D5.4 – Resilience Primitives

WP5: Cross layer resilience and online analysis for non-functional parameters

Version: 1.0
Author(s): Kiril Dichev (QUB), Thomas Heller (FAU)
Date: 31/05/17
Due date: PM16  
Submission date: day/month/year  
Project start date: 01/10/2015  
Project duration: 36 months  
Deliverable lead organization: Queen's University Belfast  
Version: 1.0  
Status: Final  
Author(s): Kiril Dichev (QUB), Thomas Heller (FAU)  
Reviewer(s): Herbert Jordan (UIBK), Stéphane Monté (NUM)  

**Dissemination level**  
PU/PP/RE/CO  
PU

**Disclaimer**

This deliverable has been prepared by the responsible Work Package of the Project in accordance with the Consortium Agreement and the Grant Agreement Nr 671603. It solely reflects the opinion of the parties to such agreements on a collective basis in the context of the Project and to the extent foreseen in such agreements.
Acknowledgements

The work presented in this document has been conducted in the context of the EU Horizon 2020. AllScale is a 36-month project that started on October 1st, 2015 and is funded by the European Commission.

The partners in the project are UNIVERSITÄT INNSBRUCK (UBIK), FRIEDRICH- ALEXANDER-UNIVERSITÄT ERLANGEN NÜRNBERG (FAU), THE QUEEN'S UNIVERSITY OF BELFAST (QUB), KUNGLIGA TEKNISKA HÖGSKOLAN (KTH), NUMERICAL MECHANICS APPLICATIONS INTERNATIONAL SA (NUMEXA), IBM IRELAND LIMITED (IBM).

The content of this document is the result of extensive discussions within the AllScale Consortium as a whole.

More information
Public AllScale reports and other information pertaining to the project are available through the AllScale public Web site under http://www.allscale.eu.

Version History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Comments, Changes, Status</th>
<th>Authors, contributors, reviewers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>19/05/17</td>
<td>First draft</td>
<td>Kiril Dichev</td>
</tr>
<tr>
<td>0.2</td>
<td>24/05/17</td>
<td>Send for internal review</td>
<td>Kiril Dichev</td>
</tr>
<tr>
<td>0.3</td>
<td>30/05/17</td>
<td>Incorporate Herbert’s feedback</td>
<td>Kiril Dichev, Herbert Jordan</td>
</tr>
<tr>
<td>0.4</td>
<td>31/05/17</td>
<td>Incorporate Stéphane’s feedback</td>
<td>Kiril Dichev, Stéphane Monté</td>
</tr>
</tbody>
</table>
Table of Contents

Executive Summary ......................................................................................................................... 5
1 Overview of Task-Based Checkpoint-Restart Protocol ......................................................... 5
   1.1 Task Logging and Data Checkpointing ............................................................................... 5
2 Requirements on the AllScale Runtime System ....................................................................... 8
3 Simulator and Cost Model ........................................................................................................ 10
4 Future Work .............................................................................................................................. 12
5 Bibliography ............................................................................................................................. 12

Index of Figures

Figure 1: Task logging and data checkpointing per task ............................................................... 6
Figure 2: Example of TC = 2 checkpoint tasks T_1 and T_2. T_1 contains the 4 TL tasks T (1,1), T (1,2), T(1,3) and T(2,2). T_1 checkpoint includes the input to T(1,1) and T(1,3) (dashed line). T_2 contains the 4 TL tasks T(1,4), T(2,3), T(2,4), T(2,5). T_2 checkpoint includes the input to T(2,3), T(1,4) and T(2,5) (dashed line). ............... 6
Figure 3: Set of restarted tasks based on dependencies ............................................................... 7
Figure 4: Illustrating rollback to last global checkpoint (left) and dependency-aware rollback (right). Failed tasks are marked in red. Cancelled and recomputed tasks are marked in yellow. Checkpoint boundaries are marked in cyan. ............... 8
Figure 5: Variation in runtime for varying checkpointing levels (TC) for each implemented rollback ........................................................................................................................................ 11
Executive Summary

This deliverable, despite being numbered 5.4, deals with Task 5.3. The main objectives of the task are:

1. Identify requirements towards the AllScale Runtime System to implement resilience primitives.
2. Provide an implementation for resilience primitives in the AllScale Runtime System. These include support for generic task-level checkpoint-restart functionality.
3. Create a cost model for task-level checkpoint-restart functionality.

To address point 1, we have outlined a task-level checkpoint-restart protocol in (D5.5 - Implementation and Evaluation of Application Specific Resilience Techniques (a)), and we further improve the design in this deliverable, as well as define our requirements on the AllScale Runtime System. We have implemented the resilience protocol in a discrete-event simulator. However, we have not yet implemented these runtime requirements in the AllScale Runtime System (point 2). In this sense, we are on schedule regarding the identification of runtime requirements, but behind schedule regarding the implementation of these requirements within the AllScale runtime. In this document, we list the requirements that the resilience protocol poses on the AllScale Runtime System.

To address point 3, we have fully implemented a resilience simulator. The simulator is suitable for experimenting with various cost models for checkpointing. In particular, the well-established Young's formula (Young, 1974) for checkpoint/restart strategies is a suitable candidate for a cost model. We have performed a preliminary evaluation via the resilience simulator, which suggests the formula is suitable also for task-based runtimes such as ours.

We also note that T5.3 mentions the exploration of “alternative devices” for checkpointing. This exploration is not planned: one important reason is that the work on in-memory checkpointing (Kalé, 2012) has led extremely scalable HPC codes, which rely on checkpointing at peer nodes rather than alternative devices. This is also the path we follow.

1 Overview of Task-Based Checkpoint-Restart Protocol

In this section summarize the task-based checkpoint-restart protocol we have designed. It incrementally builds on the protocol outlined by (D5.5 - Implementation and Evaluation of Application Specific Resilience Techniques (a)).

1.1 Task Logging and Data Checkpointing

Our design checkpoints both task closure and application data. We have precisely defined the protocol to do this the following way:

\[
\text{closure} \leftarrow \text{get\_closure}(t) \\
\text{data} \leftarrow \text{get\_data}(t) \\
\text{if granularity}(t) \geq T_l \text{ then}
\]
D5.4 – Resilience Primitives

```
send_task_log(closure)
endif
if granularity(t) == Tc and t at boundary of Tc task then
    send_checkpoint(data)
endif
process(t)
```

Figure 1: Task logging and data checkpointing per task

The pseudocode highlighted in cyan implements the checkpointing policy. Note that it may be generated as well, and precedes the processing of each task \( t \).

A task closure contains all the information needed to schedule a task for execution, including:
- global task identifier
- task dependencies
- task input arguments

The data checkpoint is any application-specific data the application works on. On the example of a stencil application, this is the range of data within, e.g. a grid that is read and written by one of the application’s tasks.

\( T_L \) and \( T_C \) are the granularities for task logging, and for data checkpointing. An example of the local checkpointing strategy for 1D stencils is shown in Figure 2.

Figure 2: Example of \( T_C = 2 \) checkpoint tasks \( T_1 \) and \( T_2 \). \( T_1 \) contains the 4 \( T_L \) tasks \( T(1,1), T(1,2), T(1,3) \) and \( T(2,2) \). \( T_1 \) checkpoint includes the input to \( T(1,1) \) and \( T(1,3) \) (dashed line). \( T_2 \) contains the 4 \( T_L \) tasks \( T(1,4), T(2,3), T(2,4), T(2,5) \). \( T_2 \) checkpoint includes the input to \( T(2,3), T(1,4) \) and \( T(2,5) \) (dashed line).
In general, we assume $T_L \leq T_C$. That is, we log tasks at least as frequently as we checkpoint data. All sub-tasks (that is, tasks of smaller granularity than $T_L$) do not need to be logged/checkpointed, and can be recovered within their larger parent tasks.

$T_L$ needs to be of a coarse enough granularity, so that work stealing (within/between nodes) pays off. This granularity should be determined by the scheduler rather than the resilience protocol.

The `send_task_log` and `send_checkpoint` calls store task logs or data checkpoints at the main memory of their guard node. The guard-protectee scheme has already been detailed in (D5.5 - Implementation and Evaluation of Application Specific Resilience Techniques (a)). In essence, it is a scheme for in-memory checkpointing at a peer node, which has shown excellent scalability for large-scale applications, such as (Gamell, 2014).

### 2.2. Dependency-Aware Recovery

We design a recovery not previously published. The recovery uses an optimization we call **dependency-aware rollback**.

The algorithm for the dependency-aware rollback relies on the following:

1. There is a task set $L_1$ – the logged tasks that failed and have not been backed up at a failed node
2. There is a task set $L_2$ – the tasks building the last enclosing checkpoint

The minimum set of restarted tasks $L_3$ that will reproduce all failed tasks can be found as follows:

```plaintext
L_3 = {}
for t_1 in L_1 do
    for t_2 in L_2 do
        for t_3 in <set of all T_L-level tasks> do
            if t_1 depends on t_3 and t_3 depends on t_2 then
                L_3 = L_3 \cup t_1
            endif
        endfor
    endfor
endfor
return L_3
```

**Figure 3: Set of restarted tasks based on dependencies**

This routine establishes all the tasks between the last consistent enclosing checkpoint and the failed tasks, which are needed to recover the input to the failed tasks. These tasks are established based on the task dependencies, which are explicitly defined. There is a number of potential optimizations of this procedure, so that the three-time nested for-loop is not run. These could be achieved via application-specific versions of this routine (e.g. for stencils $L_3$ can be easily established based on the ranges contained in the arguments). However, such optimizations may not be provided within the scope of the project.

We illustrate the default rollback to the last globally consistent checkpoint, and the dependency-aware rollback in Figure 4. We use the developed simulator, and
our application is a 1D stencil. The illustrated settings contains 128 stencil array (x axis) with 128 time steps (y axis). That corresponds to a total of $128^2$ T\textsubscript{L}-level tasks. The tasks marked in red are failed tasks. Marked in yellow are all tasks that are cancelled and recomputed to restore the failed tasks. The cyan lines are the outlines of each T\textsubscript{C}-level checkpoint. A T\textsubscript{L} task takes 7.1 seconds to process, and only 1.3 milliseconds to checkpoint. These times are translated from real benchmarks of the Pochoir stencil compiler (Tang, 2011). The Mean Time Between Failure (MTBF) we set is 900 seconds. For the used random seed, this generates 4 node failures – at 100, 1231, 1713, and 1764 seconds of a run of just over 2000 seconds.

As visualised, the area in yellow represents the cancelled tasks, and less tasks are cancelled for dependency-aware rollback under similar fault scenarios. This always leads to reduced use of compute resources. In our experiments, this also leads to reduced overall execution time.

2 Requirements on the AllScale Runtime System

Based on the developed resilience protocol, following requirements on the AllScale Runtime System exist:

1. The runtime must provide conversion from the closure and application data associated to a task into (serializable) bytes streams:
   
   closure\_buf = get\_closure(t)
   
   data\_buf = get\_data(t)

   We require the runtime to return closure\_buf and data\_buf as serializable byte streams, even if the contents of these are application-specific. Note that the compiler inserts serialization code for the runtime for simple data types. However, the application developer may need to support the serialization for types such as pointers and references.
2. The runtime needs to implement routines for sending a (serializable) byte stream to the main memory of a remote node:
   
   `send_task_log(closure_buf)`
   
   `send_checkpoint(data_buf)`
   
   These calls should be blocking on the sender node, but should not block the execution of the remote application. The target node of task logs and data checkpoints, that is the guard node of each node, is provided to the runtime by the resilience manager, and does not need to be passed as an argument.

3. The runtime must implement receive routines for (serializable) byte streams from the main memory of a remote node:
   
   `closure_buf = recv_task_log()`
   
   `data_buf = recv_checkpoint()`
   
   Again, the sender is transparently managed by the resilience manager.

4. The runtime must provide conversion routines from byte streams into closures and application data. The closure and data contain application-specific information. The closure must be sufficient to reschedule a task:
   
   `closure = get_closure(closure_buf)`
   
   `data = get_data(data_buf)`

5. The runtime should provide a boolean function
   
   `depends_on(t1,t2)`
   
   which establishes if there is a (possibly transitive) dependency between two tasks based on the task closures t1 and t2.

6. The runtime must provide a function:
   
   `cancel(closure)`
   
   which cancels a task via its closure, and all sub-tasks it may have spawned. The call is blocking until cancellation is complete.

7. The runtime must provide a function:
   
   `reschedule(closure)`
   
   which can reschedule a task via its closure using any scheduling policy the runtime chooses. It is irrelevant to the resilience manager where the task is rescheduled. The `reschedule` function is specifically for resilience purposes.

8. `reschedule(closure)` is called for the task set L₃ from Figure 3. All rescheduled tasks must modify short-lived replicas of the global data, and not the global data itself. This is a requirement on the runtime by the resilience manager in order to prevent the global data from becoming inconsistent. This is a strict requirement only for dependency-aware rollback. The short-lived replicas of global data may have following life cycle:
   
   a. The entire L₃ task set is rescheduled by the guard node as a set of new, independent tasks, which are separate from the application tasks:
      
      i. The L₃ task dependencies need to duplicate the task dependencies leading up to the failed tasks. These are a sub-set of all application dependencies. No other dependencies outside
of L₃ tasks need to be recreated, since these tasks have not failed.  
ii. L₃ tasks work on data replicas. The extent of these data replicas 
is either a sub-set of the global data (not necessarily 
continuous), or in the worst a replica of the entire global data. 
At no point may the data L₃ tasks operate on the global data of 
the application. The short-lived data replicas live globally, but 
their lifetime starts with reading the enclosing checkpoint and 
ends when all failed tasks are recovered. 
b. Upon restarting all failed tasks, the replica tasks L₃ have completed. 
The entire replica of global data needs to be terminated. The 
termination is the responsibility of the runtime.

3 Simulator and Cost Model
Any cost model of checkpoint-restart techniques should first consider the Young 
(Young, 1974) and Daly (Daly, 2006) formulation, which provides the optimal 
checkpointing period. This formulation has been both theoretically and 
practically verified to provide the optimal checkpointing interval for MPI 
applications. In its simplest form, Young formula says that the optimal 
checkpoint interval \( T_{opt} \) is given by

\[
T_{opt} = \sqrt{2 \ast \alpha \ast M}
\]

Equation 1: Young’s formula

where \( \alpha \) is the checkpoint duration, and \( M \) is the Mean Time Between Failure 
(MTBF). We maintain that we don’t need to look for other cost models before 
verifying whether this formulation is a sufficient cost model. 
There are good reasons for verifying the applicability of Young’s formula to the 
scenario imposed by the AllScale Runtime System:

- The AllScale Runtime System may not have a constant checkpoint 
duration (which Young’s formula assumes)
- The dependency-aware rollback differs from the rollback to the last 
checkpoint (which Young’s formula assumes)

In D5.5(a) we implemented a first version of a resilience simulator (Dichev, 
Resilience Simulator, 2017), which we improved throughout the project.

The simulator is the main tool to develop and verify cost models. On the example 
of a 1D stencil, we illustrate the variation in execution time for varying 
checkpoint levels \( (T_C) \), and a default recovery and dependency-aware recovery in 
Figure 5. The different checkpoint levels translate in different frequency of 
checkpointing. The setup uses a processing time of 5 seconds per \( T_L \)-level task, 
and 5 second duration per \( T_L \)-level checkpoint. The grid is 128 stencil elements 
times 128 time steps. 64 workers are simulated.
We make two observations:

- Dependency-aware rollback reduces overall execution time compared 
to the rollback to a checkpoint.
- In both cases, at checkpoint level \( T_C = 4 \) the runtime reaches a 
minimum.
The reduced overall execution time for dependency-aware rollback can be explained by less cancelled and rescheduled tasks. On the other hand, the minimum at $T_C = 4$ suggests that a cost model, such as Young's formula (or variations thereof), can analytically provide this minimum. We outline here how simulator and the formula can be compared:

- The simulator provides minimum at $T_C = 4$, which corresponds to $8^2 T_L$-level tasks.
- We can measure (via the simulator) how long it takes for 64 workers to process $8^2 T_L$-level tasks. The simulator provides a duration of 107 seconds to complete. This is the optimal checkpoint interval provided by the simulator.

We can now use Equation 1 to verify if the optimal checkpoint interval is close enough to what the simulator provides. We get $T_{opt} = \sqrt{2 \times 900 \times 5} = 94$ seconds, with $\alpha = 5$ and $M = 900$.

In summary, the simulator provides 107 second checkpoint interval, and Equation 1 provides ideal interval of 94 seconds. Note that the second closest checkpoint granularity is much further away (with $T_C = 3$, the checkpoint interval is 50 seconds). The simulator and formula results are very close and indicate that even if corrections are needed for the formula, they would be minor.

![Figure 5: Variation in runtime for varying checkpointing levels ($T_c$) for each implemented rollback.](image-url)
We have also experimented with much smaller checkpointing duration for a 1D stencil based on realistic cluster settings. The simulator, in agreement with Young’s formula, suggests a more frequent checkpointing.

4 Future Work
T5.4 specifies that both defining and implementing resilience primitives within the AllScale Runtime System will be provided by M20. As outlined in the Executive Summary, our efforts by M20 focused in designing and verifying the resilience protocol, and in defining the resilience primitives. Our future work is in implementing the resilience primitives described in this document within the AllScale Runtime System. In particular, defining data replicas during recovery, as well as an efficient implementation of the algorithm listed in Figure 3, are very challenging aspects. In addition, the serialization/deserialization of task closures and application data, will also require significant implementation efforts.

5 Bibliography