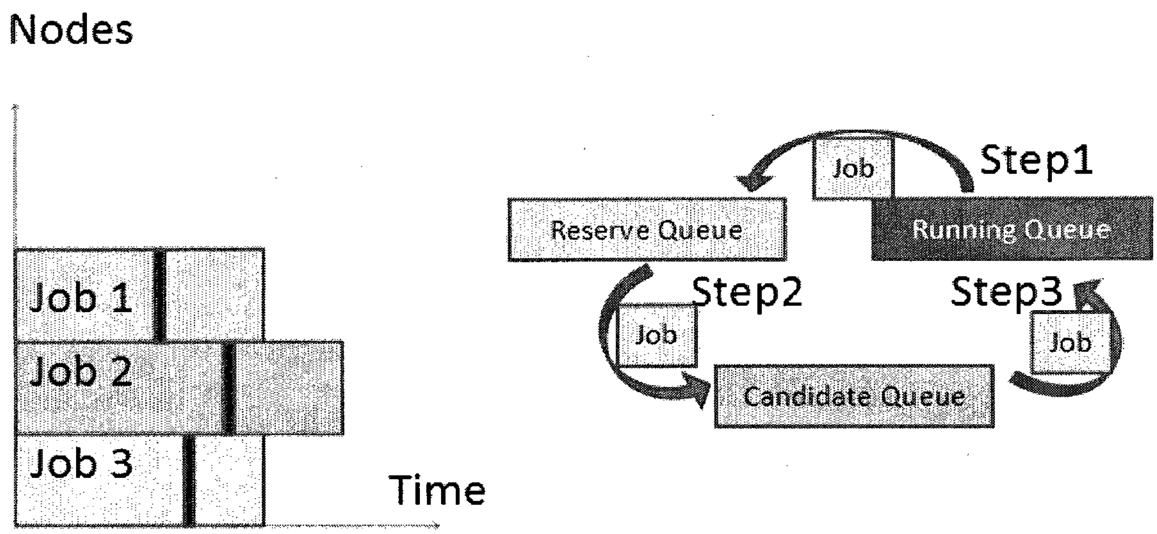


Figure 9 Migration stage

After we extended the original scheduler, we could have more flexibility on using free resources, as we have more chance to backfill and migration. We deal with the situation in Figure 10 like the following:

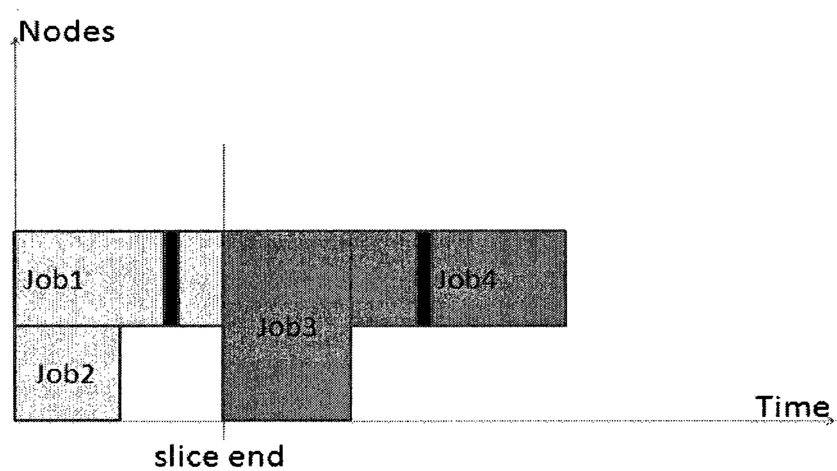
1) Job 2 will finish first, and the scheduler will search the preemption queue, waiting queue of its own type and then waiting queue of other types to see if any jobs could be backfilled. For this case, no other jobs can run.

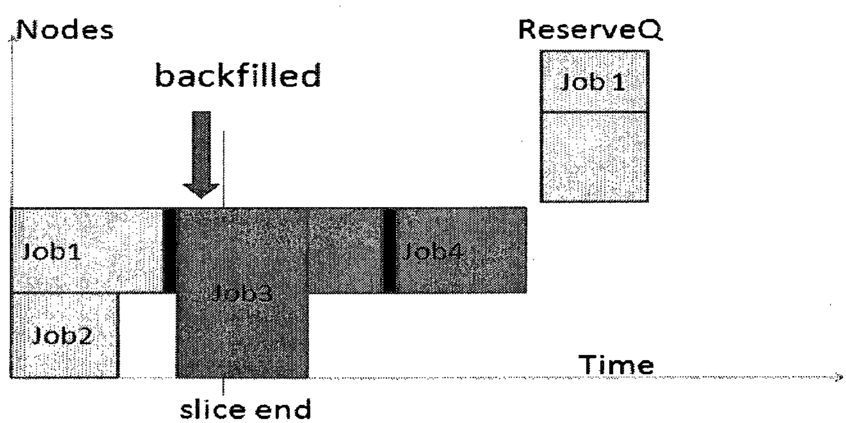
2) Job 1 will have checkpoint, it will be moved from running queue and added to reserve queue, and the original scheduler will do schedule (check preemption queue, waiting queue of own job type and other job types). Later in this case, as the free resource have been used by Job 3 (non-type backfilled), Job 1 will wait in reserve queue.

3) Slice ends and change to another type of slice, Job 3 is preempted but can continue running on its own type slice.

4) Job 3 finishes and it will do the same operations as Job 2 did.

5) Job 4 will checkpoint, at this time it will also be added to reserve queue from running queue, and the original scheduler will do schedule (check preemption queue, waiting queue of own job type and other job types). Then job 1 and job 4 in the reserve queue will be moved to candidate queue because this time the free resource are enough for them to run, and the last they are added to running queue and start to run. Here Job 1 migrates to other resources as Job 4 prefers to run on its original nodes.





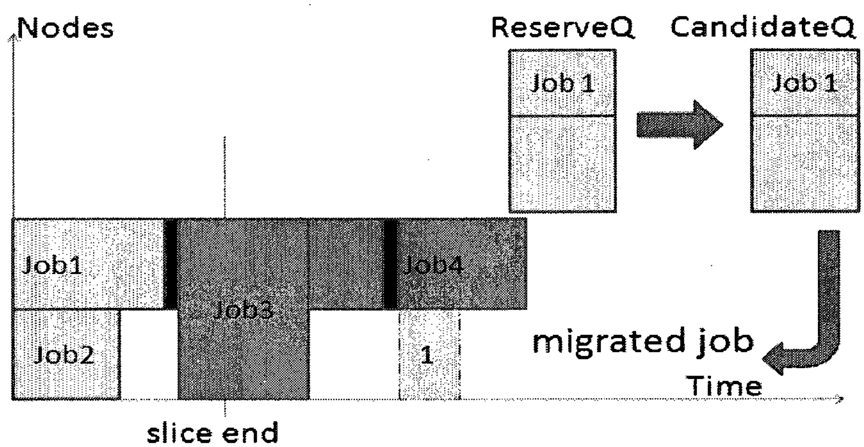


Figure 10 Migration & Backfill

The main algorithm of doing migration is like the following:

```
case (APPCPevent){
    job = getEventJob(APPCPevent);
    stop(job);
    add To reservedQ(job); // reservedQ is a list that holds checkpointed jobs and let them hang on and wait.
    Schedule(); // original Scojo-PECT scheduler function.
} // by here all possible jobs can be backfill'd or non-type backfilled will be backfilled.

for (all job in reservedQ){ // this is try to fill some of the jobs back migration supported.
    if (currentFreeNodes>=jobNodes){
        add to candidateQ(job) // jobs may back to run in own slice, if own position taken by backfilled jobs, perform "in slice" migration.
    }

for (all job in candidateQ){
    if (free Node Position != original Node Postion){
        migrate(job, freeNodes);// "in slice" migration.
```

```
        restart (job);
    }
    else if(freeNodePosition == originalNodePosition){
        run(job) // no job takes its original position. back to run normally.
    }
}
```

For this heuristic of selecting candidate jobs, in our extension, we always pick out the jobs suit best in the candidate queue to fit the resource, and keep assigning resources to jobs until the free resources have been taken used of (here may leave small number of resource idle).

In our simulation, the heuristic for job selection does not affect the complexity of original scheduler. Because the selection heuristic has already been merged into original one which is using easy to fit or called first fit. The adoption of heuristic for job selection did affect the execution time, its original runtime costs thirteen minutes for one run and the extended runtime costs around fifteen minutes for each run.

5. Experiment and Result Analysis

In this chapter, we are going to introduce the experiments on our modified scheduler. The experiment aims to test the fitting of the modified algorithm on handling jobs. We first introduce the experiment environment in 5.1, and then propose the test cases in 5.2 and deliver the testing results in 5.3 and final summarization on the observation from the experiments.

5.1 Experiment Environment Setup

We used our own laptop for the testing environment. Its configurations are:

CPU: Intel Dothan 1.73GHz (One processor)

Memory: 2 Gigabytes DDR 533

Operation System: Microsoft XP SP2.

5.1.1 Workload Modeling

In our simulation, we only use the Lublin-Feitelson statistical workload model [19] which is the best-available synthetic workload model (it includes sequential jobs, correlations between runtimes and sizes, and varying inter-arrival times at different times of the day/night). All workloads comprise of 10000 jobs. The other detailed parameters are listed in table 2.

As in this event-based simulator, jobs are generated based on Lublin-Feitelson Model and the job may take up several slices or just one slice to finish depend on the runtime they submitted. Therefore there would be different circumstances, like a job may end up during the slice or exactly at the end of slice although these situations rarely happen. While we still need to take this into consideration no matter whether it will give the positive affection or the negative affection.

Parameter	**Value**
Machine size	128
Job Class Supported	Small, Medium, Long
Classification Short jobs	Runtime<10 min
Classification Medium jobs	10 min≤Runtime< 3 hours
Classification Long jobs	Runtime≥3 hours
Classification Narrow jobs	Size ≤10% machine size
Classification Medium-Size jobs	10% machine size<size<50% machine size
Classification Wide jobs	size≥ 50% machine size
Switch Overhead	60 sec

Table 2. Scheduler Parameters

5.1.2 Evaluation Plan

The goals of these tests are trying to reveal the relation of relative average response time and several potential factors based on checkpoint/migration mechanism.

Some terms we need to specify related to the results first:

Response time: it defines as the time span from the job was submitted to the job finished.

Runtime: it defines as the time period when the job starts to run and finishes running.

Relative response time: it represent as the ratio for the average response time and the runtime.

In the test, we use different seed parameters. We are going to choose 7, 31, 13, 23 in addition to original one 71. The reason why we are trying to use different seeds number is the seed as a random parameter worked in Lublin-Feitelson model, it control the function when generating jobs, filling them with different correlation of runtime and size. It also changes the memory size of each job. So different seed may have different jobs and get different results. We can see the table 3 for details:

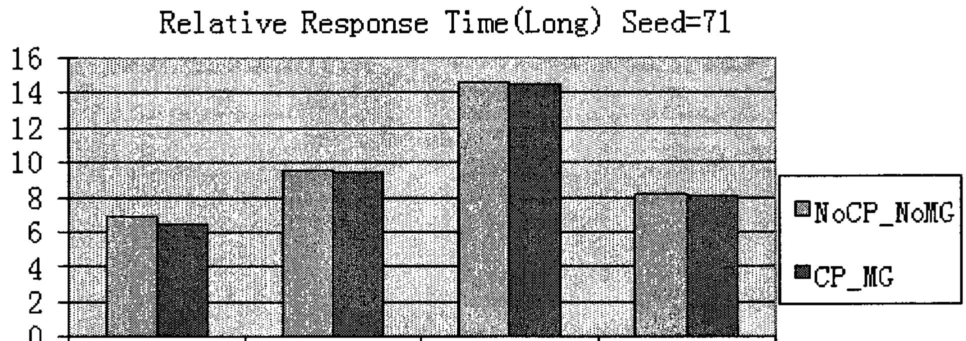
Work Load	% of Jobs			% of Work			Avg. Job Size			Avg. Inter-Arrival Time (sec)
	N_{short}	N_{med}	N_{long}	W_{short}	W_{med}	W_{long}	S_{short}	S_{med}	S_{long}	
Seed 71	64	19.5	16.5	0.5	26	73.5	9	17	19	810
Seed 23	65	19	16	0.5	27	72.5	9	17	20	1038
Seed 31	63	20	17	0.5	25	74.5	9	16	20	860
Seed 13	64	19	17	0.4	26.5	73.1	9	17	20	798
Seed 7	64	20	16	0.5	26	73.5	8	17	21	840

Table 3 Characteristics of synthetic workloads

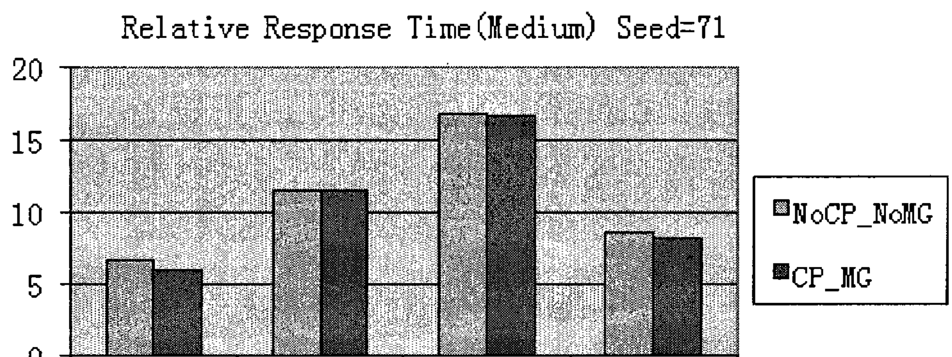
As we implemented checkpoint/migration, we will also make some combinations to test the benefit purely from the migration stage rather than compare the checkpoint/migration stage with the original simulation. Thus, there will be three cases: original, checkpoint without migration, checkpoint with migration.

5.2 Experimental Results

The experimental results for the different cases with different seed numbers are showed in the following figures.



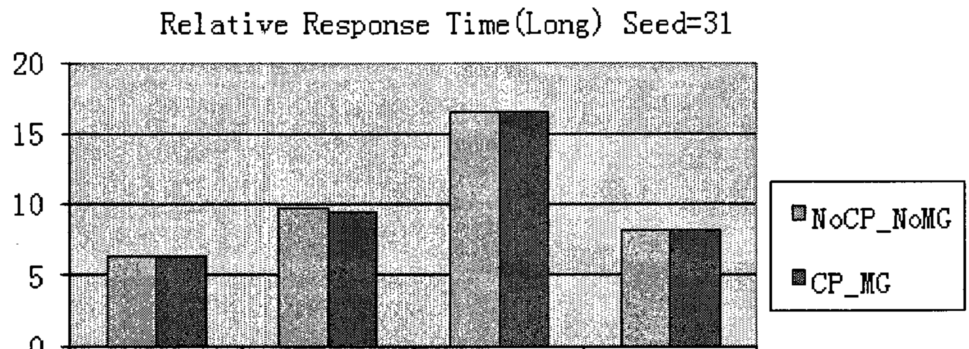
	AVG LN	AVG LM	AVG LW	AVG LA
NoCP_NoMG	6.87	9.53	14.56	8.2
CP_MG	6.48	9.5	14.53	8.02



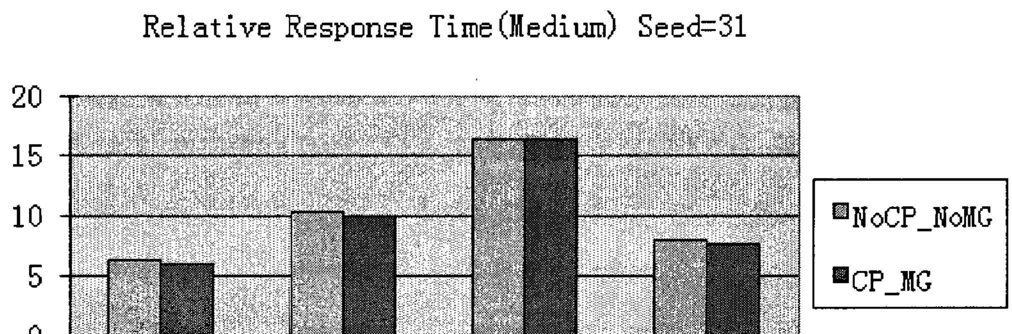
	AVG MN	AVG MM	AVG MW	AVG MA
NoCP_NoMG	6.78	11.6	16.81	8.61
CP_MG	6.04	11.57	16.78	8.23

Figure 11 Relative Response Time for long and medium jobs at seed 71

In Figure 11, we can observe that the overall relative response time get improved 2.20% for long jobs and 4.41% for medium jobs, even with the wide jobs performed very badly with migration strategy.



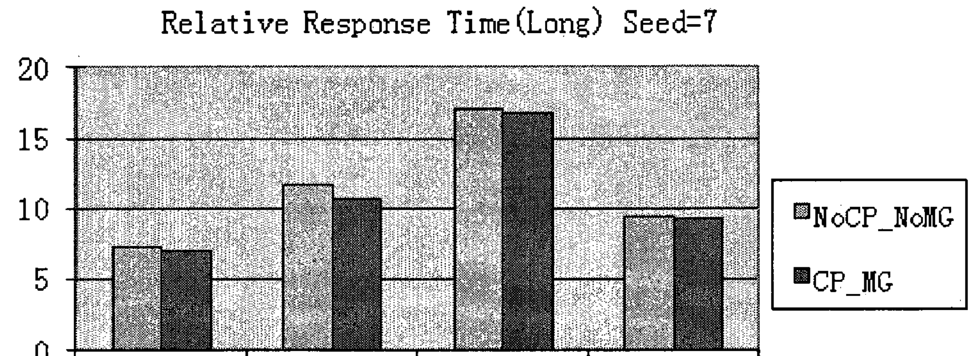
	AVG LN	AVG LM	AVG LW	AVG LA
NoCP_NoMG	6.41	9.8	16.58	8.25
CP_MG	6.35	9.5	16.53	8.15



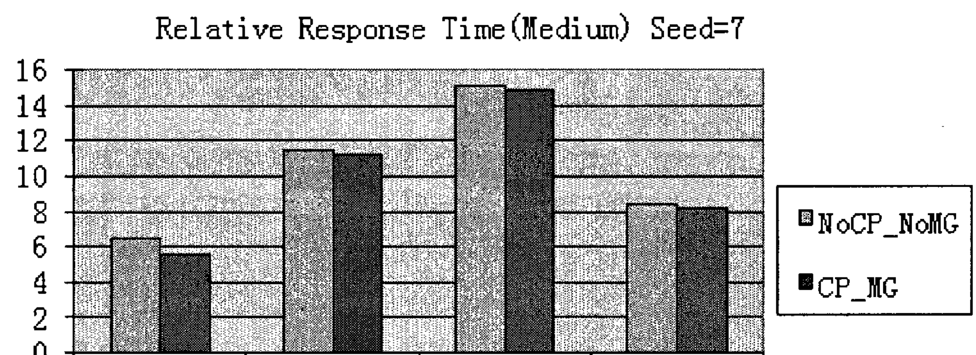
	AVG MN	AVG MM	AVG MW	AVG MA
NoCP_NoMG	6.27	10.42	16.38	8
CP_MG	6.05	10.08	16.36	7.7

Figure 12 Relative Response Time for long and medium jobs at seed 31

In Figure 12, we can observe that the overall relative response time get improved 1.21% for long jobs and 3.75% for medium jobs.



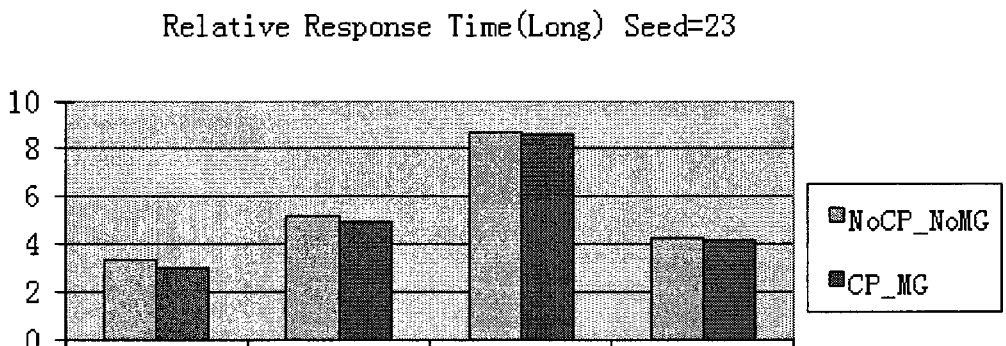
	AVG LN	AVG LM	AVG LW	AVG LA
NoCP_NoMG	7.28	11.72	17.06	9.44
CP_MG	6.99	10.73	16.78	9.34



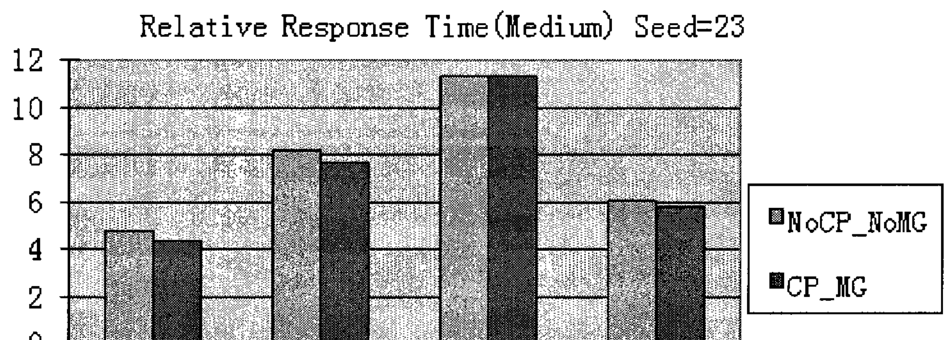
	AVG MN	AVG MM	AVG MW	AVG MA
NoCP_NoMG	6.5	11.48	15.11	8.4
CP_MG	5.61	11.26	14.89	8.23

Figure 13 Relative Response Time for long and medium jobs at seed 7

In Figure 13, we can observe that the overall relative response time get improved 1.06% for long jobs and around 2.02% for medium jobs.



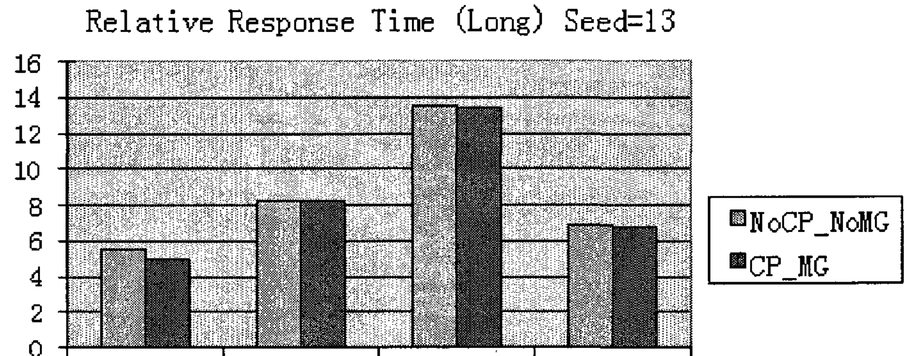
	AVG LN	AVG LM	AVG LW	AVG LA
NoCP_NoMG	3.36	5.16	8.67	4.26
CP_MG	3	4.88	8.65	4.19



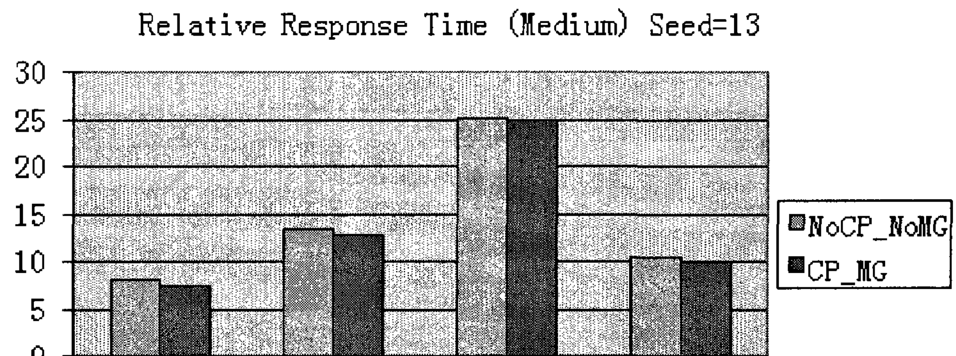
	AVG MN	AVG MM	AVG MW	AVG MA
NoCP_NoMG	4.77	8.19	11.33	6.05
CP_MG	4.34	7.64	11.32	5.8

Figure 14 Relative Response Time for long and medium jobs at seed 23

In Figure 14, we can observe that the overall relative response time get improved 1.64% for long jobs and around 4.13% for medium jobs.



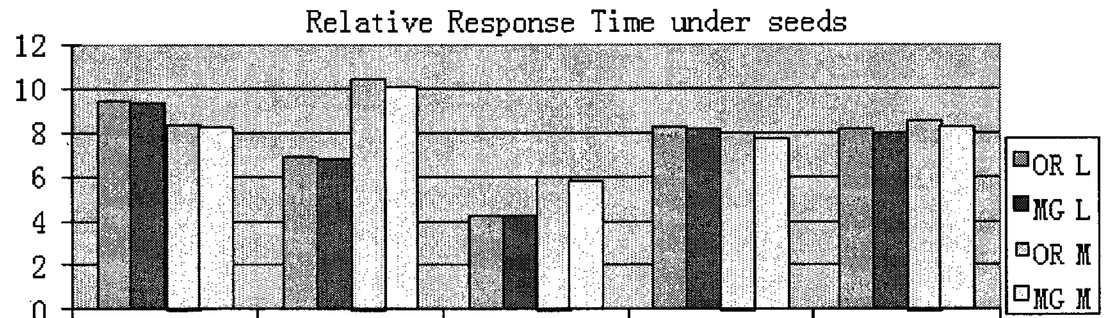
	AVG LN	AVG LM	AVG LW	AVG LA
NoCP_NoMG	5.5	8.27	13.55	6.9
CP_MG	4.96	8.23	13.44	6.78



	AVG MN	AVG MM	AVG MW	AVG MA
NoCP_NoMG	8.18	13.35	25.21	10.52
CP_MG	7.44	12.88	24.97	10.04

Figure 15 Relative Response Time for long and medium jobs at seed 13

In Figure 15, we can observe that the overall relative response time get improved 1.74% for long jobs and around 4.56% for medium jobs.



	seed7	seed13	seed23	seed31	seed71
OR L	9.44	6.9	4.26	8.25	8.2
MG L	9.34	6.78	4.19	8.15	8.02
OR M	8.4	10.52	6.05	8	8.61
MG M	8.23	10.04	5.8	7.7	8.23

Figure 16 Relative Response Time under different seeds

In Figure 16, we can observe that under different seeds the improvement of medium jobs is around 4% at average and long jobs is around 1.5% at average, as the reason long jobs are wider and medium jobs could take better use of checkpointed long jobs.

5.3 Result Analysis

Here we are going to discuss the efficiency of optimization. We will focus on the improvement on the average relative response time under different situations.

The application level checkpoint approach in our simulation generally insert regular checkpoint intervals in each job, which was calculated by program depend on their checkpoint file memory usage and some parameters we discussed before. Simulation is going in slice after slice, in each slice jobs are taking up resource at certain time, and some application level checkpoint intervals may happen at the same time but with very mere chance. Most situations will be one job does the checkpoint operation while other jobs are executing normally.

From the results, we could obtain that wide jobs are not easy to get checkpoint and migrate, as they have to wait for enough resource to have itself to be executed. If we just look at the narrow and medium jobs, the performance will be better. This also means that long jobs will have less chance to take advantage of checkpointed medium jobs, as the long jobs are wider. Also it means the medium jobs will have more chance to take better use of checkpointed long jobs.

6. Conclusion and Future Work

6.1 Conclusion

In this thesis, we have presented a job scheduling approach which employs application level checkpoint and migration scheme on the original Scojo-PECT to tune up the performance. As expected, this approach improves overall relative response time for both medium and long type jobs, especially for the narrow and medium-size jobs. Specially, the thesis has these contributions:

1) We implemented the application level checkpoint/migration as an extension to the original scheduler;

2) We found out that application level checkpoint/migration improved the relative response time a little;

3) We discovered that wide jobs usually are difficult to be migrated after being checkpointed, as it will need to wait for enough free resource to resume its former checkpointed work.

4) Narrow jobs are more suitable to use application level checkpoint as they require very little resources, and thus get greater improvement. The reason is that, as the total resources fixed, narrow jobs are more flexible to take use of free resources than other size jobs. .

6.2 Future Work

In the future work, one of the improvements will be comparing the improvements by selecting different algorithms to choose suitable jobs in candidate queue. Currently we just tried the greedy algorithm and are not sure whether it produces the best combination of jobs.

Other works may be that we try some other impact factors, like we use real traces which have different job type ratio and use different inter-arrival times. What is more, we could also try with smaller job sizes because in the experiments we proved that the wide jobs could not take advantage of application level checkpoint/migration very well.

Reference

[1] Nianen Chen ,Shangping Ren "Adaptive Optimal Checkpoint Interval and Its Impact on System's Overall Quality in Soft Real-time applications" Symposium on Applied Computing , Proceedings of the 2009 ACM symposium on Applied Computing, 2009.

[2] John W. Young "A First Order Approximation to the Optimum Checkpoint Interval" Communications of the ACM, 1974

[3] John Daly "A Model for Predicting the Optimum Checkpoint Interval for Restart Dumps", Springer Berlin / Heidelberg, 2003

[4] Petrini, F. Davis, K. Sancho, J.C."System-level fault-tolerance in large-scale parallel machines with buffered coscheduling", Proceedings of 18th International Parallel and Distributed Processing Symposium, 2004.

[5] Silva, L.M. Silva, J.G. "System-Level versus User-Defined checkpointing", Proceedings on Seventeenth IEEE Symposium 1998.

[6] Schulz, M. Bronevetsky, G. Fernandes, R. Marques, D. Pingali, K. Stodghill, P. "Implementation and Evaluation of a Scalable Application-level checkpoint-Recovery Scheme for MPI Programs", Proceedings of the ACM/IEEE SC2004 Conference Supercomputing, 2004.

[7] Stellner, G. "CoCheck: checkpointing and process migration for MPI" Proceedings of IPPS '96, The 10th International Parallel Processing Symposium, 1996.

[8] B. Esbaugh, A.C. Sodan, Coarse-Grain Time Slicing with Resource-Share Control in

Parallel-Job Scheduling, High Performance Computing and Communication (HPCC), Houston, LNCS 4782, Springer Verlag, Sept. 2007.

[9] A.C. Sodan, "Autonomic Share Allocation and Bounded Prediction of Response Times in Parallel Job Scheduling", Seventh IEEE International Symposium on Network Computing and Applications, July 2008.

[10] A.C. Sodan, Loosely Coordinated Coscheduling in the Context of Other Dynamic Approaches for Job Scheduling—A Survey, Concurrency & Computation: Practice & Experience, 17(15), Dec. 2005, pp. 1725-1781.

[11] Gamma Distribution. http://en.wikipedia.org/wiki/Gamma_distribution

[12] Normal Distribution. http://en.wikipedia.org/wiki/Normal_distribution

[13] A.C. Sodan, Adaptive Scheduling for QoS Virtual-Machines under Different Resource Availability—First Experiences,_Job Scheduling Strategies for Parallel Processing (JSSPP), 2009.

[14] Condor System, University of Wisconsin. http://www.cs.wisc.edu/condor/overview/

[15] C3 (Cornell Checkpoint (pre)Compiler) System, Department of Computer Science, Cornell University, http://www.cs.cornell.edu/~stodghil/papers/ipdps05.pdf

[16] Paul H Hargrove, Jason C Duell, Berkley lab checkpoint/restart (BLCR) for Linux clusters, Journal of Physics: Conference Series 46 (2006), pp. 494-499.

[17] John Paul Walters and Vipin Chaudhary, Application-Level Checkpointing Techniques for Parallel Programs, Springer-Verlag Berlin Heidelberg, 2006

[18] Jos'e Carlos,Sancho Fabrizio,Petrini Kei Davis,Roberto Gioiosa, Song Jiang, Current

Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance, 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), 2005

[19]Feitelson Workload, http://www.cs.huji.ac.il/labs/parallel/workload/logs.html

[20]Colt, Open Source Libraries for High Performance Scientific and Technical Computing in Java, http://acs.lbl.gov/~hoschek/colt/

[21]Peiyu Cai, Extending Scojo-PECT by migration based on system level checkpointing, Thesis work,2009

Appendix

In this appendix, we are going to test the proper migration and backfill functionalities of the extended scheduler in the simulation.

Test Parameters

Description of testing parameters

Job type definition:

Small jobs < 10 min

10min ≤< Medium jobs < 3 hours

Long jobs ≥ 3 hours

Interval setting: 800 sec, the reason we choose 800s here is to make sure the small job will be skipped as we do checkpoint and migration only on medium and long type jobs.

Checkpoint overhead: 3 sec. time costs doing the checkpoint

Average relative response time (ARRT), its definition is the ratio of the average response time compared with its optimum runtime and the smaller number the better result.

Test Example

The job have several attributes, job submit time, job runtime, optimal node for job, job memory size for application level checkpoint approach it also have the two other attributes: the interval and the checkpoint overhead. Like the definition in our simulation

SimpleJob(0, flex, new SimpleTime(0), Serialruntime, Nopt, Nmax, Nmin, Memory size, config)

We manually created 6 jobs and arranged the submission time for each at different points by setting up the parameters after calculation. After analysis the debugging process, we could find the checkpoint/migration operation at the right decision points. This could show that we have employed our implementation correctly according to the algorithm we have proposed before. The detailed can be expressed like below:

<u>Case Codes</u>

```
if(i==0)            //Serial run time/(nodes*0.65) = actual run time
tmpJob =
new SimpleJob(0, flex, new SimpleTime (0), 24876, 128,   128, 128, 10, config);//299
                    else if(i==1)
tmpJob =
new SimpleJob(1, flex, new SimpleTime(80), 24876, 128,   128,   128, 10, config);//299
                    else if(i==2)
tmpJob = new SimpleJob(2, flex, new SimpleTime(100), 32045, 29, 29, 29, 240,
config);//1700
                    else if(i==3)
tmpJob = new SimpleJob(3, flex, new SimpleTime(180), 84500, 99, 99, 99, 1024*3,
config);//1300
                    else if(i==4)
tmpJob = new SimpleJob(4, flex, new SimpleTime(560), 364000, 28, 28, 28, 1024*3,
```

```
config);//20000
                    else if(i==5)
tmpJob = new SimpleJob(5, flex, new SimpleTime(600), 1300000, 100, 100, 100,
1024*3, config);//20000
```

MATLAB Code for Calculation of Checkpoint File Size

Some codes in MATLAB using to draw the probability density function and cumulative distribution function of gamma distribution are shown below:

```
syms x y1 y2;
y1=x*exp(-x);
y2=1-exp(-x)-x*exp(-x);
subplot(2,1,1);
h=ezplot(y1,[0,10]);
title('Gamma distribution probability density function: y=x*exp(-x)');
legend('k=2, theta=1');
ylabel('y');
set( h, 'LineWidth' ,2 );
grid on;

subplot(2,1,2);
h=ezplot(y2,[0,10]);
set( h, 'LineWidth' ,2 );
```

```
title('Gamma distribution Cumulative distribution function: y=1-exp(-x)-x*exp(-x)');
legend('k=2, theta=1');
ylabel('y');
grid on;
```

Vita Auctoris

NAME:	Jiaying Shi
PLACE OF BIRTH:	JiangSu, P.R. China
YEAR OF BIRTH:	1982
EDUCATION:	Houcheng High School, Zhang Jiagang, JiangSu, China 1997 – 2000
	Nanjing University of Technology, Nanjing, China 2000 – 2004
	University of Windsor, Windsor, Ontario, Canada 2006 – 2009