# Virtual Duplex Systems with Forward Error Correction on Simultaneous Multithreaded Processors

Jörg Keller
FernUniversität Hagen
LG Technische Informatik II
Postfach 940
58084 Hagen, Germany
joerg.keller@fernuni-hagen.de

Peter Sobe
Universität zu Lübeck
Institut für Technische Informatik
Ratzeburger Allee 160, Haus 33
23538 Lübeck, Germany
sobe@iti.uni-luebeck.de

## Abstract

Virtual Duplex Systems provide detection of and recovery from transient as well as most permanent hardware faults. In contrast to duplex systems, they do so by providing temporal instead of structural redundancy: one time-shared instead of two processors. Previous studies on virtual duplex systems have focussed on either improving fault coverage or reducing overhead. We build upon this work and investigate the positive influence of an underlying processor hardware that supports multiple threads in hardware. Such processor architectures are just entering the market. Our findings are that those processors allow faster fault detection and recovery than conventional processors of the same speed. Alternatively, a multithreaded processor can be run at a lower frequency to provide the same detection and recovery rate as before.

## 1 Introduction

Duplication of processing activity is a proper technique to detect faults of the underlying hardware. If software diversity is employed, additionally design faults of the running software can be covered. A particular form of such a duplication is a virtual duplex system, where the duplicity is achieved by temporal redundancy. Virtual duplex systems (VDS) were introduced in [EHN90] and have been in the focus of research from then.

We assume a virtual duplex system (VDS) as e.g. defined in [GK00]. It consists of three versions of a software with identical functionalities. The versions show both design diversity and systematic diversity. This allows to cover transient as well as most permanent faults [Lov96]. A particular feature of our approach is the use of threads for the versions. In order to reduce the overhead of multithreading on a conventional microprocessor an emulated multithreading [Grä02] is used. In contrast, for a hyperthreaded microprocessor we employ a thread model that is supported by the operating system (e.g. POSIX threads) in order to process the two versions according to the hyperthreading abilities of the processor.

We target applications that require detection of and recovery from faults, especially transient faults, while shutting down to a fail-safe state and repair is not an option. Examples are applications in space (albeit not for mission-critical subsystems) and transportation. In the latter field, virtual duplex systems (based on conventional processors and on processes) are in commercial use for subways, e.g. in Copenhagen.

The remainder of this paper is organized as follows. Section 2 describes virtual duplex systems on conventional, and multithreaded processors, respectively, and compares them. Section 3 presents improvements on multithreaded processors, when taking into account knowledge about which version is faulty. Section 4 presents related work and Section 5 summarizes.

## 2 Virtual Duplex System Implementation

### 2.1 VDS on a Single Processor

If we employ a conventional processor, the VDS software is executed on a single processor in the following way (see Figure 1(a)). Versions 1 and 2 proceed alternately in rounds. After both versions have completed a round, the states of both versions are compared, and only in the case of identity, processing proceeds. The proceeding versions can be imagined as two threads scheduled round robin, with the context switched when they reach the end of a round. We assume that processing of a round for each version always takes time $t$, i.e. a complete round will take time

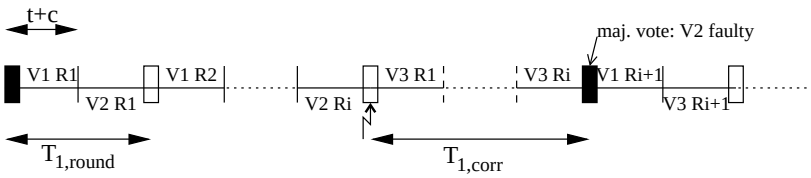
$$T_{1,round} = 2 \cdot (t + c) + t' \,, \tag{1}$$

where $t' \ll t$ is the time to compare the states, and $c \ll t$ is the time for a context switch.

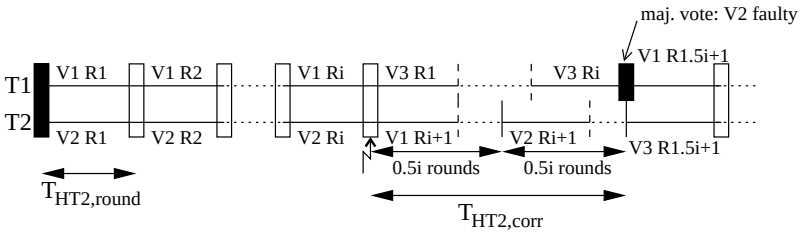
After every $s$ rounds, the state is saved in the form of a checkpoint. Now, if two differing states are detected at the end of round $i$ after the last checkpoint, where $1 \leq i \leq s$, then version 3 is started with the state from that checkpoint and executed for $i$ rounds. Then a majority vote over three available states allows to distinguish the faulty state, and proceed with the two versions that have correct states. Correction thus takes time

$$T_{1,corr} = i \cdot t + 2 \cdot t' \,. \tag{2}$$

Note that we here implicitly make the assumption that after the occurrence of a fault, no further fault will occur for the time of the error correction, as otherwise version 3 might get faulty as well.




(a) Virtual Duplex System on a Conventional Microprocessor




(b) Virtual Duplex System on a Multithreaded Processor

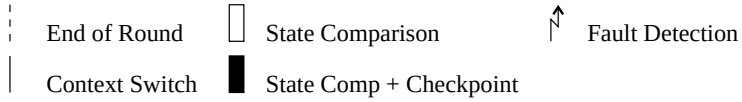



Figure 1: Execution models of a virtual duplex system on different processor architectures

### 2.2 VDS on a Hyperthreaded Processor

Now, if we replace our conventional processor by a processor that supports simultaneous multithreading [TEL95] in hardware, such as the Intel Pentium 4 with Hyperthreading [Int02], we can execute two threads, each containing a particular version in parallel[1] (see Figure 1(b)). Note that in a simultaneous multithreaded processor, the threads have register files and schedulers of their own, but share all other resources of the processor, especially the function units. As in any superscalar processor, the choice of the function unit is transparent to the user, i.e. a faulty ALU cannot be switched off.

Because of the improved processor utilization and the absence of a context switch one round will now take only

$$T_{HT2,round} = 2 \cdot \alpha \cdot t + t' \,, \tag{3}$$

with $\frac{1}{2} < \alpha < 1$. In the optimal case $\alpha = 0.5$, the two threads completely run in parallel. In the worst case $\alpha = 1$, the threads are — apart from the context switch — as slow as on the conventional processor. A value of $\alpha = 0.65$ has been reported for common applications [Wit02].

This means, that in the fault-free case, we have become faster by a factor

$$\begin{aligned} G_{round} &= \frac{T_{1,round}}{T_{HT2,round}} \\ &= \frac{2 \cdot (t+c) + t'}{2 \cdot \alpha \cdot t + t'} \\ &\approx \frac{1}{\alpha} \end{aligned} \tag{4}$$

if $c, t' \ll t$.

Now, after detection of a fault, we could in principle proceed as on a conventional processor. Then however, we would not gain any time (see footnote 1). Therefore, we employ a slightly more complex scheme: While version 3 is executed for $i$ rounds on the first thread, we execute $\lfloor i/2 \rfloor$ further rounds of versions 1 and 2 on the second thread, without state comparisons (as shown in Figure 1(b)). (We only have to make one context switch because of this!) Then we make state comparisons between version 3 and versions 1 and 2, respectively, to decide which of the latter versions was faulty.

The correction will take time[2]

$$T_{HT2,corr} = 2 \cdot i \cdot \alpha \cdot t + 2 \cdot t' \,. \tag{5}$$

Then, we copy the state of the fault-free version of versions 1 and 2 to version 3 while saving the next checkpoint. This means, that during correction we have also proceeded for about $i/2$ more rounds. Copying the state should not pose any problems because threads' memory regions are not protected against each other.

### 2.3 Comparison

Now we have to check in which cases $T_{HT2,corr} < T_{1,corr} + (i/2) \cdot T_{1,round}$, because if that equation holds, we have gained also in the case of a fault. By substitution of $T_{HT2,corr}$, $T_{1,corr}$ and $T_{1,round}$ by the full expressions we obtain

$$2 \cdot i \cdot \alpha \cdot t + 2 \cdot t' < i \cdot t + 2 \cdot t' + (i/2) \cdot (2 \cdot (t+c) + t')$$

[1] The processor considered supports two threads in hardware. If only one thread is active, the processor behaves like a conventional processor.

[2] To be exact, we would have to write $\max(t', c)$ instead of $t'$.

which can be reduced to

$$2 \cdot i \cdot \alpha \cdot t < i \cdot t + i \cdot (t + c) + \frac{i}{2} \cdot t' \; .$$

Over the full range of $\alpha$, the above relation is always true. This means that a multithreaded processor in all cases supports a faster fault localization. The gain is

$$\begin{aligned} G_{corr} &= \frac{T_{1,corr} + (i/2) \cdot T_{1,round}}{T_{HT2,corr}} \\ &= \frac{i \cdot t + 2 \cdot t' + (i/2) \cdot (2 \cdot (t + c) + t')}{2 \cdot i \cdot \alpha \cdot t} \\ &= \frac{2 \cdot i \cdot t + (2 + \frac{i}{2}) \cdot t' + i \cdot c}{2 \cdot i \cdot \alpha \cdot t} \\ &\approx \frac{1}{\alpha} \end{aligned} \quad (6)$$

if $c, t' \ll t$.

Note that our assumption of no further fault is applied here to versions 1 and 2 as well during correction time. However, this should not harm generality, because if a hardware fault only showed as a transient fault in one version before its detection, it is to be assumed that systematic diversity keeps the other version fault-free for some more time.

## 3 Using Knowledge about Faults

If we have evidence that a particular version, e.g. version 2, is most likely to be the faulty one[3], we could execute $i$ rounds of the other version, in our example version 1, in parallel to executing $i$ rounds of version 3. If we guessed correctly, we have made much more progress. Otherwise, we pay a penalty.

### 3.1 Correct selection of the fault free version

Here we have to check for $T_{HT2,corr} < T_{1,corr} + i \cdot T_{1,round}$. Obviously, the condition is always true. The gain is obtained as

$$\begin{aligned} G^{hit}_{corr} &= \frac{T_{1,corr} + i \cdot T_{1,round}}{T_{HT2,corr}} \\ &= \frac{i \cdot t + 2 \cdot t' + i \cdot (2 \cdot (t + c) + t')}{2 \cdot i \cdot \alpha \cdot t + 2 \cdot t'} \\ &= \frac{3 \cdot i \cdot t + (2 + i) \cdot t' + 2 \cdot i \cdot c}{2 \cdot i \cdot \alpha \cdot t + 2 \cdot t'} \\ &\approx \frac{3}{2 \cdot \alpha} \; . \end{aligned} \quad (7)$$

### 3.2 Incorrect selection of the fault free version

The criterion changes to $T_{HT2,corr} < T_{1,corr}$. In the best case, a hyperthreaded system performs equally to a VDS system on a single processor ($\alpha = \frac{1}{2}$). The loss is

$$\begin{aligned} L^{miss}_{corr} &= \frac{T_{1,corr}}{T_{HT2,corr}} \\ &= \frac{i \cdot t + 2 \cdot t'}{2 \cdot i \cdot \alpha \cdot t + 2 \cdot t'} \\ &\approx \frac{1}{2\alpha} \; . \end{aligned} \quad (8)$$

In the best case, the hyperthreaded processor loses nothing against the conventional processor, in the worst case it loses a factor of two.

### 3.3 Expected Gain

If $p$ is the probability of a correct guess of the faulty version, then the gain is

$$\begin{aligned} G'_{corr} &= p \cdot G^{hit}_{corr} + (1 - p) \cdot L^{miss}_{corr} \\ &\approx \frac{2p + 1}{2\alpha} \; . \end{aligned} \quad (9)$$

We first note that

$$G'_{corr} = \frac{2p + 1}{2\alpha} \geq \frac{1}{\alpha}$$

if $p \geq 0.5$. Notice that $p = 0.5$ corresponds to a random guess. Hence, if we do not make intentionally false guesses, this improvement will on average perform better in the case of a fault than the previous one. We also see that for $p \geq \alpha - 0.5$, the gain is at least one. In the best case $\alpha = 0.5$, we always gain no matter how bad our guesses are. Even in the worst case $\alpha = 1$, we gain for $p \geq 0.5$. Notice again that $p = 0.5$ corresponds to a random guess. For $p = 0.75$, the gain is $1.25/\alpha$. Thus this schema on average is 25% faster than the previous one.

For reasons of fairness, we note that in the conventional VDS, we may stretch the assumption of no further fault to the point that after the majority vote, we execute version 3 for another $i$ rounds without context switch, and copy its state to the other fault-free version while saving the next checkpoint.

However, replacing $T_{1,round}$ by $t$ in equation (7) does not change much as we assumed so far that $c, t' \ll t$. Even when we put in exact figures, the change will be not more than a few percent at the best.

## 4 Related Work

Much insight related to conventional process duplication can be used for VDS as well. For instance the effect of varied check intervals and checkpoint periods to reliability has been studied in [ZB97]. Shorter test intervals improve reliability, because the likeliness of two processes affected by a fault is decreased. In an environment of expensive stable storage access and inexpensive information transfer among processes it is advised to compare states more often than saving checkpoints. As proposed, VDS follows that way by using short rounds and longer checkpoint intervals.

For recovery of duplex based systems — also appropriate for VDS — the following ways have been described:

**Rollback**- both processes/versions are set back to the state of the last checkpoint and the processing interval is retried. If then two equal states are reached, the processing is continued.

[3] E.g. in the case of a crash fault.

**Stop and retry**- if a comparison mismatches, both processes/variants are stopped while a third process/variant computes a third status for the mismatching round. Then the fault free processes are identified by a 2-out-of-3 decision and their state is used to continue duplex processing. An algorithm for process duplication using stop-and-retry recovery in computer networks, especially with focus on message transfer, has been given in [VP92].

**Roll-forward checkpoint schemes** - can be seen as an extension of stop-and-retry exploiting parallelism of the system. While the third variant is executed, processes/variants 1 and 2 continue processing on the remaining hardware. A near variant has been described in [PV94, PSV94].

As proposed in this paper, VDS on a hyperthreaded processor follows the concept of roll-forward checkpointing schemes. Until now, recovery schemes in general have been applied exclusively to real duplex systems. In the context of VDS only fault detection and a fail safe stopping is common practice. Continuation of processing after recovery can be seen as a new aspect using VDS that allows to continue processing at least in occurrence of transient faults.

## 5 Summary

A virtual duplex system is an appropriate technique to tolerate faults by temporal redundancy. Two processes/threads are executed in short time slices one after another. Using modern microprocessors with the ability to execute two threads in parallel in a super-scalar way, one can shift time redundancy to spatial redundancy. We have evaluated the gain by using a hyperthreaded processor architecture for such a VDS and shown that even in the case that a processor does not exhibit the double performance by 2-way hyperthreading, we get a gain for the normal processing and the error correction phases too.

Alternatively, if we are already satisfied with the VDS performance, we could employ a multithreaded processor with a clock frequency reduced by a factor of at least $1/\alpha$, assuming that performance scales linear with clock frequency. This would account for lower cost, lower power consumption and lower heat dissipation.

So far, there is no experimental data for guessing $p$. However, assuming that the prediction correctness is close to 1 for crash faults, and $0.5$ (random choice) for other faults, $p$ is clearly larger than $0.5$, where the exact value depends on the fraction of crash faults among all faults. Moreover, if we consider space missions where the duration is long and the frequency of transient faults is much higher than on earth, there may even be the possibility to gather a fault history and to predict the faulty version in a manner similar to branch prediction in microprocessors [FKS04].

## References


[EHN90] K. Echtle, B. Hinz, and T. Nikolov. On Hardware Fault Diagnosis by Diverse Software. In *Proceedings of the 13th International Conference on Fault-Tolerant Systems and Diagnostics*, pages 362–367. Bulgarian Academy of Science, Sofia, 1990.

[FKS04] B. Fechner, J. Keller, and P. Sobe. Performance estimation of virtual duplex systems on simultaneous multithreaded processors. Informatik-Bericht 307, FernUniversität Hagen, January 2004.

[GK00] A. Grävinghoff and J. Keller. Fine-Grained Multithreading on the CRAY T3E. In *High-Performance Computing in Science and Engineering, LNCS*, pages 447–456. Springer Verlag, 2000.

[Grä02] A. Grävinghoff. *On the Realization of Fine-Grained Multithreading in Software. PhD thesis.* University Hagen, Germany, 2002.

[Int02] Intel. Hyper-threading technology on the intel xeon processor family of servers. white paper. 2002.

[Lov96] T. Lovrić. *Fault Detection by Systematic Diversity in Time-Redundant Computing Systems with Design Diversity, and Their Evaluation by Fault Injection (in German)*. PhD thesis, Univ. Essen, Germany, 1996.

[PSV94] D.K. Pradhan, D.D. Sharma, and N.H. Vaidya. Roll–Forward Checkpointing Schemes. In M. Banatre and P.A. Lee, editors, *Hardware and Softwarearchitectures for Fault–Tolerance*, number 774 in Lecture Notes in Computer Science. Springer, 1994.

[PV94] D.K. Pradhan and N.H. Vaidya. Roll–Forward Checkpointing Scheme: A Novel Fault–Tolerant Architecture. *IEEE Transactions on Computers*, 43(10), October 1994.

[TEL95] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–402, 1995.

[VP92] N.H. Vaidya and D.K. Pradhan. A Fault-Tolerance Scheme for a System of Duplicated Communicating Processes. In *Proceedings of the IEEE Workshop on Fault Tolerant Parallel and Distributed Systems*, pages 98–104, July 1992.

[Wit02] M. Withopf. Virtual tandem: Hyperthreading in the new pentium 4 with 3 GHz (in German). *c't*, 24:120ff, 2002.

[ZB97] A. Ziv and J. Bruck. Performance Optimization of Checkpointing Schemes with Task Duplication. *IEEE Transactions on Computers*, 46(12):1381–1386, December 1997.