Optimal Placement of Retry-Based Fault Recovery Annotations in HPC Applications *

Ignacio Laguna, Martin Schulz, Jeff Keasler, David Richards, Jim Belak
Lawrence Livermore National Laboratory
Main contact: ilaguna@llnl.gov

ABSTRACT

As larger HPC systems are built, fault recovery becomes a fundamental capability. Traditional fault recovery approaches, such as checkpointing, may not be sufficient for future exascale systems. Retry-based recovery techniques have been proposed as an alternative. These techniques simply re-execute a code region when a fault occurs and require code annotations. However, no previous work has investigated the optimal placement of these annotations in a program. Via fault injection, we evaluate how to place optimally retry annotations in a hydrodynamics mini application. We found that, contrary to our expectations, a simple scheme of protecting the main function works well for low fault rates: slowdown is up to 1.25 for a 3 faults/hour rate. We also found that the optimal recovery method is rolling a few iterations back in the application's main loop.

1. INTRODUCTION

As we build larger and more complex high-performance computing (HPC) systems, fault recovery becomes a fundamental capability to make use of theses systems. The traditional recovery approach in HPC systems is the checkpoint/restart (C/R) approach. However, checkpoints are often aggressive in saving more state than required by the application and often have high performance overheads [2]. Although hardware support may help in maintaining reasonable performance for C/R, it incurs energy usage irrespective of the occurrence of failures. It is believed that for high fault rates—as expected in exascale machines—we may not rely only on the C/R approach since applications may spend more time in checkpointing than in computation [1].

Idempotent code has been proposed as an alternate solution for C/R [3]—a code region is said to be *idempotent* if it has the property that re-execution is free of side effects. In the event of an execution fault, idempotent code is used to correct the state of the program by simply re-executing it. A difficulty is that most programs are not written as pieces of idempotent code, and recovery techniques that rely on them have to either identify idempotent code regions or to create them if they do not exist [3]. These transformations are typically done at the level of basic blocks. Creating idempotent code regions usually requires changes that may alter the original semantics of the program, e.g., adding temporal variables and control flow statements. These changes make debugging difficult since the original source code may not map directly the translated machine code.

Retry-based recovery models that do not rearrange protected code have been proposed [2]. A code region is protected by, first, in-memory checkpointing state at the beginning of the region and then, when a fault occurs, restoring saved state and re-executing the region. Users can manually annotate the program to protect specific regions or a compiler can annotate automatically regions (e.g., functions or loops). Since this model does not require finding idempotent regions, `RETRY` annotations can be placed arbitrarily in the program. For example, users could place a `RETRY` annotation within the main function or to place it in every function (Figure 1(a) shows the annotations format). The decision on where to place annotations is critical or large overheads can occur. To the best of our knowledge, no previous research has investigated how to place optimally these annotations.

We evaluate retry-based recovery models and find optimal methods to place annotations. The annotations overhead depends on the protected code-region size, the amount of state that has to be saved, and the fault rate. Through fault injection, we evaluate the overhead of placing `RETRY` annotations in a mini hydrodynamics application, LULESH [4]. We found that, the simple approach of protecting only the main function (i.e., placing only one `RETRY` annotation between the beginning and end of the main function), works well for a small fault rate (i.e., less than three faults/hour). We also found that the optimal recovery method is to roll a few iterations back in the application's main loop. Our fault-injection technique is part of the GREMLIN infrastructure [5], which allows emulating the behavior of future machines by injecting perturbations in current machines.

We also evaluate a model called RAJA that translates original code to idempotent code at the source-code level, instead of at the machine-instruction level. Using fault injection through our GREMLIN infrastructure, we show that this model has minimal overhead (a maximum slowdown of 1.1) as it is expected from previous studies [3].

2. APPROACH

Mini Application. We use LULESH [4], a shock hydrodynamics mini application for our study. LULESH has been ported to a number of programming models. In our study we used the serial C++ version 1.1.

Fault Model. We focus on future exascale machines hardware faults. We assume that techniques like error-correcting codes (ECC) will effectively protect memory and register

*This work was performed partly under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DEAC52-07NA27344 (LLNL-ABS-641414).

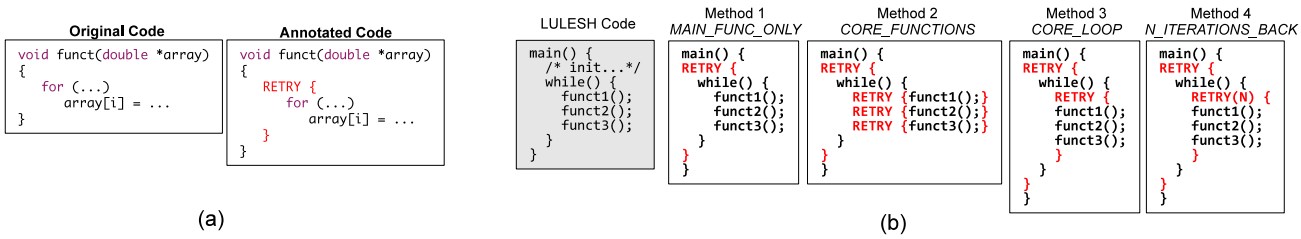

Figure 1: (a) Retry-based recovery method; (b) Multiple methods to annotate LULESH source code.

state, leaving instruction execution as the only possible source of faults—a common assumption in prior work [3]. We also assume that error detection is done in hardware and that notifications for recoverable errors are observed in software (i.e., operating system and application) in a synchronous manner (e.g., via synchronous exceptions or traps).

In-Memory Checkpoint Model. We implement a prototype of a retry-based model using C++ try/catch macros. Code is annotated manually with `RETRY` macros to intercept exceptions which emulate hardware error notifications. An annotation includes initializing the try/catch code and saving a copy of the state that is required to retry the code safely. This includes finding write-after-read (WAR) dependencies in the code region that we want to protect, which can be done via a front-end compiler pass—a code region with only read-after-write (RAW) dependencies implies it is idempotent thus it does not need to save initial state. Figure 1(b) shows the different methods of placing annotations that we evaluated. Method 1 simply protects the application's main function—if a fault occurs, the application starts again from the beginning. Method 2 protects the main functions while method 3 protects the main loop (it rolls one iteration back after a fault). Method 4 is a generalization of method 3—it can roll back N iterations instead of only one.

RAJA Model. The RAJA model creates idempotent code at the level of the application source code. This is done by breaking WAR dependencies using temporal variables. Since there are no modifications at machine-code level, the resulting code is suitable for debugging purposes at the expense of manually re-writing the code. However, this model requires more modifications to the code than methods 1–4.

3. EVALUATION

Fault Injections. We emulate hardware faults by triggering exceptions via the GREMLIN infrastructure. Time between faults follows an exponential distribution. We vary fault rate from 1 to 25 faults/hour. LULESH execution time is around 2 minutes, so for low fault rates, it is possible that the application does not receive any fault. Thus, for each fault rate, we run 25 experiments and calculate the slowdown average, where slowdown is application runtime with faults and recovery divided by runtime without faults and without recovery.

Results. Figure 2 shows the results. Method 1 (MAIN_FUNC_ONLY) works surprisingly well for low fault rates (less than 3 faults/hour). This is because no state has to be saved in this method. Methods 2 and 3 (i.e., CORE_FUNCTIONS and CORE_LOOP) perform poorly, mainly due to the high overhead incurred in in-memory checkpointing state. Method 4 is the optimal method for this application, as it incurs the minimal slowdown (We exclude RAJA from this comparison as it is a different model.). We found that rolling back around 25 iterations is the best approach to recover in this application. RAJA has the lowest overhead overall—it does not require saving state and recovery only requires re-executing short code regions.

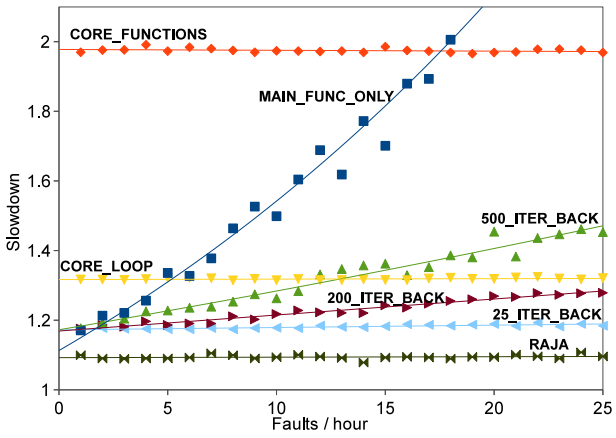

Figure 2: Slowdown of all recovery methods.

4. CONCLUSION AND FUTURE WORK

We evaluate hardware-fault recovery techniques based on annotations and source-code transformations to protect code regions. For a hydrodynamics mini application, we found that the optimal recovery method is to roll back a few iterations in the main loop. In future work, we plan to find if our conclusions are also valid in other HPC applications (i.e., that the applications can be protected mainly by protecting a number of iterations of the main loop).

5. REFERENCES

[1] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *Int. J. High Perform. Comput. Appl.*, 23(4), Nov. 2009.

[2] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. ISCA '10, 2010.

[3] M. A. de Kruijf, K. Sankaralingam, and S. Jha. Static analysis and compiler design for idempotent processing. PLDI '12, 2012.

[4] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still. Exploring traditional and emerging parallel programming models using a proxy application. IPDPS '13, 2013.

[5] LLNL. GREMLIN infrastructure. `https://scalability.llnl.gov/gremlins/`.