Project Report : Checkpointing on SystemC

Intern: Guanglin Xu
Mentor: Dr. Nicolas Ventroux
Supervisor: Prof. James C. Hoe

Created on 9/6/2012
Revised on 9/23/2012

2012 ECE Summer Internship
Carnegie Mellon University
Content

Project Report : Checkpointing on SystemC................................................................. 1
1. Overview .................................................................................................................... 1
2. Related Works ......................................................................................................... 1
3. Design .......................................................................................................................... 2
   System Components .................................................................................................. 2
   Checkpoint API ........................................................................................................ 2
   SystemC Enhancement ......................................................................................... 3
   MTCP: Checkpoint Core ..................................................................................... 5
   MTCP Interface .................................................................................................. 5
   MTCP Principle .................................................................................................. 5
   MTCP Enhancement ......................................................................................... 8
4. Tutorial ..................................................................................................................... 11
5. Performance ............................................................................................................. 12
6. Artifacts .................................................................................................................. 13
7. Pending Features .................................................................................................... 14
   Incremental Checkpoint .................................................................................. 14
   External File Checkpoint .................................................................................. 15
8. Conclusion ............................................................................................................... 15
9. References ............................................................................................................ 16
1. Overview

SystemC is a C++ based hardware / software modeling language and simulator, which has shortened the developing period of SoC devices since it emerges. However, SystemC doesn't support simulation checkpoint / restore even today. And it is usually a waste of time for modelers to relocate a specified simulation state. In this summer, we add this feature into SystemC. We hope our work will improve SystemC’s productivity.

![Simulation Checkpoint Overview](image)

2. Related Works

To checkpoint a running program, there are currently several approaches at three different abstract levels, from bottom to up, i.e. kernel level, user level, application level.

On SystemC, there are 2 known checkpoint solution. The one adopts application level approach, the other adopts user level approach. However, neither of them is opened to public.

Finally, We decided to adopt user level solution because we can take advantage of DMTCP – an opensource user level checkpoint solution, so as to achieve our goal quickly.
3. Design

System Components

We build a modularized structure in our design.

![Model Diagram](image)

According to the diagram, our work is based on SystemC 2.2.0 ([http://www.accellera.org](http://www.accellera.org)) and DMTCP 1.2.5 ([http://dmtcp.sourceforge.net](http://dmtcp.sourceforge.net)). Each model’s duty is described on the table.

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Duty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enhanced Simulation Kernel</td>
<td>Schedule checkpoint requests, tell MTCP to do checkpoint.</td>
</tr>
<tr>
<td>Enhanced MTCP</td>
<td>Do process checkpoint, write checkpoint image.</td>
</tr>
<tr>
<td>Checkpoint Repository Manager</td>
<td>Maintain limited number of checkpoint image repository.</td>
</tr>
</tbody>
</table>

Table 1 : Model Description

Checkpoint API

We design a series of checkpoint API on SystemC users, split to 3 groups: checkpoint, periodic checkpoint and checkpoint number. All of them are member functions of sc_simcontext. So whenever a SystemC user want to checkpoint, he need to get an sc_simcontext object first.

<table>
<thead>
<tr>
<th>Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td><code>sc_checkpoint()</code></td>
</tr>
<tr>
<td></td>
<td>Checkpoint at next simulation time step.</td>
</tr>
<tr>
<td>void</td>
<td><code>sc_checkpoint(sc_time&amp;)</code></td>
</tr>
<tr>
<td></td>
<td>Checkpoint at a specified simulation time step.</td>
</tr>
<tr>
<td>void</td>
<td><code>sc_checkpoint(double, sc_time_unit)</code></td>
</tr>
<tr>
<td></td>
<td>Checkpoint at a specified simulation time step.</td>
</tr>
</tbody>
</table>
Comments:
1. Duplicating `sc_checkpoint` calling at the same time can duplicate checkpoint behaviors.

<table>
<thead>
<tr>
<th>Periodic checkpoint</th>
<th>void <code>sc_set_checkpoint_period(sc_time&amp;)</code></th>
<th>Set periodic checkpoint interval, referring to simulation time step.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>void <code>sc_set_checkpoint_period(double, sc_time_unit)</code></td>
<td>Set periodic checkpoint interval, referring to simulation time step.</td>
</tr>
<tr>
<td></td>
<td>void <code>sc_set_checkpoint_period(double)</code></td>
<td>Set periodic checkpoint interval, referring to elapsed real time.</td>
</tr>
</tbody>
</table>

Comments:
1. No matter what kind of periodic checkpoint is setup, a checkpoint event will happen at next simulation time step.
2. Now we make a restrict implementation of **elapsed real time periodic checkpoint**, where we can’t change the interval, disable the periodicity or setup **simulation time step periodic checkpoint** once the periodicity of elapsed real time is setup.

<table>
<thead>
<tr>
<th>Checkpoint number</th>
<th>void <code>sc_set_checkpoint_number(sc_dt::sc_digit num)</code></th>
<th>Set number of checkpoint images we will keep.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>sc_dt::sc_digit</code> <code>sc_get_checkpoint_number()</code></td>
<td>Get number of checkpoint images we will keep.</td>
</tr>
</tbody>
</table>

**SystemC Enhancement**

We enhance the SystemC simulation kernel for timed checkpoint and simulation time periodic checkpoint, by adding **checkpoint notification phase** as long as **checkpoint event queue**. The queue is self-sorted. However, the simulation kernel doesn’t schedule elapsed real time periodic checkpoint event. Instead, this kind of periodicity will be processed by MTCP directly. The diagram shows:
Figure 3: Kernel Enhancement

When an sc_checkpoint API is called, a NONE_PERIODICITY sc_checkpoint_event will be inserted to the checkpoint event queue.

When an sc_set_checkpoint_period API is called, a SIMULATION_TIME sc_checkpoint_event or an ELAPSED_REAL_TIME_FIRST sc_checkpoint_event will be inserted to the checkpoint event queue.

We should notice the difference of such events. Diagram shows:

Figure 4: Checkpoint event

When the simulation kernel enters Checkpoint Notification Phase, it fetches all available checkpoint events out one by one, and then invoke an MTCP interface function to do checkpoint. The kernel behavior depends on the periodicity type of the event:

NONE_PERIODICITY:
1. Call do_checkpoint_by_sem(“any comments for debug info”, 0);
2. Delete the event object.

SIMULATION_TIME:
1. Call do_checkpoint_by_sem(“any comments for debug info”, 0);
2. Re-schedule this event at next interval.

ELAPSED_REAL_TIME_FIRST:
1. Call do_checkpoint_by_sem(“any comments for debug info”, m_checkpoint_elapsed_real_time_period);
   The second non-zero param will switch MTCP to elapsed real time periodic checkpoint.
2. Delete the event object.

MTCP: Checkpoint Core

MTCP Interface

MTCP Interface will be used by SystemC kernel.

<table>
<thead>
<tr>
<th>Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>mtcp_init()</td>
</tr>
<tr>
<td></td>
<td>Initialization convention of mtcp, but we don’t require any parameters here.</td>
</tr>
<tr>
<td>int</td>
<td>mtcp_ok()</td>
</tr>
<tr>
<td></td>
<td>Initialization convention of mtcp.</td>
</tr>
<tr>
<td>void</td>
<td>do_checkpoint_by_sem (char* comments, int switch_to_elapsed_real_time_periodic)</td>
</tr>
<tr>
<td></td>
<td>Do a checkpoint immediately by use of semaphore. A non-zero param 2 will cause MTCP to</td>
</tr>
<tr>
<td></td>
<td>use semaphore the last time, and then switch to use sleep() system call for ever.</td>
</tr>
<tr>
<td>void</td>
<td>set_ckpt_num_limit(int limit)</td>
</tr>
<tr>
<td></td>
<td>Set number of checkpoint images we will keep.</td>
</tr>
<tr>
<td>int</td>
<td>get_ckpt_num_limit()</td>
</tr>
<tr>
<td></td>
<td>Get number of checkpoint images we will keep.</td>
</tr>
</tbody>
</table>

MTCP Principle

Checkpoint

MTCP employs a user-level checkpoint technique for multi-threading program. Like the figure, one can integrate MTCP into his program like this:
1. Initialize MTCP;
2. MTCP will setup a signal handler for each thread, so every thread can be blocked when it
needs;
3. MTCP will setup a wrapper for \_clone() system call to track the creation of other threads;
4. MTCP thread sleeps;
5. Main() function (main thread) does something;
6. Main thread pthread\_create() sub-threads, and MTCP gets aware of each of them;
7. Each thread does something;
8. MTCP thread awakes after sleeping, block the other threads, do checkpoint;
9. MTCP finishes checkpoint, wake other threads up;
10. Everything goes on…

Figure 5 : MTCP checkpoint

**Checkpoint Image**

In fact, the checkpoint image is a structural file rather than a mass of memory dump.
1. a magic “MTCP-V1.0” is written on the head;
2. several parameters of the process are written;
3. several function pointers of the process are written;
4. file descriptors are written;
5. all memory regions are written, including executable library, anonymous area, heap and stack.

The structure of these memory regions is read from /proc/self/maps.
For example:

<table>
<thead>
<tr>
<th>MTCP-V1.0</th>
<th>stack_rlimit</th>
<th>CS_RESTOREBEGIN</th>
<th>restore_begin</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_STACKRLIMIT</td>
<td>stack_rlimit</td>
<td>CS_RESTOREBEGIN</td>
<td>restore_begin</td>
</tr>
<tr>
<td>CS_RESTORESIZE</td>
<td>restore_size</td>
<td>CS_RESTORESTART</td>
<td>restore_start</td>
</tr>
<tr>
<td>CS_RESTOREIMAGE</td>
<td>restore_begin</td>
<td>CS_FINISHRESTORE</td>
<td>frpointer</td>
</tr>
</tbody>
</table>

Table 2: MTCP checkpoint image structure
Restart from image

Unfortunately, we still need an external program named mtcp_restart to re-run our process checkpoint image. The reason is a bit complex. One important reason is that the restarting operation needs libc library’s support, but it has to unmap libc.so previously in the mean time! As a solution from DMTCP project, they make a static mtcp_restart program without dependence to any libraries because it contains a lot of dedicated system call wrappers.

Take restarting test.ckpt.1 for example, figure.

![Figure 6: MTCP restart](image)

MTCP Enhancement

Semaphore synchronization

We enhance the way MTCP thread works in. In spite of the above work flow, our MTCP thread can synchronize with main thread or any user threads by use of semaphore. Like figure. This provides us more flexible control.

In particular, MTCP thread just synchronize with one working thread, not all. This may harm the checkpoint semantic because the rest threads go further than the synchronized working thread. But on SystemC, it’s OK because all SC_THREADS give control to simulation kernel at each simulation time step.

But how to switch MTCP to elapsed real time periodic checkpoint? We do it by
implementing the MTCP interface function - do_checkpoint_by_sem (char* comments, int switch_to_elapsed_real_time_periodic). For a non-zero param 2, MTCP will use semaphore the last time, and then will use sleep() system call instead for ever.

Figure 7: MTCP semaphore

Multiple Checkpoint Images

By default, MTCP saves only one checkpoint image, overwriting the old image while writing the new one. More precisely, MTCP firstly writes checkpoint file to a .tmp file, and then rename it by deleting the .tmp suffix.

As we want to keep multiple checkpoint images, we design a CheckpointDirectoryManager class to help us. However, MTCP is written by C language. So we need a c_interface of our supported class.

The only interface function of the c_interface.h is

```c
char* get_next_ckpt_img_path(int ckpt_img_num_limit);
```

When MTCP wants to rename the .tmp file, it calls this function to get a new name of it. Inside, CheckpointDirectoryManager will scan the working directory, and delete unwanted checkpoint images before it returns a new file path.
Bug fixing

When we just got MTCP, we also got upset because most of the checkpoint image can’t be restarted by mtcp_restart. Segmentation faults happened so often!

We’ve tried to get some help from the DMTCP community. However, no one cared us. So we had to investigate the problem by ourselves. After several weeks’ hard work, we finally knew how to fix it.

The reason is, [vdso] is mapped randomly by linux kernel and MTCP can’t control this mapping behavior. Although we’ve said that each memory region of mtcp_restart will be unmapped, it’s not so accurate since [vdso] must be excluded. So if mtcp_restart’s [vdso] is mapped to an area that has been occupied by previous checkpoint image, imposed re-map will cause segmentation fault.

We have no idea how to remove mtcp_restart’s [vdso] in Linux. But fortunately we can take advantage of 2 kinds of memory mapping conventions in Linux. The secret is that one of the conventions will map [vdso] (and the other mappable) higher while the other will map [vdso] (and the other mappable) lower. So we make the initialization of MTCP to imposed higher [vdso] while impose mtcp_restart lower [vdso].

What an interesting thing is that when we’d make a fixing plan, we found this solution had been employed by DMTCP, but, in fact, the developers forgot to do the same thing on MTCP. Anyway, we summited our patch to the DMTCP community and got admiration from the developers of DMTCP.

4. Tutorial

```c
void proc_monitor() {
    if (sc_time_stamp() == sc_time(100, SC_MS)) {
        // checkpoint next step
        sc_get_curr_simcontext()->sc_checkpoint();

        // checkpoint next step, duplicate.
        sc_get_curr_simcontext()->sc_checkpoint();

        // checkpoint at 200ms
        sc_get_curr_simcontext()->sc_checkpoint(sc_time(200, SC_MS));

        // at 20ms, illegal, will be ignored by kernel
        sc_get_curr_simcontext()->sc_checkpoint(sc_time(20, SC_MS));
    }
}

SC_CTOR(Monitor) {
    SC_METHOD(proc_monitor);
    sensitive<<timerEvent;
}
```

Figure 8: Checkpoint use case 1
5. Performance

We measure the performance of our implementations, focusing on the checkpoint time and size.

We take a shift register for example. According to the figure, checkpoint time is less than half second while each checkpoint image keeps the same size of 1.8 MB.
6. Artifacts

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example/</td>
<td>ModelCheckpointing code.</td>
</tr>
<tr>
<td>git_repository/</td>
<td>Git code repository.</td>
</tr>
<tr>
<td>Reference_diffs/</td>
<td>Referenced in this document.</td>
</tr>
<tr>
<td>libsystemc.so</td>
<td>A shared SystemC library with checkpoint support.</td>
</tr>
</tbody>
</table>

Figure 10: Performance measurement

Figure 11: Artifacts
ModelCheckpointing | Executable example.
---|---
mtcp_restart | Checkpoint image restarter, provided by MTCP
SystemC-Checkpoint-README | Instruction.

Table 3: Artifacts description

## 7. Pending Features

### Incremental Checkpoint

If we observe our sequential checkpoint images, we will find that they are almost the same big size. Moreover, after a deeper investigation, we found they save a large of redundant information!

Take a look at a checkpoint image:

![Read Checkpoint Image](image)

Figure 12: Read Checkpoint Image

Yeah, at least, we may have to save the same text segment of each shared library again and again unless we take some measure. Moreover, the other memory areas may remain unmodified since last checkpoint, so we’d better drop them while writing a new checkpoint image.

In fact, MTCP has an experiment feature called HBICT, short for Hash Based Incremental Checkpoint Tool. HBICT is a binary program that cooperates with MTCP by a pipe. It has several disadvantages. Firstly, it can’t work properly with MTCP even today. Secondly, it is inconvenient to be deployed because it is a binary program. Thirdly, it is a waste of time to hash every block when we sometimes know these blocks are the same.
Since it must take more time to design a whole new incremental checkpoint solution, we decided to make use of HBICT this summer. We hoped to:

1. Make it as a thread of MTCP while we need it.
2. Make it work.

We definitely achieved the 2 above goals this summer. But we can take it as a new feature now. The problem is, when HBICT thread works with MTCP checkpoint thread by a pipe, they produce cooperative dynamic state of more than 20 Mbytes that has to be checkpointed. As a result, the effort by HBICT must to be offset.

We don’t know why tcdrain(STDOUT) function call can largely expand the memory map so far.

**External File Checkpoint**

Usually, external files must be larger than the process image. So, external file checkpoint is not so meaningful if we can’t take advantage of incremental checkpoint.

If we still want to checkpoint external files, we can enhance writefiledescrs() in MTCP.

**8. Conclusion**

Figure 13: Heuristic

<table>
<thead>
<tr>
<th>Year</th>
<th>Title</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>5/28 – 6/18</td>
<td>Learn SystemC.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read 3 papers of SystemC checkpointing.</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------------------------</td>
<td></td>
</tr>
<tr>
<td>6/26 – 7/2</td>
<td>Find DMTCP Project, try it out.</td>
<td></td>
</tr>
<tr>
<td>7/2 – 7/9</td>
<td>DMTCP Benchmark</td>
<td></td>
</tr>
<tr>
<td>7/10 – 7/23</td>
<td>Integrate MTCP into SystemC. Benchmark.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Find a deadly bug of MTCP.</td>
<td></td>
</tr>
<tr>
<td>7/24 – 7/30</td>
<td>Explore good measurement tools.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bug of MTCP more clear.</td>
<td></td>
</tr>
<tr>
<td>7/31 – 8/20</td>
<td>Fix the bug of MTCP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Clarify the possibility of incremental checkpoint.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Clarify checkpoint API on SystemC</td>
<td></td>
</tr>
<tr>
<td>8/21 – 9/4</td>
<td>Implementing</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Timeline

Obviously, as an unexperienced undergraduate student, I spent much more time to learn than to work. However, the most time-consuming activities are to fix MTCP.

Certainly, MTCP is a developing project, not so mature even today. But here I want to point out that, MTCP, a user level checkpoint approach, must face with intrinsic difficulties.

For solutions at each level, they must rebuild the lower level status when restarting. However, the upper level usually knows little about the lower level, which make the rebuilding so hard.

For example, MTCP cannot change [vdso] memory address even today because it can’t tell the Linux kernel to map [vdso] into a specified area even today. That is why we can just post a temporal patch to fix the bug of MTCP.

Another example, when I have finished implementing the incremental checkpoint feature, I got confused by the big size of each incremental image. After several days’ exploration, I realized that when I make HBICT as a thread cooperating with MTCP, the cooperation produces a big state as large as 20 Mbytes! Of course, we can jump over that memory area when writing checkpoint image, and the image can still be restored. However, we don’t know exactly why this situation emerges. So we don’t have proper fixing plan even today.

In conclusion, if we want to continue our project in the future, we have better get ready to deal with many strange problems. And so, I recommend we employ someone that has broad knowledge of Linux kernel – the underlying level of MTCP.

Thank you, Nicolas. Thank you, James. I learned a lot from you this summer. It is my honor to work with you.

9. References

3. S. Kraemer, R. Leupers, D. Petras, T. Philipp. A checkpoint/restore framework on

