Virtual Duplex Systems in Embedded Environments

Dipl.-Inf. Andreas Grävinghoff, FernUniversität-GHS Hagen, FB Informatik, D-58084 Hagen, Germany
Prof. Dr. Jörg Keller, FernUniversität-GHS Hagen, FB Informatik, D-58084 Hagen, Germany

Abstract

Structural redundancy in the form of traditional duplex systems is commonly used for fault detection. Recently an alternative, called virtual duplex systems, has emerged. These systems use temporal instead of structural redundancy, providing very good fault detection on transient hardware failures. Permanent faults are covered if systematic code diversity is applied in addition to traditional design diversity. Since embedded systems are cost-sensitive, the benefits (e.g. one instead of two processors) of virtual duplex systems provide a strong motivation to use these systems in dependable embedded environments. As embedded environments frequently contain some real-time requirements because of their interactive nature, overhead due to temporal redundancy must be low. As context-switch time constitutes a significant fraction of this overhead, we propose to use threads instead of processes to reduce the overhead in the error-free case and allow for faster detection of faults. Instead of using POSIX threads, we propose to obtain lower overhead with emulation of multithreading. This technique allows very fine-grain execution and very short times between checkpoints.

1 Introduction

Embedded systems play an important role in our daily lifes: We interact, sometimes even unknowingly, with an increasing number of embedded systems in applications ranging from cars, trains and aircraft to washing mashines and other household appliances. In the future, even our clothes may contain some form of embedded system [2]. As the number of embedded systems increases, the usage of such systems in safety-critical applications does accordingly. In safety-critical applications, the ability to detect and/or tolerate transient and permanent faults is very important. In every case, some form of redundancy (i.e. multiplication of resources) is required in order to achieve this goal. In the past, some form of duplex system (i.e. structural redundancy) has been used for safety-critical applications. However, the cost associated with duplication of hardware resources is a major drawback of duplex systems. In the case of embedded systems, which are usually deployed in very high volumes on highly competitive markts (e.g. household appliances), there is a strong motivation to lower costs. In this paper we present a new approach that avoids the high costs associated with traditional duplex systems without sacrificing the abilty to detect/tolerate faults.

Our approach is based on virtual duplex systems [4]. Virtual duplex systems (VDS) use temporal instead of structural redundancy to detect faults. Experimental results show that such systems provide excellent detection of faults in the case of transient failures. In the coverage of permanent failures, the use of systematic diversity in combination with design diversity leads to encouraging results [9]. A virtual duplex system requires only a single node, while an extension to fault-tolerance is possible with as little as two nodes [8]. Since virtual duplex systems use temporal redundancy, their use might interfere with real-time requirements of the embedded application. For example, an embedded system controlling the movements of a roboter arm allows only little time for the detection of faults, since the next movement should be initiated only after the absence of faults has been asserted.

Since virtual duplex systems use several processes, the duration of context switches between processes is a constraining factor to the fast detection of faults, i.e. the percentage of context switch time increases relatively to user time as the time available for fault detection/resolving decreases. The overhead associated with context switching is a well-known problem in the area of operating systems. Modern operating systems therefore support threads, which use resource sharing (address space, open files, etc.) in order to decrease context switch times. However, even with threads, a context switch is a quite expensive operation. We therefore propose to use emulated multithreading, which was originally developed to hide memory latency in massively parallel computers [5]. Emulation of multithreading enables very fast context switches and thus very small grain-sizes of threads.

The remainder of this paper is organized a follows: virtual duplex systems and their characteristics are introduced

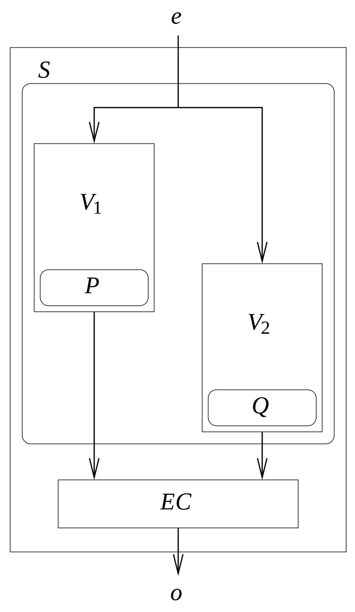

Figure 1 Basic Structure of Virtual Duplex System

in the next section. The third section covers multithreading and describes emulation of multithreading in detail. The fourth section includes a comparison between context switch overheads based on the analysis of register usage in a popular parallel benchmark and addresses problems associated with the resource sharing of threads. The last section summarizes our approach and identifies possible applications.

2 Virtual Duplex Systems

2.1 History

As already mentioned in the previous section, redundancy is required to support fault detection/tolerance. Structural redundancy in the form of duplex systems is commonly used to support fault detection. During the last two decades, the use of temporal redundancy emerged, especially in the area of circuit design: alternating circuits [16], alternate data retry [18], recomputing with shifted operands [14]. Temporal redundancy has also been used in connection with design diversity to detect design faults: self-reducible functions [17],[6], certification trails [19], program checking [1]. However, temporal redundancy in connection with design diversity aiming at detection of permanent as well as transient hardware failures was first discussed by Echtle et al [4]. The virtual duplex systems introduced in their work are described in the next subsection. Experiments performed by Echtle's group showed that design faults and transient hardware failures were detected with very good fault coverage. The system did provide limited coverage of permanent hardware failures. Based on this work, Lovrić [9] used systematic diversity and diverse error correcting codes to enhance the fault detection capability of virtual duplex systems. The combination of techniques yields very good results even in the presence of permanent hardware failures.

2.2 Basic Structure

The basic structure of a virtual duplex system is depicted in **Figure 1**. We do not consider fault-tolerant input/output devices. Instead we assume that input/output is performed by reading/writing designated memory areas. Furthermore we assume that computation proceeds in rounds, where the result of a deterministic function F is calculated in each round. To facilitate the detection of crash faults, we assume that an upper limit on the time required to calculate F can be given. Based on these assumptions, the following components of a virtual duplex system can be identified:

V_1, V_2: Two different implementations that compute the function F on the input e. The implementations should be diverse, i.e. originate from independent development teams, which is also required for conventional duplex systems. Diversity is further enhanced by systematic diversity as described below. The result $F(e)$ is encoded using different error correcting codes in both implementations. Both encoded as well as both unencoded results are delivered to the error-checker EC.

EC: This module compares the results from the two different implementations and signals a fault on non-equality. First the integrity of the two coded results is ensured by checking that both results are valid code-words. Afterwards both results are compared and possible differences signaled as a fault.

S: This module schedules the execution of the two implementations V_1, V_2. In the absence of crash faults, V_2 is executed after V_1 has finished. In the case of crash faults in V_1 or V_2, an error is signaled to the error checker.

Systematic diversity is an extension to traditional design diversity that maintains the semantics of a program. Lovrić proposed mechanisms to improve diversity covering design (e.g. specification), tools (e.g. compiler switches) and source-code modifications (high-level language and assembler). Some of these mechanisms are system dependent, since characteristics of processor hardware and development tools are utilized. An evaluation of systematic diversity on two programs of medium complexity (quality control and interconnection network managment) yielded very good results: on average, all transient as well as 99.94 % of permanent hardware failures were detected [9]. Lovrić's work did not include recovery from faults. However, well-known techniques like checkpointing can be applied to virtual duplex systems.

3 Emulation of Multithreading

3.1 Introduction

As described in the previous section, virtual duplex systems use seperate processes for the different software variants. A context switch consists of saving the state of the old process and loading the saved state for the new process. A context switch is therefore a time-consuming operation. For example, a modern operating system like Solaris requires 123 μs or 4920 cycles [1] [11] per context switch. To reduce the overhead associated with switching between processes, modern operating systems support threads.
A thread (or lightweight process) contains the program counter, register set and stack space. Code and data sections as well as resources (e. g. open files) are shared with other threads within the same task. A task is a set of threads that share the same resources, therefore a process is a task with a single thread. Compared to processes, switching between threads basically requires only a switch of the register set and is therefore faster. The size of the register set is an important factor for the context switch time of threads. Some operating systems support user-level threads (e.g. POSIX threads), which allow switching between threads without entering the kernel, which is even faster. For example, switching between POSIX threads under the Solaris operating system requires 37 μs or 1480 [1]cycles [11]. Despite this improvements, the overhead is still quite large.

3.2 Basic concept

Emulating fine-grained multithreading on off-the-shelf processors can be done as follows: For each thread, the executed program as well as the processor state of the thread (context) are stored in memory. To execute an instruction block from a given thread, the emulation program restores the processor state, fetches and executes the instructions and updates the processor state in memory. Afterwards, the next thread is executed. The context of a thread (and management information for the emulation library), is stored in a data structure called frame. To decrease the time to switch contexts, only those part of the context used or modified by the fetched instruction is restored or saved.
To implement multithreading, we modify the program code to replace every instruction block I by a subroutine. These subroutines restore every register that will be read or modified by I, execute I, save every register that has been written or modified, and return. The number and location of registers to be read, written or modified by instructions in I can be determined at compile-time from the instruction set architecture and the instructions in

[1]measured on a SparcStation IPX (40 Mhz CPU)

I. The number of instructions per instruction block can be either fixed or variable, user-selected or application-dependent. After this modification, the emulation program merely calls the subroutines from all threads in a round-robin manner.
We assume that the high-level language source code of the program to be emulated is available and that the program already uses threads. The thread-related system calls (e.g. creation, deletion) are then replaced by our own routines during recompilation. After modification of the assembler source the program is linked with a small library containing our routines (e.g. main loop). We further assume that all threads operate in user mode. Thus only the registers accessible in user mode are shared between different threads and form the context of a thread. By confining threads to user mode, the switching time between threads is significantly reduced. Calls to the operating system are not emulated, but are executed as usual by system calls and traps, i.e. there are no changes to the operating system. Obviously, we cannot handle self-modifying code, since we perform all code modifications during compilation. However, this is not a serious restriction since self-modifying code is generally considered as unfavorable.

3.3 Implementation

Several tools were developed to facilitate emulation of multithreading. The program source code is first modified by the high-level language converter *hllconv*, which searches the source file for user-supplied function names and modifies only those functions. This allows emulation of multithreading to be applied on a function-by-function basis, i.e. only on those functions that benefit from the emulation. The modification process works recursively through all of the given functions. Some of the functions detected in this process may be called, either directly or indirectly, by a non-modified function as well as a function to be modified. Therefore all functions to be modified are duplicated before the thread-related function calls are substituted with the corresponding calls to the emulation library *EMUlib*. The modified source code is forwarded to the compiler, while the list of all modified functions is forwarded to the assembler converter *asmconv*.
The modified source file is processed by a compiler, the resulting assembler source is forwarded to the assembler. If there are no functions to be modified, an object file is generated for later use of the linker. Otherwise the assembler listing output is converted to a valid assembler source by *listconv* before it is passed to the assembler converter. The use of the assembler listing instead of the assembler source allows us to ommit support for the advanced assembler features (e.g. macros) in the assembler converter.
The converted assembler source is processed by the assembler converter, modifiying only those functions given by *hllconv*. If one of those functions is found, the necessary instruction blocks are generated based on the assembler source, the selected grainsize/optimization level as well as the configuration file. This configuration file contains information about instruction styles, modification rules for instructions as well as characteristics of external calls. Since these largely machine-dependent informations are contained within the configuration file, porting of the assembler converter to a different architecture is quite simple.
The assembler converter supports fixed and variable grain-sizes up to several hundred instructions per block. Therefore the number of context switches can be tailored to the application's requirements. In addition, several optimizations that improve instruction scheduling within blocks are available. Calls to external functions are recognized and treated accordingly. After the whole source file has been processed, the assembler generates an object file from the modified assembler source. After all required object files are generated, the linker combines these object files with the standard libraries as well as the emulation library *EMUlib*. This library contains the thread-related functions for emulation of multithreading, i.e managment, communication and synchronization of threads. The resulting executable can be executed in the usual way.
The tools mentioned above currently support the Alpha architecture under the UNICOS/mk operating system as well the SimpleScalar simulation toolset [3]. However, great care was taken to maintain portability.

3.4 Embedded Virtual Systems

In embedded environments, we propose the usage of virtual duplex systems working in the following way: As with conventional virtual duplex systems, a single node alternatively executes two different implementations of the function F mentioned in the previous section. Instead of processes, threads in the form of emulated multithreading are used. We use three different implementations instead of two because we want to tolerate faults. All implemen-

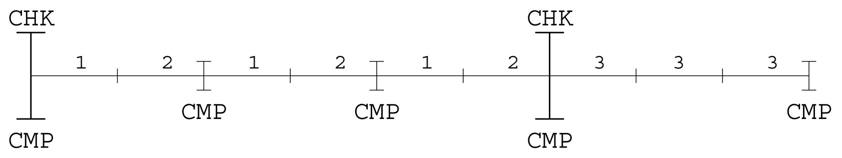

Figure 2 Scheduling of threads

tations are modified as described in the previous section. Diversity between different implementations is ensured in several ways: First traditional design diversity as well as systematic diversity are used during the design phase. If the implementations already use emulation of multithreading (i.e. contain calls to the emulation library), no modifications of the high-level language source code are necessary. Now systematic diversity techniques covering high-level language and design environment are used to generate assembler sources. These sources are modified according to the rules specified in the previous section. Afterwards, systematic diversity techniques are applied at the assembler language level and executables are generated. As proposed by Lovrić, all implementations use error-correcting codes such that encoded as well as unencoded data is sent to the error checker. In addition, a signature of the threads state is calculated by a simple signature function like the sum of all integers modulo 256 at the end of each instruction block. These signatures are used later to detect faults within a single round. Therefore, a common subset of the state must be used to calculate the signature.
The scheduling of threads in our approach is depicted in **Figure 2**. Instruction blocks from the first two implementations (1,2) are executed alternately. After execution of both instruction blocks, the state of both threads is compared by a signature check. After a selectable number of instruction blocks, a checkpoint is generated in addition to the normal signature check, i.e. the state of both threads is saved to disk to enable fault tolerance.In this way, the state of both threads is frequently compared without always incuring the overhead of checkpoint generation.
In our example, the movements of a roboter arm are controlled by an embedded system. The roboter's arm performs a sequence of movements – the next movement is initiated in the absence of errors only. Therefore the time between signature checks is bounded by the duration of a single movement and the maximum possible delay of the next movement, because of the arm's real-time operation. This restriction requires small blocksizes, fast context switches as well as fast generation/comparison of signatures. Decreasing block size while maintaining the same context switch time leads to a proportional increase of overhead, which is usually not desired.
The frequent comparison leads to faster detection of faults, since faults can only be detected at the end of instruction blocks. Since checkpointing is a quite expensive operation (e.g. disk access), the frequent use of checkpoints could lead to unacceptably high overhead. This may even be true in the case of diskless checkpointing [15], which trades checkpointing overhead to memory consumption. In the case of errors, the inactive thread (3) is enabled and its state rolled back to the last known checkpoint. After the thread reaches the point where the previous error was detected, the faulty thread is disabled based on a majority vote. Since all three threads are diverse implementations, the remaining two threads are still diverse and can continue operation.
Our virtual duplex system can be tailored to the application's requirements via several parameters. First, the grain-size (maximum length of an instruction block) can be selected. Since signature checks are performed between instruction blocks, the grain-size determines the frequency of those checks. Note that the maximum time required for each block can be calculated at compile-time assuming a worst-case scenario. Second, the ratio of checkpoints to signature checks is selectable as well. More frequent checkpoints incur more overhead, but reduce the minimum time required to resolve faults. Our virtual duplex system uses three threads, but only two are active in the error-free case. In order to resolve a fault, we do not generate a new thread, but activate the inactive thread instead.

4 Evaluation

The time required for a context switch between threads is determined by three factors: the latency of the emulation loop, the number of registers to be saved to the current thread's context and the number of registers to be restored from the next thread's context. The emulation loop basically calls the threads in a round-robin manner and contains no more than 6 instructions. Even with calls to the added functions (signature check, initiation of checkpoints etc.) the size of this loop should not increase significantly. In the case of POSIX threads, usually all user-level registers have to be saved/restored, i.e. the last two numbers are fixed and equal the size of the register-set as seen by the

programmer. In our approach, these numbers are dependent on the number of registers actually used within a given instruction block. In order to investigate the relation between register usage and grain-size, we gathered register usage statistics for a popular benchmark suite.
The NASPAR benchmark suite [12], [13] was developed at NASA Ames Research Center under the Numerical Aerodynamics Simulation (NAS) Program. The suite consists of eight problems derived from important classes of aerophysics applications:

BT: solves a three-dimensional set of block-triangular equations.

CG: computes the smallest eigenvalue of a matrix using a conjugate-gradient method.

EP: generates pairs of Gaussian random deviates, an ”embarrassingly parallel” program.

FT: solves three-dimensional differential equations based on fast-fourier-transforms.

IS: sorts a large number of integers.

LU: solves a regular, sparse triangular linear system.

MG: solves the discrete Poisson problem using a multigrid method.

SP: solves a three-diemnsional set of scalar pentadiagonal equations.

We generated the assembler source of the eight programs on a Cray T3E using the standard development tools [2]. Based on this data we determined the maximum number of registers used within an instruction block for several grain-sizes (2, 8, 32, 128). The results are depicted in **Table 1**. Results for the eight programs within the benchmark suite are given in separate rows. The last row contains average values over all eight programs. Two different values are reported for each grain-size: the left column reports the maximum number of integer registers accessed within an instruction block, while the right column reports the corresponding value for the floating-point registers. The results were averaged over all instruction blocks and source files if a given benchmark contained more than one source file (BT, LU, MG).
An analysis of the data reported in Table 1 shows that the number of used registers grows less than linear with grain-size. Even for the largest grain-size, the number of integer and floating-point registers used is well below the maximum value, which is favorable to our approach. Note that the Alpha architecture has no more than 31 integer/floating-point registers (apart from one zero source/sink register), which is therefore an upper bound on the number of registers used within an instruction block. Some benchmarks, notably IS, have inherent smaller grainsizes.
Conventional threads perform the register set switch in a single operation. Using emulation of multithreading allows us to postpone the restore of registers until these registers are actually used. The saving of registers can be performed directly after the last modification to a register in the same way. This observation allows us to spread the save/restore operations along the instruction block, possibly filling empty instruction slots. An examination of the NASPAR benchmarks with this respect was performed in a similar way as described above. The results are depicted in **Table 2**, which uses the same format as Table 1. The left and right columns report the last time a register was accessed, relative to the instruction block's length for integer and floating-point registers, respectively. The reported values are averages over all registers. These statistices show that the instructions necessary for a context switch can be distributed among a fraction (at least a fifth) of the instruction block. These instructions can therefore be used to fill empty instruction slots within the instruction stream.
Due to context switch overhead, we used threads instead of processes in our approach. Since traditional POSIX threads as well as our threads use resource sharing (address space, open files, etc.) in contrast to processes, some problems have to be addressed: The first problem is based on the quasi-parallel execution, which requires synchronization between threads that modify shared data structures. Otherwise the atomicity of updates cannot be ensured. Note that the input values used by all threads are only read and therefore require no synchronization. If shared data structures are used, proper synchronization (locks, semaphores, etc.) between threads has to be ensured via design requirements. Alternatively, synchronization is not necessary if updates to shared data structures do not cross instruction block boundaries. For instance, such updates can be marked by the programmer, allowing

[2] -O3 optimization

Table 1 Register Usage vs. Grain-Size

	2		8		32		128	
	int	fp	int	fp	int	fp	int	fp
BT	04.49	04.55	11.04	10.49	17.72	17.19	23.29	20.59
CG	04.86	04.10	12.03	06.25	22.82	06.71	27.90	08.10
EP	04.89	03.41	10.78	04.77	19.26	06.19	26.50	07.67
FT	04.82	04.62	14.00	06.00	21.09	06.90	28.94	07.27
IS	03.99	04.61	07.68	07.41	14.88	09.52	17.17	10.00
LU	04.70	04.39	10.94	10.02	19.36	15.95	26.36	21.32
MG	05.32	04.78	12.81	10.69	22.79	14.40	29.66	15.38
SP	04.46	04.53	12.03	10.43	19.89	16.57	26.71	20.66
$\sum$	04.69	04.37	11.41	08.26	19.73	11.68	25.82	13.87

Table 2 Register Access Times vs. Grain-Size

	2		8		32		128	
	int	fp	int	fp	int	fp	int	fp
BT	0.622	0.317	0.640	0.370	0.669	0.472	0.737	0.609
CG	0.640	0.118	0.607	0.207	0.632	0.352	0.722	0.508
EP	0.657	0.076	0.593	0.127	0.656	0.188	0.805	0.236
FT	0.629	0.082	0.602	0.126	0.637	0.290	0.704	0.461
IS	0.622	0.097	0.605	0.139	0.680	0.307	0.640	0.377
LU	0.589	0.268	0.630	0.309	0.660	0.414	0.727	0.595
MG	0.610	0.111	0.599	0.160	0.634	0.310	0.693	0.435
SP	0.613	0.276	0.612	0.348	0.622	0.536	0.699	0.709
$\sum$	0.623	0.168	0.611	0.223	0.649	0.359	0.716	0.491

the assembler-level converter to use this information while generating instruction blocks. However, there is little reason to believe that read/write access to shared data structures is desired or even required by the different threads, since they are not cooperating but diverse implementation of one function F.

The second problem is based on the lack of protection between threads. Since threads can access each others data structures, a faulty thread can influence other threads. In our approach, address space is partitioned between threads as follows: Input values used by all threads are mapped to a portion of the address space shared by all threads. Since neither the threads nor the error checker require write access to these values, the corresponding memory pages can be mapped read-only. Therefore every write-access causes a segment violation. The same reasoning can be applied to the output values, such that the corresponding memory pages can be mapped write-only. Since we cannot handle self-modifiying code, write access to the threads program code is not required. Again, the corresponding memory pages are mapped read-only, such that threads can only access each others data structures, not their program codes. Every thread as well as the error checker/scheduler uses its own portion of the address space, which contains heap, stack and frame information. In order to protect accesses, the different portions of the address space can be separated by read-only guard pages. The size of these pages is determined by the maximum amount of memory accessed by each thread. On the other hand, virtual duplex systems do not rely on protection between threads due to the use of error-correcting codes [10].

Our approach is extended to fault-tolerance similar as virtual duplex systems: A (dynamic) double virtual duplex system [8] is used, but both nodes are based on our modified virtual duplex system. These systems use duplication of hardware resources (two computing nodes) and four instances derived from three different implementations. The error checker can therefore base its decision on four different results. In the dynamic variant, the error-checker uses a two-phase protocol which allows operation without temporal redundancy in the absence of errors. These systems can tolerate single and detect double faults.

The modifications at the assembler level may increase code-size, since instructions related to context switches are added to the instruction stream. The amount of additional instructions depends on the number of context switches, i.e. the grain-size. Larger grain-sizes lead to less frequent context switches and therefore fewer additional instructions. However, instead of a full-blown thread library, only our small library is added, which may ease the increase in code size. If a comparable duplex system utilizes more than 50 % of each processor, the virtual duplex system must use a faster processor, otherwise the system's temporal behaviour will change. However, a faster processor is usually cheaper than two slower ones. In addition, only one instance of supporting circuitry (memory, I/O, etc.) is required.

5 Conclusions

Virtual duplex systems are a cost-effective alternative to traditional duplex systems. This benefit is especially important for embedded systems which are deployed in very high volumes on highly competitive markets. As the number of embedded systems increases, these systems conquer safety-critical areas requiring fault detection or tolerance. Studies show that virtual duplex systems provide very good fault detection on transient as well as permanent hardware failures. These results were achieved with a combination of design diversity, systematic diversity and diverse error-correcting codes.

However, the overhead associated with context switches between processes may change the temporal behaviour of applications that need fast detection and resolving of faults. Therefore we proposed to use threads to decrease context switch overhead, thus enabling virtual duplex systems in such environments. A comparison of context switch times between processes and threads showed that the overhead associated with threads was smaller, but still relatively large. Therefore we used emulation of multithreading instead of traditional POSIX threads. We performed several experiments to gather data about register usage in connection with grain-size. The results favor our approach since it is not necessary to perform a full register set switch even in the case of large grain-sizes. The corresponding faster context switch time allows the frequent usage of signature checks in order to detect faults. Problems originating from the usage of threads instead of processes were addressed and solutions given. In this work, we covered only a single process. Beyond, our approach can be extended to multiple communicating processes or threads using the techniques described by Sobe [20].

References

[1] Amato, P. E.; Loui, M. C.: Checking Linked Data Structures. Digest of Papers, 24th Fault-tolerant Computing Symposium, pp. 164–173, Los Alamitos, IEEE Press, 1994.

[2] Billinghurst, M.; Starner, T.: New Ways to Manage Information. IEEE Computer, vol. 32, no. 1, pp. 57–64, Los Alamitos, IEEE Press, 1999.

[3] Burger, D.; Austin, T.: The SimpleScalar Tool Set, Version 2.0. Technical Report TR 1342, University of Wisconsin-Madison Computer Sciences Department, 1997.

[4] Echtle, K.; Hinz, B.; Nikolov, T.: On Hardware Fault Diagnosis by Diverse Software. Proceedings of the 13th International Conference on Fault-Tolerant Systems and Diagnostics, pages 362–367, Sofia, Verlag der Bulgarischen Akademie der Wissenschaften, 1990.

[5] Grävinghoff, A.; Keller, J.: How to Emulate Fine-Grained Multithreading. Proceedings of the 2nd IASTED Conference on Parallel and Distributed Computing and Networks, pp. 584–589, Calgary, ACTA Press, 1998.

[6] Lipton, R. J.: New Directions in Testing. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 2, Providence, American Mathematical Society, 1991.

[7] Lovrić, T.: Systematic and Design Diversity - Software Techniques for Hardware Fault Detection. Proceedings of the First European Dependable Computing Conference, pp. 309–326, Berlin, Springer Verlag, 1994.

[8] Lovrić, T.: Dynamic Double Virtual Duplex System: A Cost-Efficient Approach to Fault-Tolerance. Proceedings of the 5th International Working Conference on Dependable Computing for Critical Applications, pp. 35–42, Los Alamitos, IEEE Press, 1995.

[9] Lovrić, T.: Detecting Hardware Faults with Systematic and Design Diversity: Experimental Results. International Journal of Computer Systems Science and Engineering, vol. 11 no. 2, pp. 83–92, London, CLR Publishing Ltd, 1996

[10] Lovrić, T.: Fehlererkennung durch systematische Diversität in entwurfsdiversitären zeitredundanten Rechensystemen und ihre Bewertung mittels Fehlerinjection. PhD thesis, Universität Essen, Germany, 1996.

[11] Mueller, F.: A Library Implementation of POSIX Threads under UNIX. Proceedings of the 1993 Winter USENIX Conference, pp. 29–42, 1993.

[12] Bailey, D.; Barszcz, E.; Barton, J,; Carter, R.;Dagum, L.; Fatoohi, R.; Fineberg, S.; Frederickson, P.; Lasinski, T.; Schreiber, R.; Simon, H.; Venkatakrishnan, V.; Weeratunga, S.: The NAS Parallel Benchmarks. RNR Technical Report RNR-94-007, Moffet Field, NASA Ames Research Center, 1994.

[13] Bailey, D.; Harris, T.; Saphir, W.; van der Wijngaart, R.; Woo, A.; Yarrow, M.: The NAS Parallel Benchmarks 2.0 NAS Technical Report NAS-95-020, Moffet Field, NASA Ames Research Center, 1995.

[14] Patel, J. H.; Fung, L. Y.: Concurrent Error Detection in Multiply and Divide Arrays. IEEE Transactions on Computers, vol. C-32, no. 4, pp. 417–422, 1983.

[15] Plank, J. S.; Li, K.; Puening, M. A.: Diskless Checkpointing. IEEE Transactions on Parallel and Distributed Systems, vol. 9, no. 10, pp. 972–986, Los Alamitos, IEEE Press, 1998.

[16] Reynolds, A.; Metze, G.: Fault Detection Capabilities of Alternating Logic. IEEE Transactions on Computers, vol. C-27, no. 12, pp. 1093–1098, Los Alamitos, IEEE Press, 1978.

[17] Rubinfeld, R.: A Mathematical Theory of Self-Checking, Self Testing and Self Correcting Programs. PhD thesis, University of California, Berkeley, 1990.

[18] Shedletsky, J. F.: Error Correction by Alternate-Data Retry. IEEE Transactions on Computers, vol. C-27, no. 2, pp. 106–112, Los Alamitos, IEEE Press, 1978.

[19] Sullivan, G. F.; Masson, G. M.; Certification Trails for Data Structures. Digest of Papers, 21th Fault-tolerant Computing Symposium, pp. 240–247, Los Alamitos, IEEE Press, 1991.

[20] Sobe, Peter: A Study of Roll-Forward Recovery from Faults among Communicating Processes. Technical report TR 98/02, Technische Universität Dresden, Fakultät Informatik, 1998.