Transparent, Incremental Checkpointing at Kernel Level:
a Foundation for Fault Tolerance for Parallel Computers

Roberto Gioiosa, Jos’e Carlos Sancho, Song Jiang, Fabrizio Petrini, Kei Davis
Los Alamos National Laboratory, NM 87545, USA
Pacific Northwest National Laboratory, WA 99352, USA

Proceedings of the 2005 ACM/IEEE SC’05 Conference (SC’05)
Outline

• Introduction
• Background
• TICK
  – Design of TICK
  – Algorithm
  – Saving Dirty Pages
• Performance Evaluation
• Conclusions
Introduction – 1/4

• Although the MTBF failures for individual components may be very high, more components will inevitably lead to frequent individual failures.

• Systems should automatically detect, diagnose, possibly without any change to existing application software.
• However, checkpoint implementations are predicated on various simplifying assumptions
  – besides the memory image of the process itself, operating system may contain other essential state.

• The major contributions of our implementation, TICK:
  – **Full transparency** : Doesn’t require any change to applications or system libraries.
  – **Full or incremental checkpointing**
  – **Very low overhead** : checkpointing to memory is less than 4%, checkpointing to disk is less than 6%.
• It is hard to track of processor communication on physical interconnection of nodes.

• We plan to attack the problem by using an innovative methodology called *buffered coscheduling*.

• BCS is that complexity of system software may be greatly reduced by constraining system-wide asynchronicity.
  – There are no messages in transit and where all of the processes can be easily checkpointed.
• Example by other research of BCS (buffered coscheduling)
Checkpointing schemes may be usefully categorized as being *user level* or *system level*.

- **user level**:  
  - The saving of state is performed explicitly by the application, in most cases only with library support.  
  - In any case, determining the optimal frequency for checkpointing may be non-trivial.

- **system level**:  
  - Generally-applicable approach, the application is ‘unaware’ that it is being checkpointed.
The essential properties of TICK:

- **Kernel level:**
  - Implemented at kernel level to allow unrestricted access to processor registers, memory allocation, file descriptors, signals pending, etc.

- **Implemented as a kernel module and user transparent:**
  - Writing, debugging kernel code can be time consuming and non-portable.

- **General purpose:**
  - Processes may be restarted on any node with the same operating environment.

- **Flexibly initiated:**
  - The checkpointing mechanisms of TICK can be triggered by a local event, such as a timer or a remote event.

- **Easy to use:**
  - Provides an interface that can be used by the user to dynamically checkpoint or restart a user process on demand.
There are several approaches to implementing a kernel-level checkpoint/restart mechanism.

- **Internal checkpointing:**
  - **System call** - system call method requires source code modification or library support.
  - **Signal** - An application that frequently sleeps (I/O-bound, for example) may delay indefinitely before executing the signal handler.

- **External checkpointing:**
  - A separate process checkpoints the running App. Our choice of mechanism at kernel level is a kernel thread.
To checkpoint a process, the kernel thread performs the following steps:

1. Stops the running process;
2. Switches to the process address space, if needed;
3. Saves the contents of the registers;
4. Saves the signal status, the signal handlers, and the file descriptors;
5. Saves memory region or dirty pages (incremental);
6. Restores the address space, if needed;
7. Restarts the process.
TICK provides a choice of three mechanisms for incremental checkpointing.

- **Dirty bit:**
  - After a checkpoint, all the writable pages are marked as clean (non-dirty).
  - In order to maintain the correct kernel functionalities, the original dirty bit is mirrored by one unused bits in page table entry.

- **Bookkeeping:**
  - After a checkpoint, all the writable pages are marked as read-only.
  - Page fault exception handler saves the address which App’s going to write in an external data structure.

- **Bookkeeping and Saving:**
  - Same as the previous one except that the page
  - is also copied into a buffer.
Performance Evaluation – 1/5

Overhead of two basic incremental checkpointing mechanisms
### Table 1: Checkpoint Latency

<table>
<thead>
<tr>
<th>Memory Footprint Size</th>
<th>Disk</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Bandwidth</td>
</tr>
<tr>
<td>100MB</td>
<td>1.67s</td>
<td>59.8MB/s</td>
</tr>
<tr>
<td>200MB</td>
<td>3.60s</td>
<td>55.5MB/s</td>
</tr>
<tr>
<td>300MB</td>
<td>5.38s</td>
<td>55.7MB/s</td>
</tr>
</tbody>
</table>

### Table 2: Restart Latency

<table>
<thead>
<tr>
<th>Memory Footprint Size</th>
<th>Disk</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Bandwidth</td>
</tr>
<tr>
<td>100MB</td>
<td>4.39s</td>
<td>22.7MB/s</td>
</tr>
<tr>
<td>200MB</td>
<td>8.176s</td>
<td>22.8MB/s</td>
</tr>
<tr>
<td>300MB</td>
<td>13.18s</td>
<td>22.7MB/s</td>
</tr>
</tbody>
</table>
Number of dirty pages obtained for the application CG Class C using a checkpointing timeslice of 40s.

Number of dirty pages obtained for the application Sage 200MB using a checkpointing timeslice of 20s.
Runtime overhead for Sage and Sweep3D when storing the checkpoints to main memory.

Runtime overhead for Sage and Sweep3D when storing the checkpoints to the local disk.
(c) Runtime overhead for the application Sage 200MB.

(d) Runtime overhead for the application Sweep3D.
Conclusion

• Currently, the mechanism is low-overhead but only suitable for sequential App that does not use interprocess communication (sockets, pipes, FIFOs, and IPC) or dynamic loaded shared libraries.

• We plan to support all of these features as well as checkpointing of MPI parallel applications.