LLNL-CONF-541631

ROSE::FTTransform - A Source-to-Source Translation Framework for Exascale Fault-Tolerance Research

J. Lidman, D. Quinlan, C. Liao, S. McKee

March 26, 2012

2nd International Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS 2012)
Boston, MA, United States
June 25, 2012 through June 28, 2012

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

ROSE::FTTransform – A Source-to-Source Translation Framework for Exascale Fault-Tolerance Research

Jacob Lidman*†, Daniel J. Quinlan*, Chunhua Liao*, Sally A. McKee†
*Lawrence Livermore National Laboratory
Livermore, CA, USA
{lidman1, dquinlan, liao6}@llnl.gov

†Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
{lidman, mckee}@chalmers.se

***Abstract*—Exascale computing systems will require sufficient resilience to tolerate numerous types of hardware faults while still assuring correct program execution. Such extreme-scale machines are expected to be dominated by processors driven at lower voltages (near the minimum 0.5 volts for current transistors). At these voltage levels, the rate of transient errors increases dramatically due to the sensitivity to transient and geographically localized voltage drops on parts of the processor chip. To achieve power efficiency, these processors are likely to be streamlined and minimal, and thus they cannot be expected to handle transient errors entirely in hardware. Here we present an open, compiler-based framework to automate the armoring of High Performance Computing (HPC) software to protect it from these types of transient processor errors. We develop an open infrastructure to support research work in this area, and we define tools that, in the future, may provide more complete automated and/or semi-automated solutions to support software resiliency on future exascale architectures. Results demonstrate that our approach is feasible, pragmatic in how it can be separated from the software development process, and reasonably efficient (0% to 30% overhead for the Jacobi iteration on common hardware; and 20%, 40%, 26%, and 2% overhead for a randomly selected subset of benchmarks from the Livermore Loops [1]).**

***Index Terms*—High Performance Computing, Redundancy, Fault Tolerance, Exascale, Source-to-Source Compiler**

I. Introduction

Future High Performance Computing (HPC) platforms must perform correctly in the face of transient faults (i.e., soft errors). These errors manifest themselves as bit-flips, and they originate from either external sources (e.g., radiation events) or internal sources (e.g., voltage drops, power supply noise, or leakage). They are *soft* in the sense that they do not permanently damage the device [2], [3]. Soft errors have become a design issue in all segments of computing, largely because decreasing transistor feature sizes and lower voltage levels increase their occurrence [2]. In the future, these errors will become especially troublesome, particularly for exascale systems comprising up to millions of commodity cores [4]. To address this problem, architects have proposed hardware enhancements (ranging from buses [5], to on-chip memory [2], [5], pipelines [6], and functional units [5], [7]). Unfortunately, introducing dedicated hardware requires both time and effort, and this approach may not be feasible for low-volume, emerging architectures.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work was funded by DOE ASCR. LLNL-CONF-541631

Large-scale supercomputers have historically been built entirely from commodity parts that are not designed for use at such scales and that lack dedicated hardware support. In this context, scalable software solutions emerge as attractive (defensive) alternatives. Nonetheless, software approaches can incur high performance overheads and should be introduced with care. Furthermore, the fault-tolerant software itself can potentially introduce new errors, since it adds to the existing complexity of developing application software. These trade-offs need to be carefully balanced, and thus we study the problem of automatically introducing fault tolerance in software for hardware-induced faults. In contrast to other uses [8] of the term "*software fault tolerance*", we assume that the given software is free from errors. This is generally referred to as *"software-implemented hardware fault tolerance"* (SIHFT).

Our motivation is to simplify and to increase the efficiency of introducing the necessary software constructs. To this end, we implement a translator using the ROSE source-to-source compiler infrastructure to detect and handle transient processor errors [9], [10]. In addition, we specifically evaluate both whether the compiler-based resiliency transformations make sense and whether they can be implemented via a source-to-source approach. If so, then this research area becomes significantly more accessible to the entire HPC community.

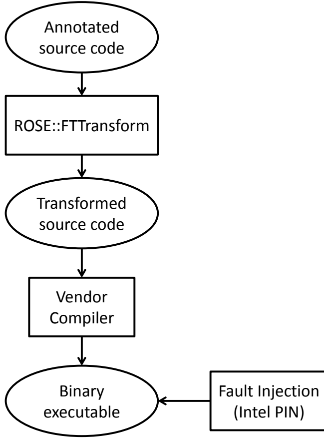

Fig. 1. Fault-Tolerance Research Using ROSE::FTTransform

Controls	Second-chance	Final-wish
Attributes	**NUM-ITER** – Max. number of iterations.	**STM** – Statement to be executed. **ALWAYS-EXEC** – Always execute statement.
Pragma	**#pragma** FT-SC(NUM-ITER)	**#pragma** FT-FW(STM, ALWAYS-STM)

Semantics	`for(int rI = 0; ; rI++) {`
	`y[0...N−1] = ...`
	`Y = PICK_RANDOM(y[0],...,y[N−1])`
	`if(EQUALS(y[0],...,y[N−1],Y))`
	`break`
	`else if (rI == NUM–ITER)`
	`NEXT–POLICY`
	`}`
		`y[0...N−1] = ...`
		`Y = PICK_RANDOM(y[0],...,y[N−1])`
		`if(!EQUALS(y[0],...,y[N−1],Y)) {`
		`STM`
		`(NEXT–POLICY)`
		`} else if(ALWAYS–EXEC)`
		`STM`

Fig. 2. Semantics of Fault-Handling Policies

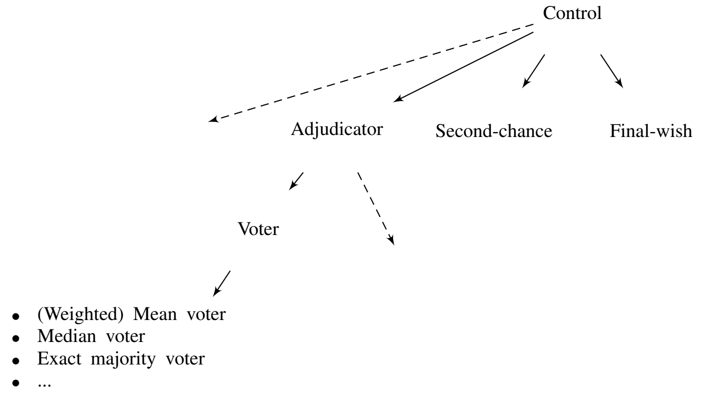

Fig. 3. Structure of Fault-Handling Policies

II. Approach

Figure 1 shows how we use a source-to-source translator (ROSE::FTTransform) to support adding resilience to HPC applications. In particular, ROSE::FTTransform is a tool to insert both fault detection and fault handling mechanisms into a given input source code. Our work has initially focused on application kernels, but the approach can also be driven via source code pragmas (shown in Figure 2), which enable whole-application SIHFT. Ongoing work attempts to measure the sensitivity to errors of HPC applications (to form a more completely automated approach), but we report here only on the support derived from user-directed specification of where in an application to apply our transformations. A vendor compiler generates executables from the transformed code.

Our ongoing work defines a framework to support such compiler-based resiliency research. We have implemented a flexible system supporting an extensible range of techniques and policies to support software resiliency. We expect that a full range of techniques will be required in practice; our work to date represents only a piece of this. To simplify community access to this work and to motivate collaborations with others, our framework will be included in the next release of the open source ROSE source-to-source compiler.

We have measured the overhead introduced by our current resiliency transformations and have analyzed the complexity of using a source-to-source approach in optimizing compilers. Our focus has been specific to what we expect to see in future exascale architectures. In all cases, we consider our approach to be an alternative to a much more expensive check-point restart system that may or may not be tractable at exascale.

A. Fault-Handling Policies

The semantics of fault-handling, illustrated in Figure 2, define how a single statement is transformed into a semantically equivalent, hardened version. This approach to fault tolerance performs N redundant computations and checks for inequalities among the N results. Results are saved into N temporary variables (y[0] to y[N−1]). Of these, one is randomly picked and returned in the variable Y as the result. Both the temporary and the result variables are checked for consistency by invoking EQUALS(y[0], ..., y[N−1], Y). If the results are inconsistent the fault handler is invoked.

As shown in Figure 3, fault-handling policies can have a hierarchical structure. Some of the policies serve merely as control extensions to a possible next-level policy. This allows more fault categories to be handled. An example of this is the second-chance policy that handles transient faults of varying duration, and if the fault remains after the specified number of iterations, a secondary policy is invoked. The final-wish policy allows an arbitrary statement to be executed before invoking a secondary policy. For instance, the specified statement could clean up possible side-effects. Terminal policies, on the other hand, make the final decision on how to unify the results of the redundant computations. This category includes the adjudicator policy that implements common voting strategies [11]. Currently, our framework includes a (weighted) mean voter, a median voter, and an exact majority voter based on the MJRTY algorithm [12].

B. Optimizer-Proof Source Code Redundancy

ROSE::FTTransform detects transient processor faults via redundant execution of critical source code statements. However, naive duplication of source code does not work well with back-end compilers that detect and remove redundant computation via common subexpression elimination (CSE). We therefore design a special source code translation to preserve the redundant computation even when compiler optimizations are enabled.

Figure 4 shows the original kernel (kernel1() at line 2) of a Jacobi iteration on three points of a 1-D array. Many N-modular redundancy systems can be implemented by extending a double modular redundancy (DMR) system in which additional N-2 redundancy is introduced only when the baseline DMR reports issues. In the transformed code (kernel2() at line 11), instead of naively duplicating the statement to be protected (line 18), we introduce a pointer (c2 at line 19) to access all array elements. The pointer is obtained through an additional function argument of kernel2(). The pointer is assigned the correct value at the kernel's call site, before kernel2() is invoked. From the point of view of a compiler without interprocedural pointer analysis, the right-hand expression of the duplicated statement (line 19) may access totally different array elements from the right-hand expression of the original statement. The redundant right-hand side computation will thus survive CSE.

```
1  /* Original Jacobi 1-D , 3-points computation kernel */
2  void kernel1()
3  {
4    int i;
5    for (i=1; i<SIZE-1; i=i+1)
6    {
7      d[i] =  0.25*c[i-1] + 0.5*c[i] +0.25*c[i+1];
8    }
9  }
10 /* Transformed kernel with redundant computation */
11 void kernel2(double *c2 )
12 {
13   double B_intra[3];
14   int i;
15   for (i=1; i<SIZE-1; i=i+1)
16   {
17     /* Baseline double modular redundancy (DMR) */
18     B_intra[0]= 0.25*c[i-1]+0.5*c[i]+ 0.25*c[i+1];
19     B_intra[1]= 0.25*c2[i-1]+0.5*c2[i]+ 0.25*c2[i+1];
20     d[i]= B_intra[0];
21     if (!equal(B_intra[0],B_intra[1],d[i] )
22     {
23        /* Additional N-2 redundancy and
24           fault handling mechanism omitted here... */
25     }
26   }
27 }
28 ...
29 /* call site doing pointer declaration and assignment */
30  double *c2 = c;
31  kernel2(c2);
```

Fig. 4. Source Code Redundancy

C. Implementation

ROSE::FTTransform is built using the ROSE source-to-source compiler infrastructure [9], [10], an open source infrastructure to build source-to-source program transformation and analysis tools for large-scale Fortran 77/95/2003, C, C++, OpenMP, and UPC applications. Internally, it generates a uniform abstract syntax tree (AST) as its intermediate representation (IR) for input codes. Sophisticated compiler analyses, transformations, and optimizations are developed on top of the AST and encapsulated as simple function calls that tool developers can readily leverage. ROSE is particularly well suited for building custom tools for static analysis, program optimization, arbitrary program transformations, domain-specific optimizations, performance analysis, and cyber-security.

The translator is initialized by defining which fault-handler configuration(s) will later be used. Each actual transformation depends on a traversal function to retrieve applicable statements. In particular, the right-hand expression of a statement may have side effects (via function calls, for example) that invalidate the legitimacy of redundant computation. Side effect analysis becomes essential for detecting such situations. An AST visitor function is used to collect statements following "***#pragma*** *resiliency*" nodes. AST transformation is then performed to add fault detection and handling support for the collected statements.

III. Results

A. Experimental Environment

As test inputs, we choose a set of computation kernels that include the Livermore Loops [1], Jacobi iteration, and some specific stencils. Our test platform is Hopper, a Cray XE6 machine provided by the National Energy Research Scientific Computing Center (NERSC). The node we use runs 64-bit SUSE Linux Enterprise Server 11.1 on four quad-core AMD Opteron processors and 129 GB main memory. Each of the AMD processors has 64KB L1 data cache (with a 64-byte line size), 512KB L2 cache, and 6MB L3 cache. The compiler we use is GCC 4.3.4.

B. Effectiveness of Optimizer-Proof Code Redundancy

We use hardware counters to determine whether redundant statements introduced for fault detection survive compiler optimizations. We instrument the kernel in Figure 4 using PAPI [13] to collect the number of floating point instructions (PAPI_FP_INS). Compared to the original version, the reported number of floating point instructions is doubled for the transformed kernel (for all optimization levels [O1 to O3] of GCC 4.3.4), which means that the introduced redundant computation successfully survived the GCC optimizations.

To check the necessity of our special transformation, we tested two alternatives: 1) naive duplicating of source code; and 2) burying naively duplicated statements in a new basic block. In both cases, the reported number of floating point instructions of the transformed code is the same as that of the original kernel for all optimization levels. This means that: 1) naive redundant computation could not even survive optimization level O0, and 2) GCC uses global common subexpression elimination (CSE) over each function entirely, instead of just applying it locally within each basic block.

C. Overhead of Code Redundancy

To study the fixed overhead of duplicated execution introduced by our fault-tolerance transformation (excluding overhead from the incidental N-2 redundancy and fault-handling mechanism), we use three versions of Jacobi iteration micro-kernels: 1D with one point, 1D with three points, and 2D with five points. The fixed overhead is calculated as $(T_trans - T_orig)/T_orig$, where T_orig denotes the execution time of the original kernel, and T_trans denotes the execution time of the transformed kernel with redundant computation for fault detection (with the optional handling mechanism not activated).

Theoretically, duplicated execution should have little impact on the final execution time if the original execution suffers many cache misses or bandwidth limits that leave room to hide the cost of computation. Similarly, if the original execution is already computation-bound, the duplicated execution cannot enjoy this benefit. To prove this theory, we conduct experiments varying factors directly related to memory latencies: 1) using different sizes for the input data set, including larger arrays that cannot fit in cache; 2) using different iteration strides (one and eight) to access array element such that accessing non-consecutive elements touches different cache lines, causing more cache misses; and 3) using different element sizes (single and double precision), where the larger array elements demand more memory bandwidth.

	1-D 1-Point	**1-D 3-Point**	**2-D 5-Point**
Iteration stride = 1			
Array size: 1 million for 1-D, 4096x4096 for 2-D			
float	17.33%	29.91%	30.19%
double	27.55%	22.27%	22.60%
Array size: 16 million for 1-D, 16Kx16K for 2-D			
float	13.87%	25.58%	25.03%
double	17.43%	19.97%	17.37%
Iteration stride = 8			
Array size: 1 million for 1-D, 4096x4096 for 2-D			
float	8.36%	19.12%	25.39%
double	5.10%	6.33%	5.44%
Array size: 16 million for 1-D, 16Kx16K for 2-D			
float	3.57%	10.30%	14.54%
double	0.05%	0.80%	1.59%

TABLE I
FIXED OVERHEAD OF OUR TRANSFORMATION, USING GCC -O3

Table I shows that the fixed overhead introduced by duplicating execution of key kernel statements ranges from 0% to 30%. It is obvious that this overhead is heavily influenced by the memory latencies of the original code: the more latency, the better hidden the cost of the duplicated computation. The minimum overhead is observed when the Jacobi iteration uses a non-consecutive stride, a large data set, and a large element size. Out of the three factors we explore, changing stride dominates the impact on overhead.

Note that this study is a worst-case analysis based on sequential execution of kernels, and thus only the uniprocessor (single-threaded) memory access latency can be exploited to hide duplicated computation. We expect that exascale computing applications will provide more opportunities to hide the overhead (e.g., latencies caused by multi-threading, data movement across CPUs and GPUs, and MPI calls).

As additional data points, we apply the resiliency transformation to a few randomly selected benchmarks (#1, #4, #5, and #11) from the Livermore Loops, for which we observe overheads of 20%, 40%, 26%, and 2%, respectively. Ongoing work is evaluating more of these benchmarks to generate more data beyond the initial results presented here.

D. Fault Coverage

In evaluating the fault coverage of our approach we again choose kernels #1, #4, #5, and #11 of the Livermore Loops suite. For each kernel, we compare the original kernel to two versions produced using the proposed framework.

- **Mean** – DMR with a possible extra computation and mean voting is added.
- **Exact** – DMR with a possible extra computation and exact majority voting is added.

We use the dynamic analysis Intel PIN infrastructure [14] to inject faults into the execution and to measure how well we have supported detection and correction of these faults during execution. During an initial training run, the number of dynamic instructions (within the application — we ignore library functions) and the correct outputs are recorded. In subsequent testing runs (in our case 500), we randomly choose one dynamic instruction and one input register (among the general-purpose and floating-point registers). We flip a random bit in the input register and track the termination condition or output of the program. Each run is categorized as *correct output*, *fail silent*, *access fault* (invalid memory access), *arithmetic error* (e.g., divide by zero), or *invalid instruction* (in the current evaluation, the latter two categories were not applicable to the mentioned kernels or the fault-injection configuration used). This is similar to the approach used by Zhang et al [15].

Figure 5 illustrates the results of the evaluation. Although the framework is able to improve fault coverage in the first kernel, it makes only marginal improvements in the remaining three. In kernel #4 the framework adds excessive redundancy in some cases, without being able to correctly update the read references to the new variable with the redundant values. This causes subsequent calculations to use values that haven't been checked for faults, which increases the chance of propagating a fault. Kernels #5 and #11 perform many more (array based) memory accesses than calculations. Although this exposes more opportunities to hide the overhead of redundant computations, it also adds more single points of failures as a fault in the array index could result in an access fault.

IV. RELATED WORK

Several semi-automated approaches to software fault tolerance have been proposed. The Master-Slave pattern [16] performs requested functions in N-Version Programming (NVP). Daniels et al. [17] extend this by including other patterns, such as recovery blocks, as building blocks. The blocks can

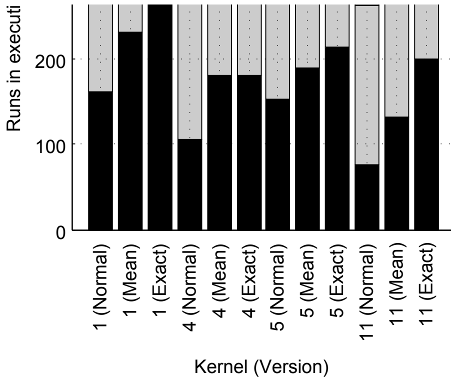

Fig. 5. Fault coverage evaluation of Livermore loops kernels #1, #4, #5 and #11

be recursively combined, and different adjudicators allow the programmer more flexibility in choosing a fault-handling strategy. Although semi-automated approaches are useful in implementing SIHFT, we believe that much of the required information for implementing fault tolerance can be deduced by analysis of the source code. Together with a fault-handling specification, the compiler can produce a hardened and semantically equivalent implementation.

Previous work on compiler solutions to fault tolerance include control-flow checking [18], [19], algorithm-based fault tolerance (ABFT) [20], heuristic rules [21], [22], assertions [23], and process [24] and instruction duplication [25], [26]. These methods range from specialized (efficient but only applicable to certain parts/algorithms of the program) to general (protecting whole programs but potentially compromising performance). A major obstacle to software-based redundancy is its high overhead in terms of code size and execution time (Rebaudengo et al. [22] report code size increases of 4.72 times and slowdowns of 4.56 times). Minimizing overhead requires the consideration of and careful choice among multiple methods. For example, Yu et al. [27] exploit the inherent redundancy in programs and use boolean logic to find outcome-tolerant branches, which can help reduce the overhead. Our work provides source code annotations to selectively harden critical portions of programs.

V. Conclusion and Future Work

Our work uses the compiler to introduce redundant computations into source code in order to detect transient processor faults and to recover from them. Two major concerns have been explored. The first concern is the overhead introduced by our approach. N-way replication of computation in a computationally bound kernel multiplies the overhead by N, but we have demonstrated that double modular redundancy (e.g., $N = 2$) can be used as a first step to reduce the overhead of larger N ($N >= 3$). The overhead is sensitive to the input code and is tied directly to the extent to which the original code is computationally bound. In the worst case the overhead could reach 100%. In our tests, the overhead is typically significantly less, since many kernels are memory-intensive. Where the overhead is judged to be excessive, this may drive compiler-based resiliency transformations to be used selectively. In the future, compiler analyses can determine where to apply redundancy.

The second concern is the impact of compiler optimizations on the redundant computation introduced for fault detection. Our work has shown that naively duplicated computation cannot survive even the lowest level of optimization. Exploiting the difficulty of interprocedural pointer analysis, we designed a special transformation that avoids the negative impact of compiler optimizations on the desired redundant computation.

Given that not all Single Event Upset (SEU) events [15], [24] lead to a fault, we are currently looking into minimizing the number of redundant computations by taking into account the failure probability of the operation and its sensitivity to input changes. Finally, introducing redundancy at the high-level source may have the drawback that preserving the redundant computation may require that compiler optimizations be disabled or made more complicated, which could seriously hamper the introduction of resiliency transformations. More research is required to explore this subject.

References

[1] LLNL, "Livermore loops benchmark," http://www.wikipedia.org/Livermore_loops.

[2] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, ser. DSN '04. Washington, DC, USA: IEEE Computer Society, 2004.

[3] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, sept. 2005.

[4] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward exascale resilience," *International Journal on High Performance Computing Applications*, vol. 23, pp. 374–388, November 2009.

[5] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama, "A 1.3 ghz fifth generation sparc64 microprocessor," in *IEEE International Solid-State Circuits Conference (ISSCC), 2003. Digest of Technical Papers.*, vol. 1, Feb 2003, pp. 246–491.

[6] T. Austin, "Diva: a reliable substrate for deep submicron microarchitecture design," in *International Symposium on Microarchitecture (MICRO'99*, 1999, pp. 196–207.

[7] J. Wakerly, "Partially self-checking circuits and their use in performing logical operations," *IEEE Transactions on Computers*, vol. C-23, no. 7, pp. 658–666, july 1974.

[8] V. D. Florio and C. Blondia, "A survey of linguistic structures for application-level fault tolerance," *ACM Comput. Surv.*, vol. 40, no. 2, pp. 6:1–6:37, May 2008.

[9] D. Quinlan *et al.*, "ROSE Compiler Infrastructure," http://www.rosecompiler.org/.

[10] D. Quinlan and C. Liao, "The ROSE Source-to-Source Compiler Infrastructure," in *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011*, Galveston Island, Texas, USA, Oct. 2011.

[11] D. Blough and G. Sullivan, “A comparison of voting strategies for fault-tolerant distributed systems,” in *Symposium on Reliable Distributed Systems*, oct 1990, pp. 136 –145.
[12] R. S. Boyer and J. S. Moore, “Mjrty - a fast majority vote algorithm,” in *Automated Reasoning: Essays in Honor of Woody Bledsoe*. Kluwer Academic Publishers, 1991, pp. 105–117.
[13] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, “A scalable cross-platform infrastructure for application performance tuning using hardware counters,” in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, November 2000.
[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: building customized program analysis tools with dynamic instrumentation,” in *Programming Language Design and Implementation*, ser. PLDI ’05, 2005, pp. 190–200.
[15] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August, “Daft: decoupled acyclic fault tolerance,” in *International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’10. ACM, 2010, pp. 87–98.
[16] F. Buschmann, “Pattern languages of program design,” J. O. Coplien and D. C. Schmidt, Eds., New York, NY, USA, 1995, ch. The Master-Slave Pattern, pp. 133–142.
[17] F. Daniels, K. Kim, and M. Vouk, “The reliable hybrid pattern: a generalized software fault tolerant design pattern,” in *Pattern Languages of Programming (PLoP)*, 1997, pp. 1–9.
[18] S. Yau and F.-C. Chen, “An approach to concurrent control flow checking,” *IEEE Transactions on Software Engineering*, vol. SE-6, no. 2, pp. 126 – 137, march 1980.
[19] N. Oh, P. Shirvani, and E. McCluskey, “Control-flow checking by software signatures,” *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 111 –122, mar 2002.
[20] K.-H. Huang and J. Abraham, “Algorithm-based fault tolerance for matrix operations,” *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518 –528, june 1984.
[21] A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri, “A c/c++ source-to-source compiler for dependable applications,” in *International Conference on Dependable Systems and Networks (DSN), 2002.*, 2000, pp. 71 –78.
[22] M. Rebaudengo, M. Reorda, M. Violante, and M. Torchiano, “A source-to-source compiler for generating dependable software,” in *First IEEE International Workshop on Source Code Analysis and Manipulation, 2001.*, 2001, pp. 33 –42.
[23] M. Rela, H. Madeira, and J. Silva, “Experimental evaluation of the fail-silent behaviour in programs with consistency checks,” in *Proceedings of Annual Symposium on Fault Tolerant Computing, 1996.*, jun 1996, pp. 394 –403.
[24] A. Shye, J. Blomstedt, T. Moseley, V. Reddi, and D. Connors, “Plr: A software approach to transient fault tolerance for multicore architectures,” *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 2, pp. 135 –148, april-june 2009.
[25] N. Oh, P. Shirvani, and E. McCluskey, “Error detection by duplicated instructions in super-scalar processors,” *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63 –75, mar 2002.
[26] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, “SWIFT: software implemented fault tolerance,” in *International Symposium on Code Generation and Optimization (CGO), 2005.*, march 2005, pp. 243 – 254.
[27] J. Yu, M. J. Garzarán, and M. Snir, “Languages and compilers for parallel computing,” V. Adve, M. J. Garzarán, and P. Petersen, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. Techniques for Efficient Software Checking, pp. 16–31.