Pragma-Controlled Source-to-Source Code Transformations for Robust Application Execution

Pedro C. Diniz[1(✉)], Chunhua Liao[2], Daniel J. Quinlan[2], and Robert F. Lucas[1]

[1] USC Information Sciences Institute,
4676 Admiralty Way, Suite 1001, Marina del Rey, CA 90292, USA
{pedro,rflucas}@isi.edu
[2] Lawrence Livermore National Laboratory,
7000 East Avenue, Livermore, CA 94550, USA
{liao6,dquinlan}@llnl.gov

Abstract. The most widely used resiliency approach today, based on Checkpoint and Restart (C/R) recovery, is not expected to remain viable in the presence of the accelerated fault and error rates in future Exascale-class systems. In this paper, we introduce a series of pragma directives and the corresponding source-to-source transformations that are designed to convey to a compiler, and ultimately a fault-aware run-time system, key information about the tolerance to memory errors in selected sections of an application. These directives, implemented in the ROSE compiler infrastructure, convey information about storage mapping and error tolerance but also amelioration and recovery using externally provided functions and multi-threading. We present preliminary results of the use of a subset of these directives for a simple implementation of the conjugate-gradient numerical solver in the presence of uncorrected memory errors, showing that it is possible to implement simple recovery strategies with very low programmer effort and execution time overhead.

1 Introduction

The resilience of High Performance Computing (HPC) applications in the presence of faults and errors on future extreme scale supercomputing systems is a growing concern. With process technology scaling, future exascale-class systems will be constructed from transistor devices which are less reliable than those used today. Furthermore, the recent trend of aggressive scaling of processor cores and memory chips in order to drive floating-point performance suggests that future exascale class systems will require exponential growth in compute and memory resources [1,13]. However, with increase in the number of system components, the overall reliability of the system will decrease proportionally. The projections on fault rates based on current HPC systems and technology roadmaps predict that exascale class systems will experience several errors per day. This will impact long running scientific applications which will terminate abnormally, or worse, may complete with incorrect results [4].

F. Desprez et al. (Eds.): Euro-Par 2016 Workshops, LNCS 10104, pp. 660–670, 2017.
DOI: 10.1007/978-3-319-58943-5_53

The de-facto approach used today to provide resilient operation is based on Checkpoint and Restart (C/R) which periodically commits the application state to persistent storage. Recovery is initiated only upon failure of a process and entails terminating all the remaining processes and restarting the application from the latest stable global checkpoint. If unchecked by aggressively reducing its frequency and/or the volume of saved data, this approach is inviable. As the applications scale to leverage the capabilities of these large scale machines, the amount of state to be gathered will grow considerably, resulting in proportional increases in the intervals required to create and commit checkpoints as well as recover them from storage. Given that the projected Mean Time to Failure (MTTF) of the exascale systems will be of the order of a few minutes [1], C/R will no longer be effective in scenarios where the C/R interval is greater than the system MTTF.

Still, various HPC applications offer rich possibilities to algorithmically detect and correct the errors in their program state. Algorithm-based fault tolerance (ABFT) [2,9] techniques for linear algebra kernels enable the identification of the error location and correction of bit flip errors using checksum error encoding in the data structures and adapting the algorithms to operate on the encoded data. Similarly, iterative numerical algorithms such as the Adaptive Multi-Grid Solver [5], can tolerate errors at the expense of longer convergence rates or iterations. Even more extreme, algorithms that rely on *random* events, as is the case of algorithm that leverage Monte-Carlo simulation techniques can (in specific contexts) tolerate memory errors provided they do not lead to catastrophic program behavior. Yet, in the face of the wealth of such application acceptability or tolerance characteristics, such algorithmic features are not exposed to the programming environment due to the lack of convenient interfaces.

In this paper, we extend our previous approach that is based on programming model extensions that incorporates simple language-level support that is tightly coupled with the compiler and runtime system to adaptively and dynamically apply redundancy [11,12]. The approach allows users to specify various detections conditions that strongly suggest silent-data corruption in addition to the traditional error detection through abnormal program execution. The programming model extension described here is based on `#pragma` directives which are then translated as source-to-source code transformations to support application level detection and recovery strategies through retry and multi-threading checking and correction semantics. These detection and recovery mechanisms can be coupled with an introspection runtime system enabling the use of redundant multithreading when too many faults are observed. We have implemented these directives and the corresponding code transformations in the ROSE Compiler Infrastructure [16] and use them to demonstrate their suitability to an illustrative scientific kernel – the conjugate gradient iterative solver. The results, albeit very preliminary, do reveal that with very little programming effort, this pragma-based code transformations approach substantially increases the ability of selected section of codes to survive uncorrected memory errors.

2 Pragma-Based Code Transformation Directives

We next present the various pragma directives and illustrate their use via source-to-source transformations and examples. We begin with the simplest forms where fault-tolerance is indicated to very sophisticated source code transformations where detection and recovery with redundant code execution is used.

2.1 Hardware Error Detection and Correction

In this work we assume that some, but not all the memory faults are corrected via hardware mechanisms such as Error-Correcting-Codes (ECC) and chipKill [14]. As such, the only faults that are signaled via hardware that trigger the execution of the amelioration actions defined by the directives described here, include detected but uncorrected memory errors.[1]

2.2 Tolerant Storage Declaration

This first *tolerant* directive simply indicates that a specific data declaration (to follow the directive) can tolerate a specific maximum number of (uncorrected) errors. Alternatively, when present, the directive also indicates that the corresponding data structure should be placed in a specific data storage as indicated by a secondary integer identifier.[2] The directive has the syntax shown below where `exp1` and (the optional) `exp2` denote compile-time integer values.

```
#pragma failsafe tolerant ( exp1 : exp2 )
```

A simple example of the use of this directive is shown below where the array `A` is to be allocated preferentially to the storage labelled with id zero and can only tolerate 1 uncorrected error.

```
#pragma failsafe tolerant (1: 0)
int A[M][N];
```

In the absence of both expression `exp1` and `exp2` fields, the run-time system assumes that any number of errors are to be tolerated for this specific variable.

This pragma is also applicable to global variables or heap-allocated variables, although the later needs to be explicitly controlled by the use of a tolerant variant

[1] As a minor point, we further assume that upon restart (via state restore) data is flush out of cache storages so that erroneous values are not restore as part of the application's state.

[2] We envision a memory system where distinct regions of the storage space have distinct resilience characteristics each of which is identified by a unique numerical or symbolic identifier.

of malloc[3]. The variable to be associated with this behavior can either be a scalar or a statically allocated arrays. In case of a pointer variable, it is the pointer that needs to be labelled as tolerant but not the heap-allocated storage it points to. To that effect we also provide a variant of the `malloc` function labelled as `tolerant_malloc(size, N, K)`. Upon parsing and translation the compiler will produce a simple text file, with the scope of the variables labelled as tolerant and the corresponding statically declared name including the numeric values for `exp1` and `exp2`.

This tolerant data is then parsed by the run-time system and can be incorporated as part of an introspection system. As variables that are deemed less tolerant reach their limit of tolerated errors, they can be migrated to increasingly more robust regions of the address space thus allowing a run-time system to dynamically manage the underlying state of the machine while meeting (or at least attempting to meet) the tolerance requirements of each data structure.

2.3 Sentinel Values for Silent Data Corruption Detection

These constructs, akin to the `#assert` specify a user-defined predicates that must hold at specific execution points of the application. Using these pragmas the user can attempt to correct silent data or uncorrected errors in specific variables and thus proceed with the computation. Still, and even in the event of user amelioration, error variables record the error events and interface with a resilience introspection engine for subsequent application adaptivity.

The syntax of this pragma is shown below and it is the programmers responsibility to ensure that the evaluation scope of the arguments of the handler functions and assertion predicates are appropriately scoped.

```
#pragma failsafe assert ( predicate ) error ( function handler )
```

A simple example of the use of this pragma is shown below:

```
#pragma failsafe assert ( a > 0 ) error ( MyFunction(&b))
```

where it is assumed that `MyFunction` is an integer-returning function and where a non-zero value will indicate the inability to correct an erroneous condition and a zero valued return success in correcting such situation. The translation of the above directive in terms of source C code is as shown below.

[3] Automatically, converting the heap-allocated use of a malloc into a tolerant-malloc is rather tricky to do statically as in the general case a compiler would have to track the use of address as function argument and allocation across procedure boundaries to understand when the address of a pointer could have been declared in another scope as tolerant. As a results we restrict the use of this pragma to the storage that is statically allocated either at the file or at the global scope levels.

```
if(predicate(...) == 0){
  if(function_handler (...) != 0){
    failsafe_error++;
    FAILSAFE_REPORT_ERROR(0,failsafe_error);
    failsafe_error_flag = 0;
  } else {
    FAILSAFE_REPORT_CORRECTION(0,failsafe_error);
  }
}
```

In the absence of the `error` clause, and should the predicate evaluation not hold at runtime, the generated code will terminate the applications execution via the `exit` function as illustrated in the sample code below.

```
if(predicate(...) == 0){
  FAILSAFE_REPORT_ERROR_EXIT(0);
}
```

2.4 User-Controlled State Saving and Restoring with Retry

In order to provide users with the capability to control the saving and restoring of program state, we have included a save/restore directive. The directive thus include which program variables constitute relevant program state that needs to be saved and restored and for how many retries the execution of the subsequence control-flow program blocks should be attempted.

This directive can be combined with the assert pragmas described above to detect erroneous execution conditions resulted from silent data corruption.

```
#pragma failsafe save_restore ( var_list ) retry ( exp )
{/* code block */ }
```

This directive is translated into code that saves the state of the set of variables listed in the `var_list` into auxiliary variable via a memory copy construct[4] Upon detection of an uncorrected error in the code block, the control is transferred to the beginning of the block with the state of the saved variable reinstated. The snippet of code below depicts the structure of the generated code as the result of the translation of this directive for a retry value of 2.

```
int fs_num_tries;
volatile int fs_num_errors;
fs_num_tries  = 0;
fs_num_errors = 0;
<code for saving data objects>
do {
  if (fs_num_tries != 0){
```

[4] In the current implementation only supports scalar and statically allocated array variables with known compile-time bounds. The support of dynamically allocated arrays with multiple pointer levels can, however, pose serious implementation challenges in terms of correctness and performance.

```
    <code for restore data objects>
  }
  fs_num_errors = 0;
  <original code block here>
  fs_num_tries++;
}  while ((fs_num_errors != 0) && (fs_num_tries < 2));
if(fs_num_errors != 0){
  FAIL_SAFE_EXCEPTION();
}
```

2.5 Redundancy-Based Fault Detection and Recovery

In addition to the pragmas described above, we have also implemented two simple redundancy-based detection and recovery pragmas, namely, using dual and triple computing redundancy that can in some context detect and correct, respectively, errors in the computation by direct comparison of the values in a selected lists of variables. As with the previous pragma directive, a maximum number of retries is attempted before an abnormal execution is reported.

```
#pragma failsafe dual_redundancy save_restore ( var_list1 )
compare ( var_list2 ) retry ( exp )
 { /* code block */ }
```

In addition to the aspects of computation redundancy this directive extends the notion of redundancy by including dual and triple threading (using the OpenMP directives) and detection of errors via the direct comparison of a selected set of variables specified in the `compare` list `var_list2` which is assumed to be disjoint of `var_list1` the former list assumed to be the *output* of the code block. In other words all the variables in `var_list1` are output variables of the computation so their state need not be saved and restored upon re-execution.

For a simple but generic code, the `dual_redundancy` directive can be translated into the source code as shown below with a maximum retry of 2 times. The `triple_redundancy` variant would include three OpenMP threads and three, rather than two, copies of the variable specified in the `compare` list.

```
int fs_num_tries;
volatile int fs_num_errors;
< declaration of duplicated of variables in var_list2>
...
fs_num_tries = 0;
fs_num_errors = 0;
<code for saving data in var_list1>
do
    if (fs_num_tries != 0){
       <code for restore data in var list1>
    }
    fs_num_errors = 0;
```

```
#omp parallel num_threads(2)
{
    <original code relabeling variables in var_list2>
}

  <compare variables in var_list2 for each thread>

   if(mismatch(var_list2))
     fs_num_errors++;
    fs_num_tries++;
} while ((fs_num_errors != 0) && (fs_num_tries < 2));
if(fs_num_errors != 0){
  FAIL_SAFE_EXCEPTION();
}
```

The code variant for triple redundancy, includes a voting functions rather than a compare function to determine of the three concurrent threads have executed correctly.

3 Experimental Evaluation

We conducted a set of preliminary experiments to evaluate the ability of the proposed program pragmas to lead to applications that survive uncorrected (but detected) memory errors.

For these experiments we focused on a key numerical kernel code, the popular conjugate-gradient iterative linear system solver for the system $Ax = b$. We used an input 40×32 A matrix with a specific structure with a randomly generated b vector as the linear system to be solved. The algorithm requires about 4 MBytes of storage for the system matrix and 0.223 MBytes for the auxiliary intermediate computation vectors. As the system matrix A remains constant throughout the computation, we opted to use checksum column- and row-wise error correction for detected but uncorrected memory errors afflicting the address space regions associated with A.

For these experiments we use the fault-injection infrastructure described in [10] to inject memory errors in the data address space of the application at specific error rates, leading to approximately a single memory error per algorithm iteration to one error per 20 iterations (or a single error per system solve cycle). In these experiments we do not inject errors in the code section of the application address space.

For the errors impinging on data section we opt from two different amelioration strategies. When the error impinging on A we recover by executing the error correction using the column- and row-wise checksums and restart the solver iteration. When the memory error impinges on the auxiliary vectors, we restart the iteration of the algorithm using the previous iterations values of the x vector only as all the other vectors used are temporaries[5]

[5] In the parlance of the compiler analysis, there vectors can be privatizable as no data flows across iterations of the loop through them.

The Table 1 below present the numerical results showing the overhead of the use of the `#pragma failsafe save_restore` directive for this example. In the absence of any error of software copy overhead, the specific linear system requires 21 iterations to converge for a preselected numerical convergence tolerance over 10.680 secs for a sequential execution on a desktop computing system.

Table 1. Execution times vs. injected memory rates for CG simple solver.

Error Interval (secs)	Checksum Recovery	Iteration restart Recovery	Algorithm Iterations	Execution Time (secs)	Execution Overhead
2	12	1	34	23.761	122.5%
4	5	1	27	20.537	92.3%
5	4	1	26	18.569	73.9%
10	1	1	23	15.368	4.5%
20	1	0	22	11.310	0.6%

A couple of simple observations are in order. First, in this controlled experiments, all executions are survivable as the error rate is not high enough that the maximum number of retries (set at 2) for the same iteration of the algorithm is ever exceeded. Second, as the storage size of the matrix A dwarfs the storage space of the auxiliary vectors it was thus expected that the number of errors impinging on the matrix A. As such the retries with checksum correctness and copy of previous state are more numerous (and also computationally more expensive) that simple retries where the only the vector x and a couple of integer control variables need to be restated.

4 Implementation Status

We have implemented the parsing and the corresponding source-to-source code transformations of the `#pragma` directives described here in the ROSE compiler infrastructure [16] and tested them for simple C programs. Still, the current implementation has some limitations. First, the code generation for the directives can only support the comparison and voting of the values of either scalar variables or statically declared arrays with compile-time dimension bounds. In other words, we do not yet support the use of dynamically allocated arrays. Second, there is currently no checking of the disjointness of the compare and save/restore list of variables in the redundancy directives. Lastly, there is also no data-flow analysis verification that the variables in the `compare` list are strictly output, *i.e.*, only output variables.

5 Related Work

The most widely HPC programming models do not contain capabilities to offer error resilient application execution. However, various researchers have begun

exploring the possibility of incorporating resiliency capabilities into the programming models. The abstraction of transactions has been proposed to capture programmers fault tolerance knowledge. The basic idea is that the application code is divided into blocks of code at the end of which the results of the computation or communication are checked for correctness before proceeding. If the block execution condition is not met, the results are discarded and the block can be re-executed. Such an approach was proposed for HPC applications through the concept of Containment Domains [6] which are based on weak transactional semantics. They enforce the check for correctness of the data value generated within the containment domain before it is communicated to other domains. These domains can be hierarchical and provide means for local error recovery.

Other research has focused on discovery idempotent regions of code that can be freely re-executed without the need to checkpoint and restart program state. Their original proposal however [15] was based on language level support for C/C++ that allowed the application developer to define idempotent regions through specification of relax blocks and recover blocks that perform recovery when a fault occurs. The FaultTM scheme adapts the concept of hardware based transactional memory where atomicity of computation is guaranteed. The approach entails application programmer created vulnerable sections of code for which a backup thread is created. Both the original and the backup thread are executed as atomic transactions and their respective committed result values compared [17].

Complementary to approaches that focus on resiliency of computational blocks, the Global View of Resiliency (GVR) project [8] concentrates on application data and guarantees resilience through multiple snapshot versions of the data whose creation is controlled by the programmer through program annotations. Bridges *et al.* [3] proposed a `malloc_failable` that uses a callback mechanism with the library to handle memory failures on dynamically allocated memory, so that the application programmer can specify recovery actions. In the Global Arrays Partitioned Global Address Space (PGAS) implementation, set of library API for checkpoint and restart with bindings for C/C++/FORTRAN the enable the application programmer to create array checkpoints [7].

6 Conclusion and Future Work

The very limited experiments presented here do confirm the potential benefits of the programming language extension to increase the survivability rate of iterative scientific algorithms such as the case of *conjugate gradient* linear solvers. Here we have only exploited the use of a limited form of computation redundancy for error detection and amelioration.

Clearly, an extension of the practical impact of the use of the proposed `#pragma` directives needs to be carried out, in particular to algorithms other than scientific iterative solvers. In particular, we are actively working on the concurrent threading implementation which require the manipulation of the state of the shared cache storage.

This work also suggests a richer interface that would allow programmer to control the need to restore state of the program based on the progress of the algorithm. This is the case of storage whose life-time includes long periods of inactivity and can thus be considered intermittently dead or simply not contributing to the corruption of further state. Such interface would clearly allow a run-time system to use less expensive recovery strategies than a full-blown computation or iteration restart.

Acknowledgment. Partial support for this work was provided by the US Army Research Office (Award W911NF-13-1-0219) and through the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under award number DE-SC0006844. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

1. Bergman, K., et. al: ExaScale computing study: technology challenges in achieving exascale systems (2008)
2. Bosilca, G., Delmas, R., Dongarra, J., Langou, J.: Algorithmic Based Fault Tolerance Applied to High Performance Computing. CoRR abs/0806.3121 (2008)
3. Bridges, P.G., Hoemmen, M., Ferreira, K.B., Heroux, M.A., Soltero, P., Brightwell, R.: Cooperative application/OS DRAM fault recovery. In: Alexander, M., et al. (eds.) Euro-Par 2011. LNCS, vol. 7156, pp. 241–250. Springer, Heidelberg (2012). doi:10.1007/978-3-642-29740-3_28
4. Cappello, F., Geist, A., Gropp, W., Kale, L., Kramer, W., Snir, M.: Toward exascale resilience. Int. J. High Perform. Comput. Appl. **23**(4), 374–388 (2009)
5. Casas, M., de Supinski, B.R., Bronevetsky, G., Schulz, M.: Fault resilience of the algebraic multi-grid solver. In: Proceedings of the 26th ACM International Conference on Supercomputing, ICS 2012, pp. 91–100 (2012)
6. Chung, J., Lee, I., Sullivan, M., Ryoo, J.H., Kim, D.W., Yoon, D.H., Kaplan, L., Erez, M.: Containment domains: a scalable, efficient, and flexible resilience scheme for exascale systems. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, CA, USA, pp. 58:1–58:11. IEEE Computer Society Press, Los Alamitos (2012)
7. Dinan, J., Singri, A., Sadayappan, P., Krishnamoorthy, S.: Selective recovery from failures in a task parallel programming model. In: Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID 2010, pp. 709–714. IEEE Computer Society, Washington, DC (2010)
8. Fujita, H., Schreiber, R., Chien, A.: Its time for new programming models for unreliable hardware, provocative ideas session. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2013)
9. Huang, K.H., Abraham, J.A.: Algorithm-based fault tolerance for matrix operations. IEEE Trans. Comput. **33**(6), 518–528 (1984)
10. Hukerikar, S., Diniz, P., Lucas, R.: A programming model for resilience in extreme scale computing. In: Proceedings of the Dependable Systems and Networks Workshops (DSN-W), June 2012

11. Hukerikar, S., Diniz, P., Lucas, R., Teranishi, K.: Opportunistic application-level fault detection through adaptive redundant multithreading. In: Proceedings of the International Conference on High Performance Computing Simulation (HPCS), pp. 243–250, July 2014
12. Hukerikar, S., Teranishi, K., Diniz, P., Lucas, R.: An evaluation of lazy fault detection based on adaptive redundant multithreading. In: Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–6, September 2014
13. Dongarra, J., et al.: The international exascale software project roadmap. Int. J. High Perform. Comput. Appl. **25**(1), 3–60 (2011)
14. Jian, X., Sartori, J., Duwe, H., Kumar, R.: High performance, energy efficient chipkill correct memory with multidimensional parity. IEEE Comput. Archit. Lett. **12**(2), 39–42 (2013)
15. de Kruijf, M., Nomura, S., Sankaralingam, K.: Relax: an architectural framework for software recovery of hardware faults. In: Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA 2010, NY, USA, pp. 497–508. ACM, New York (2010)
16. Quinlan, D., et. al: The ROSE Compiler Infrastructure. http://rosecompiler.org
17. Yalcin, G., Unsal, O., Cristal, A.: FaulTM: error detection and recovery using hardware transactional memory. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE), DATE 2013, San Jose, CA, USA, pp. 220–225 (2013)