Data Fault Tolerance using ROSE

Kamal Sharma
Lawrence Livermore National Lab / Rice University

Introduction

Exascale computing provides challenges for data fault tolerance. As number of cores increase for Exascale, so does the number of components associated with each core. Components comprises of memory controllers, DRAM chips, network controllers and caches. As components grow, they may produce faults. Faults in memories can be classified as soft errors and hard errors [1].

Soft errors occur due to particle strikes, random noise and unstable device behavior leading to single or multi-bit errors. Hard errors occur to device malfunction either during manufacturing or usage over time. During manufacturing, hard errors are easier to detect by means of rigorously testing the hardware. Overall, we can only expect soft and hard errors to increase for Exascale machines. In a DARPA study for Exascale [2], it is pointed that one uncorrectable single event due to memory might occur as frequently as 6 hours. Data fault tolerance mitigates this problem by providing fault resiliency in the application.

Data Fault tolerance may be implemented in different ways such as checkpointing/rollback and data encoding algorithms. Checkpointing method saves the application state at a given instance in time so that when an error occurs, the application state can be reverted back in time. However, checkpointing incurs high overhead in memory as well as application stall time during checkpointing. For example, a single checkpoint on BlueGene/L takes 12 minutes of application time [2]. In contrast, data encoding methods mitigate the faults at a finer granularity level. These approaches typically apply an encoding algorithm to add more bits with data to detect and correct errors. In this work, we focus on automatic data encoding techniques using ROSE for array data accesses. We present the necessary steps for invoking data encoding translator and runtime checks performed on array elements.

Project Source Organization

This section presents the source repository structure for ROSE project. The entire source can be found in projects/DataFaultTolerance.

DataFaultTolerance
+- includes
    This folder contains the necessary C header files for fault tolerance runtime library calls.
+- src

This folder includes all the source files for data encoding ROSE translator and generating the fault tolerance library, which can be linked to an application.

ROSE Translator – faultToleranceArrayPass.C, arrayBase.C , AstFromString.C, pragmaHandling.C (Used for pragma processing)

Fault Tolerance Library - faultToleranceArrayLib.C, faultToleranceArrayLibUtility.C

+- test

This folder contains the test cases for translator and fault tolerance library.

+- faultCheck

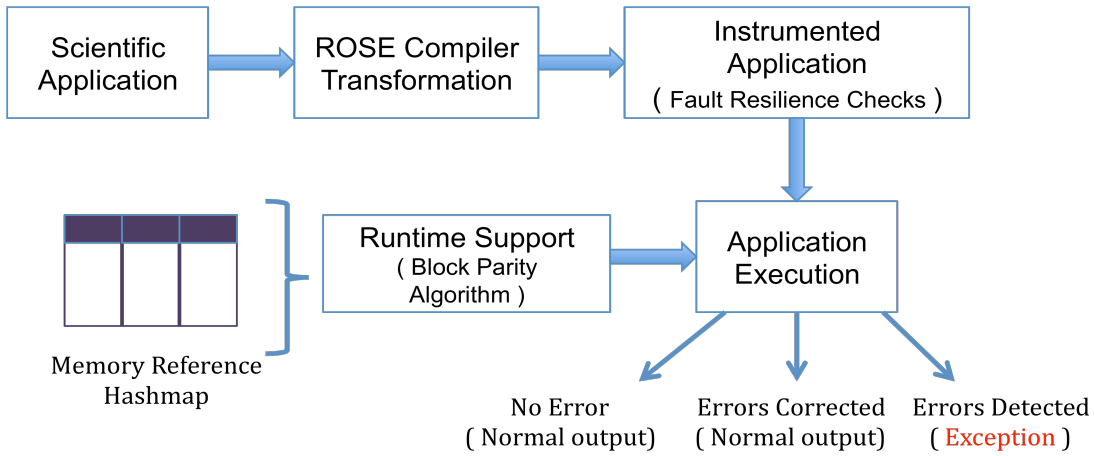
Contains two examples, rose_testSample.C and rose_ErrtestSample.C for fault tolerance checking using the runtime library.

+- transformation

Contains several test cases for single array and multi-dimensional arrays to test the correctness for ROSE Transformation.

Overall Approach

The overall flow for data fault tolerance is shown in the figure below.



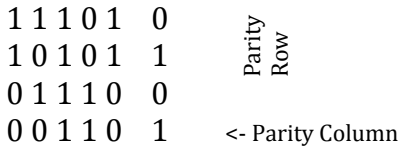
A scientific application is used an input for source level instrumentation. We employ a pragma-based approach for instrumentation. In this case, the user specifies specific loops for fault tolerance by using mem_fault_tolerance pragma along with other options like chunksize and array information. ROSE Transformation checks for these pragmas present in the source code to insert appropriate data fault tolerance methods. These fault tolerance methods are inserted based on array

accesses. On update to an array element, we need to correspondingly update its value in a hash table, where encoded data is kept for fault tolerance check.

Once instrumentation is performed, we can now run the application using the linked fault tolerance library. The fault tolerance library is essential to store the encoded data in a hash table and appropriately check the data value. We use a block parity algorithm for bit error detection and correction. If all array accesses correctly verify their value in the hash table, an application produces normal output. However, if there is an uncorrectable error detected, the application produces an exception.

Block Parity Algorithm

In this section, we explain the block parity algorithm. The block parity algorithm is a standard parity algorithm that introduces a block row and column for a group of data. We apply the block parity algorithm for integer data type.
Figure below shows the block parity calculated for a block of three elements.



The parity row and column is calculated using EXOR operation with rows and columns. Block parity helps in detecting burst error codes. If the parity row or column does not match for given block of data, we can easily detect the error. However, if there are multiple bit flips in rows as well as columns then block parity might not be able to help. More details about block parity algorithm can be found in [3].

ROSE Transformation

As explained in earlier, the ROSE transformation instruments the source program based on pragma specified. Here, we demonstrate this transformation using an example.

Example 1 - Original Source Program

```
int main()
{

      int A[100], i;

      #pragma mem_fault_tolerance (A:<0:100>) (Chunksize=8)
      for(i=0; i<100; i++)
            A[i] = 5;
}
```

This example shows a simple update of Array A for 100 elements. In order to enable fault check, a user specifies mem_fault_tolerance pragma. The pragma in this case uses two arguments, array A description and chunksize. The array description consists of array name tagged with the dimension information. In the current implementation, we have assumed that the dimension information is added in the pragma as C provides different variations to access an array. Moreover, the loop itself might access limited data elements as compared to full arrays.

Chunksize is the second parameter present in the pragma. This is an optional parameter. Default value for this parameter is 1. The chunksize parameter basically groups the array elements together to insert encoded data into the hash table. The user should select this parameter based on the expected faults in a given scenario. Larger values would require less space in hash table but would be able to correct fewer errors, whereas smaller chunksize values causes the hash table to require larger space but in turn would be able to catch more faults.

Example1- ROSE Transformed Output

```
#include "faultToleranceArrayLib.h"

int main()
{
  int A[100UL];
  int i;

#pragma mem_fault_tolerance ( A : < 0 : 100 > ) ( Chunksize = 8 )
  for (i = 0; i < 100; i++) {
    int _memTemp0 = A[i];
    A[i] = 5;
    updateElem("A",A[i],i,_memTemp0,8);
  }
  clearHashTable();
  return 0;
}
```

The above code shows the transformed version for Example 1. We see that source code has instrumented calls. A new variable `_memTemp0` is introduced to retain the old value of array element before a write to array element. This is necessary for block parity algorithm. An `updateElem` call is introduced after the write operation. This enables the hash table to have consistent encoded data. `clearHashTable` clears all the data values since there may be intermediate accesses to array elements, which might cause inconsistent hash entries. Moreover, the chunksize parameter might change across the loops.

This was a simple example of single array write operation. Thus, we do not need additional checks of reading all the array elements prior to entering the loop and validating the updates after the loop. In the later example, we show more instrumentation calls.

Example 1 - Original Source Program

```
int main()
{

      int A[100], B[100], i, temp;

      #pragma mem_fault_tolerance (A:<0:100>) (B:<0:100>)
(Chunksize=10)
      for(i=0; i<100; i++)
      {
            A[i] = 5;
            B[i] = 0;
      }

      #pragma mem_fault_tolerance (A:<0:100>) (B:<0:100>)
(Chunksize=10)
      for(i=0; i<100; i++)
      {
            B[i] = B[i] + A[i];
      }
}
```

Example2- ROSE Transformed Output

```
#include "faultToleranceArrayLib.h"

int main()
{
  int A[100UL];
  int B[100UL];
  int i;
  int temp;

#pragma mem_fault_tolerance ( A : < 0 : 100 > ) ( B : < 0 : 100 >
) ( Chunksize = 10 )
  for (i = 0; i < 100; i++) {
    int _memTemp0 = A[i];
    A[i] = 5;
    updateElem("A",A[i],i,_memTemp0,10);
    int _memTemp1 = B[i];
    B[i] = 0;
    updateElem("B",B[i],i,_memTemp1,10);
  }
  clearHashTable();

#pragma mem_fault_tolerance ( A : < 0 : 100 > ) ( B : < 0 : 100 >
) ( Chunksize = 10 )
  updateArray("A",A + 0,0,100,10);
  updateArray("B",B + 0,0,100,10);
  for (i = 0; i < 100; i++) {
    int _memTemp2 = B[i];
    B[i] = (B[i] + A[i]);
    updateElem("B",B[i],i,_memTemp2,10);
  }
  validateArray("A",A + 0,0,100,10);
  validateArray("B",B + 0,0,100,10);
  clearHashTable();
  return 0;
}
```

Fault Tolerance Library Check

In the earlier section, the ROSE transformed output was demonstrated. However, we would like to see the error detection capability of the block parity algorithm. In this section, an example with artificially introduced error is shown for fault tolerance check.

Example1 – Fault Tolerance Check

```
#include "faultToleranceArrayLib.h"

int main()
{
  int i;
  int A[100UL];
  int B[100UL][100UL];
  int C[100UL];

for (i = 0; i < 10; i++) {
    A[i] = 0;
  }

#pragma mem_fault_tolerance ( A : < 0 : 100 > ) ( B : < 0 : 100 >
< 0 : 100 > ) ( C : < 0 : 100 > ) ( Chunksize = 8 )
  updateArray("A",A + 0,0,100,8);
  updateArray("B",B[0] + 0,0,100,0,100,8);
  for (i = 0; i < 10; i++) {
    int _memTemp0 = B[i + 1][i + 5];
    B[i + 1][i + 5] = A[i];
    updateElem("B",B[i + 1][i + 5],i + 1,i + 5,_memTemp0,8);
    int _memTemp1 = C[i];
    C[i] = B[i][i + 2];
    updateElem("C",C[i],i,_memTemp1,8);
    B[2][7] = 64; // Error location
  }
  validateArray("B",B[0] + 0,0,100,0,100,8);
  validateArray("A",A + 0,0,100,8);
  return 0;
}
```

Output

```
Parity row or column for array entry B_2_0_7 doesn't match
```

Conclusion

Overall a complete ROSE Transformation framework for data fault tolerance is presented in this work. This includes a ROSE Transformation along with a runtime check library using block parity algorithm.

References:
[1] “Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding”, Jangwoo Kim, Nikos Hardavellas, Ken Mai, Babak Falsafi, James Hoe, MICRO Conference 2007.
[2] ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems, DARPA Report 2008.
[3] http://www.mathcs.emory.edu/~cheung/Courses/455/Syllabus/2-physical/errors.html