# Evaluation of Thread-Based Virtual Duplex Systems in Embedded Environments

Jörg Keller Andreas Grävinghoff*

LG Technische Informatik II
FernUniversität Hagen
58084 Hagen, Germany
joerg.keller@fernuni-hagen.de

**Abstract:** Virtual duplex systems have emerged as an alternative to traditional duplex systems, trading structural for temporal redundancy. When used in dependable embedded systems, virtual duplex systems provide a cost benefit because they require only one instead of two processors. In order to lighten the burden of this single processor, and in order to obey real-time requirements in embedded systems, the overhead due to the 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 emulated multithreading to further reduce overhead. This technique allows very fine-grain execution and very short times between checkpoints. We evaluate the proposed concepts quantitatively.

## 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 machines and other household appliances. In an increasing number of applications, the ability to detect and/or tolerate transient and permanent faults is important. To achieve this goal, some form of redundancy (i.e. multiplication of resources) is required. In the past, some form of duplex system (i.e. structural redundancy) has been used for these 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 in highly competitive markets (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 [EHN90]. Virtual duplex systems (VDS) use temporal instead of structural redundancy to detect faults. Experimental results show

*Now with ETAS GmbH, Stuttgart, Germany

that such systems provide excellent detection of faults in the case of transient failures. For permanent failures, the use of systematic diversity in combination with design diversity leads to encouraging results [Lov96a]. A virtual duplex system requires only a single processing node, while a traditional duplex system needs two of them. If that duplex system utilizes more than 50% of each processor's performance, the virtual duplex system must use a faster processor. 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 and a smaller printed circuit board can be used.

Since virtual duplex systems use temporal redundancy, their use might interfere with real-time requirements of the embedded application. For example, consider an autonomous guided vehicle that uses a camera to control its movements: The embedded system controlling the vehicle has to issue instructions at a rate proportional to the vehicle speed, i.e. there is little time for the detection of faults because instructions are only issued 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 share resources (e.g. address space, open files, etc.) in order to decrease context switch times. However, even with threads, a context switch is an expensive operation and there are no embedded processors that support multithreading in hardware. We therefore propose to use emulated multithreading, which was originally developed to hide memory latency in massively parallel computers [GK98, Gra02]. Emulated multithreading provides very fast context switches and thus enables very small grain-sizes.

The remainder of this paper is organized a follows: virtual duplex systems and their characteristics are introduced in the next section. The third section reviews multithreading, especially emulated multithreading. The fourth section introduces embedded virtual duplex systems using emulated multithreading. The fifth section presents a quantitative evaluation of virtual duplex system performance. The last section summarizes.

## 2 Virtual Duplex Systems

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 [RM78], alternate data retry [She78], recomputing with shifted operands [PF83]. Temporal redundancy has also been used in connection with design diversity to detect design faults: self-reducible functions [Lip91], certification trails [SM91], program checking [AL94]. Temporal redundancy in connection with design diversity aiming at detection of permanent as well as transient hardware failures is e.g. discussed by Echtle et al [EHN90]. 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 [Lov96a] 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.

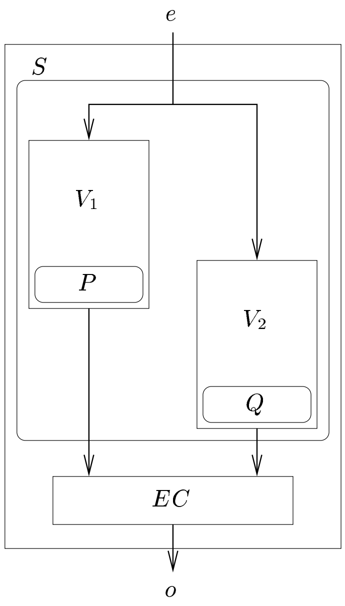

Figure 1: Basic Structure of Virtual Duplex Systems

The basic structure of a virtual duplex system is depicted in Figure 1. We do not consider fault-tolerant input/output devices. Based on the model by Lovrić [Lov96b] we make the following assumptions: 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 $EC$. The result $F(e)$ can be seen as a hash value over the state of the respective process.

$EC$: 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 [Lov96a]. Lovrić's work did not include recovery from faults. However, well-known techniques like checkpointing can be applied to virtual duplex systems and duplex systems alike to achieve fault-recovery. A simple variant is to store the state from time to time. In case of a fault, i.e. in case of different states, a third instance of $F$ is executed, starting from the last stored (valid) state. Now a majority vote over the three state encodings allows to select two identical states, from which the application can continue.

## 3 Emulated Multithreading

Virtual duplex systems as introduced in the last section have been based so far on traditional heavyweight processes known from operating systems. Since switching between these processes is a costly operation, the typical grainsize is on the order of tens of thousands of executed instructions. Modern operating systems also support lightweight processes, i.e. threads, that share some of the resources associated with heavyweight processes, e.g. the address space. Due to this sharing of resources, switching betweeen lightweight processes is less costly compared with their heavyweight counterpart, such that grainsizes on the order of thousands of instructions are possible. POSIX threads (pthreads) [But97] are a popular form of lightweight processes that are supported in almost all modern operating systems. However, the overheads and the corresponding grainsizes of these threads are still too high for our purposes. Fine-grained multithreading, i.e. grainsizes between one and hundreds of instructions, requires low overhead context switches as provided by emulated multithreading [GK98, Gra02]. The concept behind emulated multithreading is reviewed briefly in the next paragraph.

Each thread is defined by its context and the program counter, i.e. the current location within the program code. The context is defined as the processor state of the thread, i.e. the registers that are visible to the application programmer. The context typically contains all general-purpose registers, the program counter as well as any special registers, e.g.

condition codes. The program code is divided into instruction blocks ranging in size from one to several hundred instructions. These instruction blocks are modified in the following way: Each modified instruction block contains the necessary instructions to save/restore that part of the context which is read/modified by the instructions in the original instruction block. In contrast, traditional thread packages save/restore the whole context if execution is switched betweeen threads. To execute a given thread, the modified instruction block pointed to by the thread‘s program counter is called and executed; execution is then switched to the next thread. A complex scheduling algorithm is not required by virtual duplex systems, thus ensuring low overheads. Therefore the main routine from which all threads are executed consists of a simple loop that calls the modified instruction blocks in a round-robin manner.

Our current implementation uses several tools that enable emulated multithreading: A high-level langugage converter (*hllconv*) processes the high-level language (C, Fortran) source code in order to identify procedures that need to be modified. The assembler converter (*asmconv*) modifies the corresponding assembler instruction sequences. The emulation library (*emulib*) contains a lightweight thread package and is linked to all programs after modification by the two converters. Apart from these tools the standard development tools (compiler, assembler, linker) are used in the design flow.

## 4 Embedded Virtual Duplex Systems

In embedded environments, we propose the usage of virtual duplex systems working as follows: Like 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. In total, we use three different implementations instead of two because we want to tolerate faults. All implementations 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 emulated 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 thread’s 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 signature also serves to check whether a faulty thread has modified the other thread’s data, which may happen as there is no memory protection as between processes.

The scheduling of threads in our approach is depicted in Figure 2. Instruction blocks

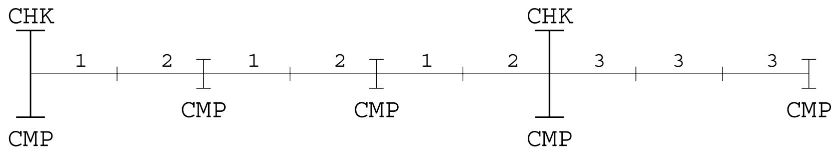

Figure 2: Scheduling of threads

from the first two implementations (labeled 1 and 2 in the figure) 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.

Since, in the presence of real-time requirements and mission-critical control operation, the next control instruction is only issued in the absence of faults, the time between signature checks is bounded by the maximum time between two control instructions. 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 [PLP98], which trades checkpointing overhead to memory consumption. In the case of errors, the third implementation (labeled 3 in the figure) 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.

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 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.

| Parameter | Meaning |
|---|---|
| $t_H$ | Time to compute hash value on state |
| $t_S$ | Time to compute signature on unchanged data |
| $t_P$ | Time to switch between two processes |
| $t_T$ | Time to switch between two POSIX threads |
| $t_E$ | Time to switch between two emulated threads |
| $t$ | Time for useful work |

Table 1: Evaluation Parameters. All parameters may be expressed as multiples of $t_H$,i.e. $t = \beta \cdot t_H$ and $t_X = \alpha_X \cdot t_H$, where $X = S, P, T, E$ and where $\beta, \alpha_X \geq 0$ are reals.

## 5 Evaluation

To evaluate our proposal we pursue two paths. First, we fix the performance of the processor and calculate how much faster one can switch in a virtual duplex system when moving from POSIX threads to emulated threads. Second, we fix the context switch frequency of a particular application and calculate the performance requirements of several types of implementation of duplex systems and virtual duplex systems. We do both calculations in a parameterized way first and apply some realistic values to these parameters afterwards. The parameters we use are given in Table 1.

### 5.1 Typical Parameter Values

Concerning the time to compute a signature, we assume that the methods used are of similar complexity to those used when computing a hash value of the state. Furthermore, we assume that the sizes of the state and the unchanged data are of the same order. Hence, we assume $t_S \approx t_H$, i.e.

$$\alpha_S = 1$$

Concerning $\alpha_P$, we start with bounding the absolute time for context switch, which is $t_P \geq 5\ \mu$sec even for a process with a small size [Cha98, Che01]. So even if $t_H$ should reach a value of e.g. $t_H = 12.5\ \mu$sec, which is considered large, we would still have

$$\alpha_P \geq 0.4$$

Normally however, one would assume $t_P \geq t_H$, i.e. $\alpha_P \geq 1$, because computing the hash value consists of browsing over all memory containing part of the state and performing some arithmetic on it, while switching process context involves storing all memory contents relating to that process.

Concerning the context switch time for POSIX threads, we assume $t_T \leq t_P$, because switching between threads does not involve more operations than switching between processes. In Linux, we have $t_T = t_P$, because in Linux each POSIX thread is implemented as a process of its own [Che01]. Under Solaris, a value reported is $t_P \approx 4 \cdot t_T$ [LB96]. Hence, we assume

$$\alpha_T = \alpha_P$$

or

$$\alpha_T = 0.25 \cdot \alpha_P$$

Regarding our own implementation of threads, the overhead is so small compared to the times previously discussed, that we dare to assume $t_E \ll t_H$ and thus assume

$$\alpha_E = 0$$

The parameter $\beta$ models the relation between useful work ($t$) and overhead ($t_H$) in a duplex system. A typical value for $\beta$ is 10, which means a 10% overhead.

## 5.2 Fixed Processor

Let us assume a processor running a virtual duplex system with POSIX threads, where a context switch occurs after a thread has been executed for a time $t$. The time to compare states and switch contexts is

$$t_1 = t_H + t_S + t_T$$

When we employ a virtual duplex system with emulated threads instead, this value changes to

$$t_2 = t_H + t_S + t_E$$

The former system can switch once in the time $t + t_1$, while the latter system can switch once in the time $t + t_2$. Hence, in the time $t + t_1$ the latter system can switch $A$ times, where the acceleration quotient $A$ is

$$\begin{aligned} A &= \frac{t + t_1}{t + t_2} \\ &= \frac{t + t_H + t_S + t_T}{t + t_H + t_S + t_E} \\ &= \frac{\beta + 1 + \alpha_S + \alpha_T}{\beta + 1 + \alpha_S + \alpha_E} \end{aligned} \tag{1}$$

If we assume that $t_S \approx t_H$, i.e. that $\alpha_S = 1$, that $t_E$ is close to 0, i.e. that $\alpha_E = 0$, and that $\beta = 10$, then Equation (1) simplifies to

$$A = 1 + \frac{\alpha_T}{12} \tag{2}$$

For values $\alpha_T \geq 3$, there is at least a 25% gain. Even when one assumes a POSIX thread system whose context switch is quite fast compared to the state comparison, or put the other way round, a state comparison that takes much longer then a context switch, there is still a noticeable gain of 3.3%, as for $\alpha_T = 0.4$ we obtain $A = 1.0\overline{3}$.

### 5.3 Fixed Context-Switch Frequency

To perform this evaluation, we start with a duplex system consisting of 2 processors. The overhead here consists of computing a hash value of the state. To get a valid starting point, we have to relate the overhead with the amount of work done between state checks. Therefore we define that in the duplex system, a state check taking time $t_H$ shall happen after time $t = \beta t_H$ of useful work.

We will express the performance of all other processors used in the sequel relatively to the processors used here, i.e. the processors used here have performance $P_1 = 1$ and perform a piece of useful work and a state check in time

$$T_1 = t + t_H = (\beta + 1) \cdot t_H$$

A virtual duplex system based on processes needs to perform, with each of two processes, the useful work, a state check and a context switch. This takes time

$$T_2 = 2 \cdot (t + t_H + t_P) = 2 \cdot (\beta + 1 + \alpha_P) \cdot t_H$$

In order to perform this work in time $T_1$, the processor needs a relative performance of

$$P_2 = \frac{T_2}{T_1} = 2 + \frac{2\alpha_P}{\beta + 1}$$

Note that we have made the implicit assumption that memory bandwidth scales with processor performance, or that at least memory bandwidth is not the limiting performance factor.

For a virtual duplex system based on POSIX threads, the overhead increases by $t_S$ for each thread, and the context-switch time changes from $t_P$ to $t_T$, leading to

$$\begin{aligned} T_3 &= 2 \cdot (t + t_H + t_S + t_T) = 2 \cdot (\beta + 1 + \alpha_S + \alpha_T) \cdot t_H \\ P_3 &= \frac{T_3}{T_1} = 2 + \frac{2(\alpha_T + \alpha_S)}{\beta + 1} \end{aligned}$$

For a virtual duplex system based on our threads, the context-switch time changes from $t_T$ to $t_E$, leading to

$$\begin{aligned} T_4 &= 2 \cdot (t + t_H + t_S + t_E) = 2 \cdot (\beta + 1 + \alpha_S + \alpha_E) \cdot t_H \\ P_4 &= \frac{T_4}{T_1} = 2 + \frac{2(\alpha_E + \alpha_S)}{\beta + 1} \end{aligned}$$

We already see, that under LINUX, where $t_T = t_P$, an implementation based on POSIX threads will be slower than an implementation based on processes! When we use typical values $\beta = 10$, $\alpha_S = 1$, $\alpha_E = 0$, and $\alpha_P = 4 \cdot \alpha_T$, we get

$$\begin{aligned} P_2 &= 2 + \frac{8\alpha_T}{11} \\ P_3 &= 2 + \frac{2\alpha_T + 2}{11} \\ P_4 &= 2 + \frac{2}{11} \end{aligned}$$

Hence, a virtual duplex system based on our threads needs a processor only about 2.2 as fast as the processor required for a duplex system. However, there we would need two of those processors! For a virtual duplex system based on POSIX threads, for $\alpha_T \leq 5$, we have $P_3 \leq 3$. Even this may be an acceptable choice.

The gap between duplex systems and virtual duplex system narrows if we also take into account the performance in case of an error. Assuming that state is saved after each comparison, the presented solutions lose a performance factor 2 during error recovery time in case of a duplex system, and 1.5 in case of a virtual duplex system. This holds because in both systems a third instance has to be run from the last checkpoint on to get a valid state again, before the normal work can be continued.

In this case, $P4/P1$ reduces by a factor of $1.5/2 = 0.75$ to about 1.64, i.e. we need one processor which is less than twice as fast as the processor in a duplex system, where two processors are needed. For a virtual duplex system based on POSIX threads, we obtain $P3/P1 \leq 2$ for $\alpha_T \leq 8/3$. Hence, for thread switch times that are not too large compared to state check times, also these virtual duplex systems require one processor less than twice as fast as a processor in a duplex system.

In general, if we allow the performance to shrink by a factor of $s$ during recovery, where $1 \leq s \leq 2$, the performance required by a duplex system increases by a factor of $2/s$, and the performance required by virtual duplex system increases by a factor of $1.5/s$ (for $s \leq 1.5$). Hence, also in this case the performance ratio reduces by a factor of 0.75 for $s \leq 1.5$, leading to similar results as above.

## 6 Conclusion

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. In an increasing number of applications, the ability to detect and/or tolerate transient and permanent faults is important. 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. As the overhead associated with these threads is still too large for the intended applications, we used emulated multithreading instead of traditional POSIX threads. The corresponding faster context switch time allows the frequent usage of signature checks in order to detect faults.

We have presented a qualitative and quantitative evaluation of virtual duplex systems based on threads. The evaluation indicates that the overhead of emulated multithreading is small enough to allow for fast fault detection and recovery.

## References

[AL94] 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.

[But97] Butenhof, D. R.: Programming with POSIX Threads. Addison-Wesley, 1997.

[Cha98] Chabrillange, P.: Context Switch Time with Threads. http://www.geocrawler.com/archives/3/35/1998/4/50/98906, April 1998.

[Che01] Chelf, B.: The Fibers of Threads. *Linux Magazine*, May 2001.

[EHN90] 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, pp. 362–367, Sofia, Verlag der Bulgarischen Akademie der Wissenschaften, 1990.

[Gra02] Grävinghoff, A.: On the Realization of Fine-Grained Multithreading in Software. PhD thesis, Universität Hagen, Germany, 2002.

[GK98] Grävinghoff, A.; Keller, J.: Fine-Grained Multithreading on the Cray T3E. High-Performance Computing in Science and Engineering, LNCS, pp. 447–456 , Berlin, Springer Verlag, 2000.

[LB96] Lai, K.; Baker, M.: A Performance Comparison of UNIX Operating Systems on the Pentium. Proceedings of the USENIX Annual Technical Conference, pp. 265–278, 1996.

[Lip91] Lipton, R. J.: New Directions in Testing. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 2, Providence, American Mathematical Society, 1991.

[Lov94] 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.

[Lov95] 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.

[Lov96a] 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

[Lov96b] Lovrić, T.: Fehlererkennung durch systematische Diversität in entwurfsdiversitären zeitredundanten Rechensystemen und ihre Bewertung mittels Fehlerinjection. PhD thesis, Universität Essen, Germany, 1996.

[PF83] 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.

[PLP98] 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.

[RM78] 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.

[She78] 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.

[SM91] 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.