Adaptive Incremental Checkpointing via Delta Compression for Networked Multicore Systems

Itthichok Jangjaimon and Nian-Feng Tzeng
The Center for Advanced Computer Studies
University of Louisiana at Lafayette

The 27th IEEE International Parallel & Distributed Processing Symposium
21 May, 2013
Networked Multicore Systems

- networked multicore system (NMS) is a computing system with group(s) of multicore processing components connected via network

- Large-scale computing system is usually in the form of NMS
  - Cluster
  - Grid computing system
Failures

- Failures are common in big NMS
  - Processor, memory, disk, NIC, etc.
  - Current system faces one failure every 13 hours

Trend

- Hardware example: Processor
  - Shrinking CMOS feature sizes
  - Lowering threshold voltages
  - Hiking power densities

- Scale factor: exascale system is envisioned to fail every 3-26 minutes
Checkpointing

- Saves the states of a running process
- Allow the process to be restarted from that stored state (called a checkpoint)
- Checkpointing overheads
  - Longer execution time
  - Storage space
  - Network bandwidth
Three Main Goals

- Develop Checkpointing that
  - Fast
    - By average, make the application finished faster
  - Low overheads
  - High usability
    - No profiling
Checkpointing

- Usually done periodically
  - The *checkpoint interval* (time between each checkpoints) is calculated from the average *checkpoint latency* (time to finish checkpointing).

- Multi-level checkpointing


- Use different checkpoint types for best performance
- Local checkpoint (e.g., to local hard disk) is not sufficient for node failure.
- Remote checkpoint via network is necessary but expensive

- Checkpoint file size is critical
  - Incremental checkpointing
  - Delta compression (diff)
Incremental checkpointing

- Major portion of checkpoint is the contents of process memory
- Incremental Checkpointing


- Save only new and modified memory pages since the last checkpoint

```
1st Checkpoint
A B C
D E F
G

2nd Checkpoint
A B C
D E F
G H I
```
Delta Compression (diff)

- Save only difference (called *delta*) between each modified pages and its old version

- Size of the difference is called *delta size*

- Time to execute delta compression is called *delta latency*
Delta Compression

- Prior work employs simple delta compression (like XOR-based compression)


- Reason: Job execution is suspended during the delta compression time
Adaptive Incremental Checkpointing (AIC)

- Our approach: Adaptive Incremental Checkpointing (AIC)
  - Concurrent Checkpointing
  - Adaptive Checkpointing
AIC: Concurrent Checkpointing

- **Observed:** *An idle core is frequently available at each node of real world systems*
- Exploit the unused core for delta compression and remote checkpoint concurrently
- Working process continues its execution
- Allow more aggressive delta compression
AIC: Adaptive Checkpointing

- **Observed**: Delta compression performance (delta time and delta latency) are dynamic with respect to the checkpointing moment

- Checkpoint at desirable time (i.e., low costs)

- AIC is the first to consider adaptive checkpointing at the delta compression level
Main Contributions

- Study opportunity for concurrent + adaptive checkpointing with delta compression
- New Markov model for adaptive multi-level concurrent checkpointing
- Implementation and evaluation of AIC for non-MPI jobs
Motivation for Adaptive Checkpointing

- "Does checkpointing moment (checkpoint time) really matter?"
- Measures delta latency and delta time if checkpoint is taken at different point of times for 6 SPEC benchmarks

Normalized delta latency and delta size of three SPEC benchmarks (Sjeng, Lbm, and Bzip2) obtained using our testbed.

95% decrease
Motivation for Concurrent Checkpointing

- "Does an idle core in the same node exist?"
- Analyze usage log of 5 computing systems at Los Alamos National Laboratory (LANL)
  - 5-year logs
  - Over 3 million job records

<table>
<thead>
<tr>
<th>System ID</th>
<th>System Type</th>
<th># of nodes in logs</th>
<th># of cores per node</th>
<th>% of candidate jobs</th>
<th>% of candidate jobs after rescheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>NUMA</td>
<td>1</td>
<td>256</td>
<td>50%</td>
<td>50%</td>
</tr>
<tr>
<td>20</td>
<td>Cluster</td>
<td>256</td>
<td>4</td>
<td>17%</td>
<td>32%</td>
</tr>
<tr>
<td>23</td>
<td>Cluster</td>
<td>5</td>
<td>128</td>
<td>77%</td>
<td>78%</td>
</tr>
<tr>
<td>8</td>
<td>Cluster</td>
<td>164</td>
<td>2</td>
<td>47%</td>
<td>75%</td>
</tr>
<tr>
<td>16</td>
<td>Cluster</td>
<td>16</td>
<td>128</td>
<td>41%</td>
<td>42%</td>
</tr>
</tbody>
</table>
Markov Model for Concurrent Multi-level Checkpointing

- Develop Markov model
  - **Computation core**: program execution, local checkpoint
  - **Checkpointing core**: delta compression, remote checkpoint

- The expected runtime can be calculated
- **normalized expected turnaround time** (NET$^2$)
  - \( \text{NET}^2 = \frac{T}{t} \)
    - \( T \): expected runtime calculated from Markov model
    - \( t \): base process execution time (no failure, no checkpoint)
  - Range from 1.0 to \( \infty \)
  - The lower, the better
Numerical Results for *Static* Concurrent Checkpointing

- Compare our model with Moody
- pF3D – laser-plasma simulation, 1-GB memory per process
- Coastal cluster at Lawrence Livermore National Laboratory (LLNL)
  - 1024 nodes
  - $c_1 = 0.5$ (Local memory), $c_2 = 4.5$ (partner nodes), $c_3 = 1052$ (distributed file system)
Numerical Results for Static Concurrent Checkpointing

- “How many processes can share one checkpointing core?”
- Sharing Factor (SF) – number of application instances that run concurrently on the same node

**Answer:** Up to 15 processes under current system.
Design and Implementation

- Modification of Berkeley Lab Checkpoint/Restart (BLCR)
- Computation core and Checkpointing core
- **Lightweight predictor**: predict checkpoint latencies
  - No profiling
  - Use Stepwise Regression to select predictor variables
  - Adopt Gradient Descent algorithm to adjust the prediction model online based on feedbacks
Design and Implementation

- **Checkpoint Decider**
  - Given predicted checkpoint latencies, decide whether to checkpoint or not

- **Incremental Checkpointer**
  - Track dirty pages by the signal handler
Design and Implementation

- **Page-aligned delta compressor** (*Xdelta3-PA*)
  - Page-by-page delta compression
  - Use *Xdelta3* library (also used in *Rsync*)
  - Delta files are sent for remote checkpoint

![Diagram showing the process flow for the page-aligned delta compressor](chart.png)
Experimental Setup

- Three multi-level checkpointing software
  - **Moody**: static full checkpoints without delta compression
  - **SIC** (Static Incremental checkpointing with Compression): static incremental + concurrent checkpointing + delta compression
  - **AIC**: our adaptive checkpointing mechanism
- Moody and SIC require profiling
- Applications: 6 SPEC benchmarks (processor-memory intensive benchmarks fitting in 1-GB memory)
Experimental Setup

- Compile SPEC with one of Moody, SIC, AIC libraries
- Execute programs on real testbed
  - Collect delta time, delta latency, and local checkpoint latency
- Simulated network bandwidth for remote checkpoint
  - Parameters are taken from Coastal cluster at LLNL
- \( \text{NET}^2 \) can be derived from collected data
  - ranges from 1.0 to \( \infty \)
AIC Performance

- AIC yields lowest $\text{NET}^2$
- $\text{NET}^2$ reduction:
  - Compared with Moody: 8.5% (Sphinx3) to 40% (Milc)
  - Compared with SIC:
    - up to 9% reduction for Milc
    - the gap is larger with higher $\text{NET}^2$ application
The Effect of System Size (AIC vs. SIC)

- System size affects the bandwidth per node available to remote storage.
- **Milc**: $\text{NET}^2$ difference gap widens when the system grows (from 14% to 47%).
- **Sphinx3**: less than 0.5% reduction due to its small file size.

![Bar chart showing NET² of Milc under AIC and SIC](image)
Conclusion

- Observe opportunity for adaptive + concurrent checkpointing
- New Markov model for adaptive multi-level concurrent checkpointing
- AIC implementation and evaluation
  - **Faster**: leads to lower $\text{NET}^2$ (by up to 47%)
  - **Low overheads**: Less than 2.6% overall runtime overhead
  - **High usability**: No profiling
Thank you !!