Performance Estimation of Virtual Duplex Systems on Simultaneous Multithreaded Processors

Bernhard Fechner and Jörg Keller
FernUniversität Hagen
LG Technische Informatik II
Postfach 940
58084 Hagen, Germany
{bernhard.fechner|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 transient as well as most permanent hardware faults by executing two versions of a program on a single processor in a time-shared manner. 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 architecture that supports parallelism in the form of multiple threads in hardware. Such processor architectures are just entering the market, with a die area only slightly larger than that of a conventional processor. A performance prediction shows that those processors allow faster fault detection than conventional processors of the same speed. Moreover, the parallelism can be utilized for a recovery that extends the concept of virtual duplex systems. Additionally, we present a technique that further increases the above mentioned gain by using prediction of the faulty version in a manner similar to branch prediction.

1 Introduction

Duplication of processing activity is a proper technique to detect faults of the underlying hardware. When software diversity is employed, design faults of the running software can be covered as well. 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 [1] and have been in the focus of research from then on. Originally, the temporal redundancy is obtained by executing the same program with the same input data twice. One may vary the way of duplication. One option consists in running entire programs twice, whereby another option is to execute the duplicated processes in short rounds and switch between them. The switching introduces extra overhead, but can be used to compare the intermediate results more often in order to detect a fault soon after the first effects. Virtual duplex systems can be extended from only detecting faults to tolerating faults, as we will show in this article.

Virtual duplex systems are not likely to be applied when the risk is very high, because they cannot tolerate all permanent hardware faults. Yet, in the medium range they provide a cost advantage over duplex systems because of reduced hardware requirements. The ability to not only detect but also tolerate faults enables their use in situations where a speedy repair cannot be supplied after a fault. An example are soft mission critical systems, e.g. computers that serve scientific experiments on space missions. Here, a single experiment is not mission critical, its failure however still is expensive. In outer space transient faults are much more frequent due to radiation, and repair is impossible for obvious reasons. Also, due to increasingly smaller feature sizes, it is to be expected that transient faults due to radiation will occur much more frequently on earth missions, both in memories and in logic [10]. Virtual duplex systems are already in commercial use in transportation environments, e.g. in the Copenhagen subway.

We assume a virtual duplex system (VDS) as e.g. defined in [2]. It consists of three versions of a software with identical functionalities. Two versions are used to detect transient faults, the third will be needed for detection of permanent faults and for recovery. The versions show both design diversity and systematic diversity to be able to recover from transient as well as from many permanent hardware faults [6]. The diverse versions can be generated automatically [4].

Our goal is to investigate the advantages of using a microprocessor that supports multiple threads in hardware. So-called simultaneous multithreaded processors are just

entering the market, e.g. the new Pentium 4. Intel calls this microprocessor hyperthreaded, which is synonymous for 2-way multithreaded. These processors are interesting because they promise performance gains (runtime reduction up to 35% has been reported) at a moderate increase in cost (the die area increases by only 5%) [13]. For a hyperthreaded microprocessor we employ a processing model that is supported by the operating system, i.e. we assume that it maps user processes onto the hardware threads of the processor in the same manner as on a two-processor machine. When talking about threads in the sequel, we mean hardware threads carrying a user process, unless otherwise stated.

The remainder of this paper is organized as follows. Section 2 specifies the system and fault model and discusses related work. In Section 3 virtual duplex systems on conventional and multithreaded processors, respectively, are described and compared. Further improvements on multithreaded processors, when taking into account knowledge about which version is faulty are presented in Section 4. Section 5 summarizes our work.

2 Preliminaries

2.1 System and fault model

For both architectures that are compared in this paper — a conventional microprocessor and a simultaneous multithreaded processor — we assume the same basic system and fault models. The system models are differentiated in a few points.

System model:

- Versions are functions that can be executed as processes. A version can be run for a specified number of rounds. Here a well defined portion of process activity is executed and then the function returns. Later, the version can be continued from the point.

- Classical virtual duplex systems either use processes or (user) threads. In the latter case, each version/function operates on a private subspace of a common address space while no fault occurs. A fault leading to accesses in a different versions subspace may lead to data corruption of both versions. The detection of this case can be covered by applying error detecting codes for data in the memory. For a virtual duplex system on a simultaneous multithreaded processor, we restrict to separate address spaces for all versions, which are protected against each other by operating system and hardware means. An access to the data of another version then leads to an access violation which is signaled as a fault but leaves the other version's data unchanged.

- Recovery is enabled by saving state to a disk from time to time (checkpointing).

Fault model:

- Transient and permanent faults are assumed.

- A constraint applies for both types of faults: a fault may not corrupt states/output of any two versions in the same way. For transient faults this is very likely, because they can be modeled as bit flips in registers, and as such only directly affect one version. The exclusion of consequences for other versions has already been discussed above. For permanent faults, diversity is used to employ the hardware in different ways and to make it unlikely that a single fault shows the same effect on two versions.

- A fault is able to stop a version and also to stop the entire processor including all versions. In the latter case, recovery is only possible by rollback (see next subsection).

2.2 Related work

Much insight related to conventional process duplication can be used for VDS as well. For instance the effects of varied check intervals and checkpoint periods on reliability have been studied in [14]. A main result from that study is that shortening test intervals improves reliability, because the likeliness of two processes affected by a fault is decreased. Thus, it is advised to test states more often than saving checkpoints. In common environments, additionally stable storage access for checkpointing is relatively expensive that is a reason for relative long checkpoint intervals. As proposed, VDS follows that way by using short rounds after that the states of the versions are compared and longer checkpoint intervals.

Using a multithreaded processor to achieve fault detection has been investigated by Reihardt and Mukherjee [9]. They run two identical versions, and they work in a cycle-by-cycle lockstep, to reduce detection time to a minimum. The price they pay is a loss in performance and extra hardware for state comparison after each cycle, and extra methods (also supported in hardware) to detect multi-cycle faults.

For recovery of duplex based systems, generally one may embark on the following strategies:
Rollback recovery- both processes/versions are set back to the state of the last checkpoint and the processing interval is retried. If two equal states are reached afterwards, the processing is continued.
Stop and retry recovery- if a state comparison mismatches, both processes/variants are stopped until a third process/variant computes a third status for the mismatching

round. Then a 2-out-of-3 decision is made to identify the fault free version that 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 [12].

Roll-forward checkpoint schemes- can be seen as an extension of stop-and-retry recovery. Recovery is designed to exploit parallelism of the system. While the third variant is executed in order to find the fault free state, simultaneously processes/variants 1 and 2 continue processing on the remaining hardware.

The VDS on a hyperthreaded processor will follow the concept of the roll-forward checkpointing scheme (a near variant has been described in [7, 8]).

3 Virtual duplex system implementation

3.1 VDS on a conventional 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 continues. The proceeding versions can be imagined as scheduled round robin, with the context switched when they reach the end of a round. We assume that the processing of a round for each version always takes time t, i.e. a complete round will take time

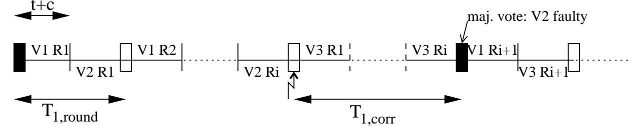
$$T_{1,round} = 2 \cdot (t + c) + t' \ , \qquad (1)$$

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

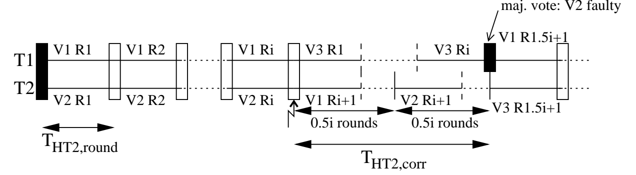
After every s rounds, the state is saved in the form of a checkpoint. Now, if two differing states are detected at the end of round i after the last checkpoint, where $1 \leq i \leq s$, then version 3 is started with the state from that checkpoint and executed for i rounds. Then a majority vote over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

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

Note that in the case of an additional fault during recovery, or in case of a permanent fault not covered by the diversity of the versions, we will have three different states, and no majority vote is possible. In this case, one has to resort to a rollback scheme, i.e. starting versions 1 and 2 again from the last checkpoint. However, we will not consider that situation in the sequel.



(a) Virtual Duplex System on a Conventional Microprocessor



(b) Virtual Duplex System on a Multithreaded Processor

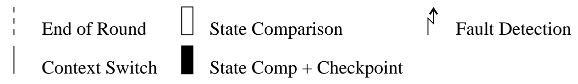


Figure 1. Execution models of a virtual duplex system on different processor architectures.

3.2 VDS on a hyperthreaded processor

Now, if we replace the conventional processor by a processor that supports simultaneous multithreading [11] in hardware, such as the Intel Pentium 4 with Hyperthreading [3], we can execute two threads, each containing a particular version in parallel[1] (see Figure 1(b)). Because of the improved processor utilization and the absence of a context switch one round will now take only time

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

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. In the Intel Pentium 4, $\alpha = 0.65$ is reported [13]. The extra hardware cost is quite small, merely a 5% increase in die size [13].

This means that in normal processing periods a speedup of G_{round} is obtained.

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

if $c, t' \ll t$.

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

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 the first thread executes version 3 for i rounds, the other thread is employed in a roll-forward scheme, i.e. used to proceed versions 1 and 2 beyond round i. We will consider both a deterministic and a probabilistic roll-forward scheme.

In order to be able to detect a fault during the roll-forward, we proceed with versions 1 and 2 starting from a common state. Here we have the possibility to choose from the states P and Q of both versions at the end of round i, respectively. However, these states are different, and we do not know which of these states is affected by the fault just detected. In the probabilistic scheme, we choose one of the states, and execute both versions for $i/2$ rounds[2] each, which needs about the same time as executing i rounds of version 3 in the first thread. Figure 1(b) depicts this scheme. If we chose the state of the fault-free version, our roll-forward is successful. If we chose the state of the faulty version, we did not gain anything by the roll-forward. Note that our choice can be random, so that the probability to choose the correct version is 0.5, or can try to use some prediction scheme which might increase this probability, see Section 4.

In the deterministic scheme, we start from both states, and roll-forward $i/4$ rounds of each version, starting from each state, so that in total we execute i rounds, which needs about the same time as the retry of version 3 in the first thread.

Note that the roll-forward may have to be shortened due to the checkpointing interval. If $i > 4s/5$ and $i > 2s/3$ in the deterministic and probabilistic schemes, respectively, we only roll forward until round s. Hence, when we intend to roll forward for x rounds, then we really do so by $\min(x, s-i)$ rounds. The latter case happens if $i > s-x$.

During the roll-forward, no state comparisons are performed. In the probabilistic scheme, we first execute $i/2$ rounds of version 2, and then switch to version 1, of which $i/2$ rounds are executed afterwards. In this way, only a single context switch is necessary. In the deterministic scheme, we first execute $i/4$ rounds of version 2 starting from state P (the state of version 1 after round i), then $i/4$ rounds of version 1 starting from state P, then $i/4$ rounds of version 1 starting from state Q (the state of version 2 after round i), and finally $i/4$ rounds of version 2 starting from state Q. In this way, only a single context switch is necessary.

After the roll-forward, the final states of the versions are compared. If those states are different, then an additional fault has been detected during roll-forward. Hence, the roll-forward has to be discarded. Figure 2 depicts the flow diagram of normal processing and fault recovery on a multithreaded processor with the probabilistic scheme. Figure 3 depicts the corresponding diagram for the deterministic scheme.

The recovery will take time[3]

$$T_{HT2,corr} = 2 \cdot i \cdot \alpha \cdot t + 2 \cdot t' \ , \quad (5)$$

assuming that the roll-forward in the second thread does not take longer than the retry in the first thread. To complete recovery in case of a successful roll-forward, the state of the fault-free version (version 1 or 2) is copied to version 3. So, version 3 is rolled forward to the fault-free version and forms a new VDS with the remaining fault free version.

3.3 Comparison

We will figure out how much faster our recovery including roll-forward is in relation to a VDS on a conventional processor. We treat the probabilistic and deterministic variants separately. For the deterministic variant, the gain is

$$\begin{aligned} G^{det}_{corr}(i) &= \frac{T_{1,corr} + \min(i/4, s-i) \cdot T_{1,round}}{T_{HT2,corr}} \\ &\approx \begin{cases} \frac{3}{4\alpha} : i \leq 4s/5 \\ \frac{2s-i}{2i\alpha} : i > 4s/5 \ . \end{cases} \end{aligned} \quad (6)$$

if $c, t' \ll t$.

We assume a fault to happen with equal probability in any round i, where $1 \leq i \leq s$. Hence the average gain is

$$\begin{aligned} \bar{G}^{det}_{corr} &= \frac{1}{s} \cdot \sum_{i=1}^{s} G^{det}_{corr}(i) \\ &\approx \frac{1 + 2\ln(5/4)}{2\alpha} \\ &\approx \frac{0.723}{\alpha} \ , \end{aligned} \quad (7)$$

if we use $\sum_{i=n+1}^{m} 1/i \approx \ln(m/n)$.

In the probabilistic case, we have to distinguish whether our prediction of the faulty case is correct or not. We will only state the expected gain here, because the computation will be explained in detail in the next section:

$$\bar{G}^{prob}_{corr} \approx \frac{1 + 2p\ln(3/2)}{2\alpha} \approx \frac{0.405p + 0.5}{\alpha} \ . \quad (8)$$

For $p = 0.5$, a random choice, both expressions (7) and (8) have approximately equal values, as one would expect. For $p > 0.5$, the probabilistic scheme provides a larger gain. The gain of the deterministic scheme is larger than one for $\alpha < 0.723$, i.e. a medium utilization of the processor suffices to gain.

---

[2]For simplicity, we do not consider the detail that $i/2$ may not be an integer.

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

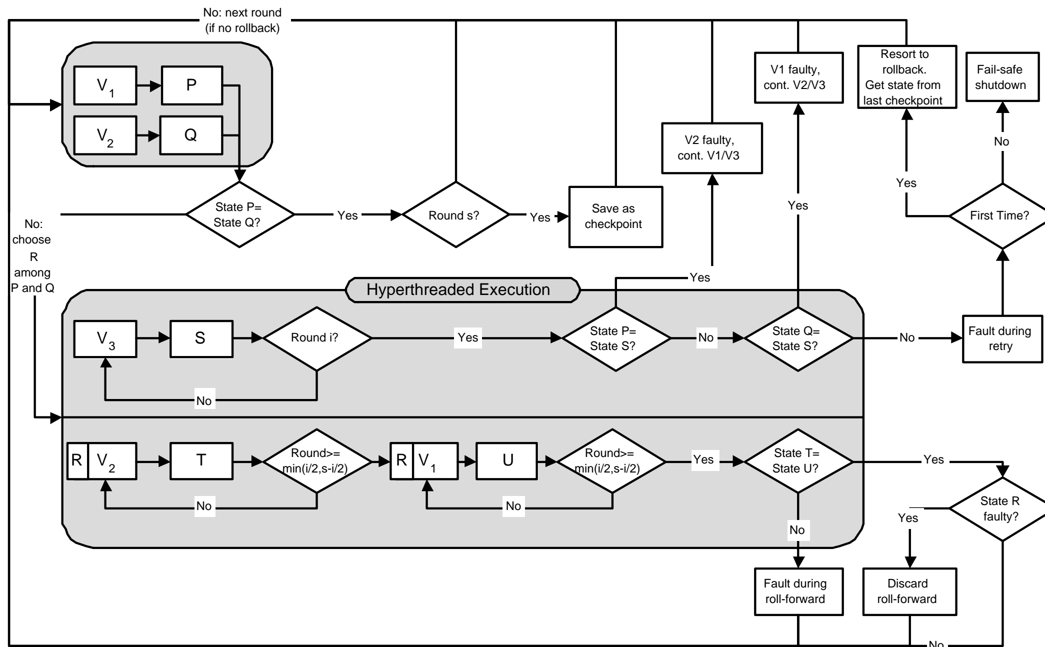


Figure 2. Flow chart of VDS on a multithreaded processor with probabilistic roll-forward.

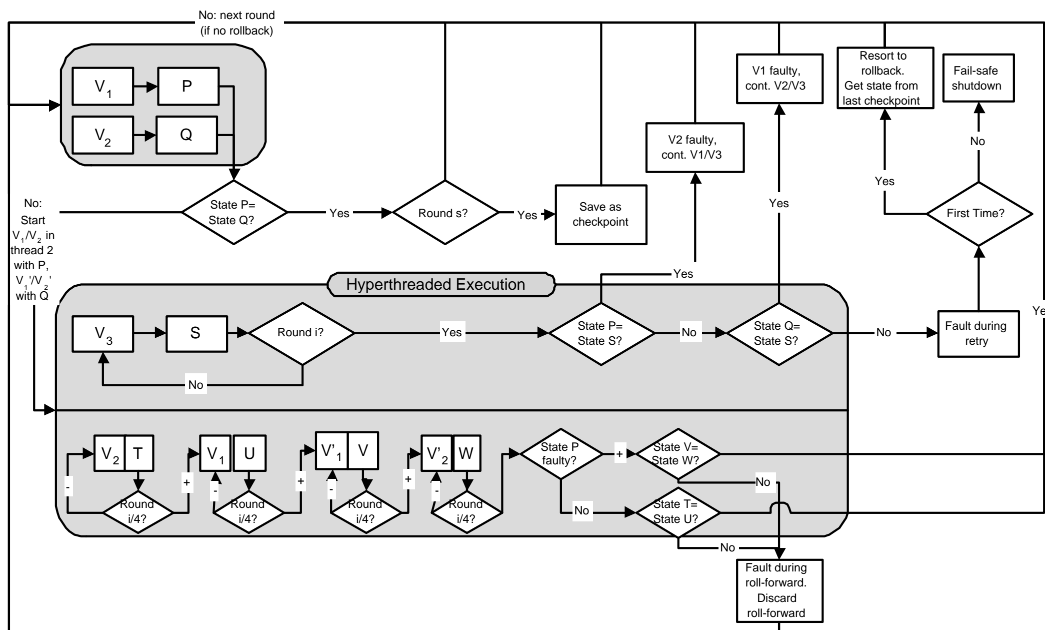


Figure 3. Flow chart of VDS on a multithreaded processor with deterministic roll-forward.

4 Using knowledge about faults

If we refrain from the detection of faults during roll-forward, we can simply execute i further rounds of one of the versions in the second thread while version 3 does the retry in the first thread. Here again, we have a probabilistic scheme as in general we do not know for sure which version is faulty. However, sometimes there is evidence that a particular version, is most likely to be the faulty one, e.g. in the case of a crash fault. If we guessed the faulty version correctly, we have made much more progress than in previous schemes. Otherwise, we pay a penalty. Note that we still have the restriction that the roll-forward will not continue beyond round s, because of the checkpointing interval. Hence, we roll forward for $\min(i, s-i)$ rounds. The latter occurs if $i > s/2$.

4.1 Correct prediction of the fault free version

If we predict and choose the fault-free version, then we indeed achieve a roll-forward of $\min(i, s-i)$ rounds during the retry. The gain we achieve against a VDS on a conventional processor is

$$G_{corr}^{hit}(i) = \frac{T_{1,corr} + \min(i, s-i) \cdot T_{1,round}}{T_{HT2,corr}} \quad (9)$$

$$= \begin{cases} \frac{3it+(2+i)t'+2ic}{2i\alpha t+2t'} & : i \leq \frac{s}{2} \\ \frac{(2s-i)t+(2+s-i)t'+2(s-i)c}{2i\alpha t+2t'} & : i > \frac{s}{2} \end{cases} \quad (10)$$

$$\approx \begin{cases} \frac{3}{2\cdot\alpha} & : i \leq \frac{s}{2} \\ \frac{2s/i-1}{2\cdot\alpha} & : i > \frac{s}{2} \end{cases} .$$

4.2 Incorrect prediction of the fault free version

Here, the roll-forward does not provide any benefit. In the best case, a hyperthreaded system performs equally to a VDS system on a single processor ($\alpha = \frac{1}{2}$). The loss is

$$L_{corr}^{miss}(i) = \frac{T_{1,corr}}{T_{HT2,corr}}$$

$$= \frac{i \cdot t + 2 \cdot t'}{2 \cdot i \cdot \alpha \cdot t + 2 \cdot t'} \quad (11)$$

$$\approx \frac{1}{2\alpha} .$$

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

4.3 Expected gain

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

$$G'_{corr}(i) = p \cdot G_{corr}^{hit}(i) + (1-p) \cdot L_{corr}^{miss}(i) \quad (12)$$

$$\approx \begin{cases} \frac{2p+1}{2\alpha} & : i \leq \frac{s}{2} \\ \frac{2p(s/i-1)+1}{2\alpha} & : i > \frac{2}{2} \end{cases} .$$

We assume a fault to occur at any round i with equal probability. Hence the expected gain is

$$\bar{G}'_{corr} = \frac{1}{s} \cdot \sum_{i=1}^{s} G'_{corr}(i) \quad (13)$$

$$\approx \frac{1 + 2p\ln 2}{2\alpha}$$

if we use $\sum_{i=s/2+1}^{s} 1/i \approx \ln 2 \approx 0.693$.

We first note that

$$\bar{G}'_{corr} \approx \frac{0.5 + p\ln 2}{\alpha} > \bar{G}_{corr}^{prob} \geq \bar{G}_{corr}^{det}$$

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 ones.

We also see that for $p \geq (\alpha - 0.5)/\ln 2$, the gain is at least one. In the best case $\alpha = 0.5$, we always gain no matter how bad our guesses are. For random guesses ($p = 0.5$) we gain for $\alpha \leq (1 + \ln 2)/2 \approx 0.85$.

We present our results graphically in Figures 4 and 5 for $p = 0.5$ and $p = 1.0$, respectively. Here, $p = 0.5$ represents a worst case, as we do not expect any strategy to be worse than a random choice, and $p = 1.0$ represents a best case. For both figures, we set

$$c = t' = \beta \cdot t \quad (14)$$

to reduce the number of unknowns. We assume $0 \leq \beta \leq 1$, where $\beta = 0$ represents the case where overhead can be neglected, and $\beta = 1$ represents the (unrealistic) case where a context switch or a comparison take as much time as a round. For both figures, we assume $s = 20$. We obtain the figures not by using the approximated values that neglect c and t', but by using exact equations (10), (11), (12), (13), and (14).

Since the time for a context switch is much smaller than the time for a round, we may set $\beta = 0.1$. For a hyperthreaded processor like the Intel Pentium 4, we get $\alpha = 0.65$ as its reported gain is 35% [13]. The maximum gain for these values is obtained by calculating the limit for s going towards infinity:

$$G_{max} = \lim_{s\to\infty} \bar{G}'_{corr} = \frac{23 \cdot \ln 2}{13} \cdot p + \frac{10}{13} .$$

Note that beyond $s = 20$, $\bar{G}'_{corr}$ is already very close to the limit, independently of the values for α and β. Therefore, we chose $s = 20$ in the figures.

If we pessimistically set $p = 0.5$, we get an acceleration of $G_{max} \approx 1.38$ over the non-hyperthreaded version. Even

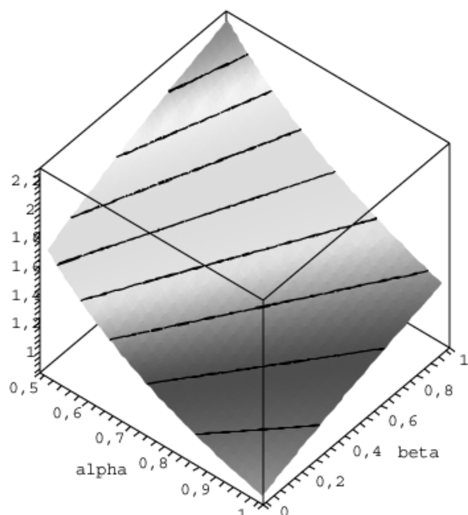


Figure 4. Gain $\bar{G}'_{corr}(\alpha, \beta)$ for $p = 0.5$.

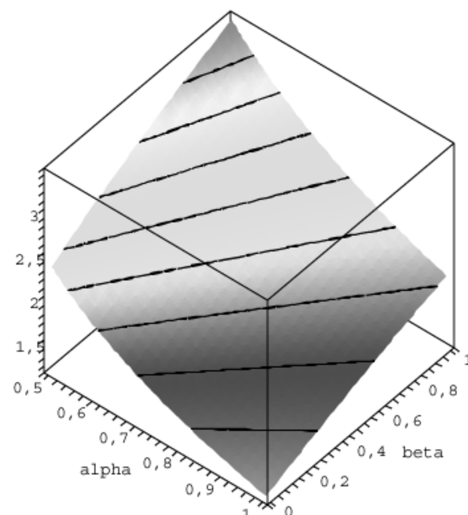


Figure 5. Gain $\bar{G}'_{corr}(\alpha, \beta)$ for $p = 1.0$.

if we apply the results from [5], where it was reported that multithreading improved execution time by less than 10 percent for most of the applications investigated, we still would not lose as $G_{max} \approx 1.0$. However, we have to note that the hardware employed there (Alewife multiprocessor with Sparcle processor, a modified Sparc) is not a simultaneous multithreaded processor. Threads were supported by using different parts of the register file, and context switches were executed when a thread was waiting for a remote memory access.

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 (9) 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.

5 Summary and outlook

A virtual duplex system is an appropriate technique to detect and 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 maximum possible performance by hyperthreading, we get a gain for the normal processing and the error correction phases.

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.

For a multithreaded processor supporting more than two threads in hardware, we are able to boost the variants with fault detection during roll-forward: in the probabilistic scheme we could execute versions 1 and 2 for i rounds each in two separate threads (needing 3 threads in total), in the deterministic scheme we could execute versions 1 and 2, starting from states P and Q, for i rounds each (needing 5 threads in total).

The prediction probability p could be further improved using techniques similar to branch prediction in microprocessors: we keep a history of faults. This may be useful in situations where the probability of transient faults due to radiation is high enough that several of them may occur. If a particular part of the hardware is more likely to be affected by faults of this kind due to process variations, this can be detected. While branch prediction takes a lot of hardware resources in microprocessors to take no more than one cycle, our fault prediction can be done in software as we are operating on much larger time scales. For the same reason, we may be able to apply more sophisticated algorithms.

References

[1] 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.

[2] 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.

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

[4] M. Jochim. Detecting processor hardware faults by means of automatically generated virtual duplex systems. In *Proceedings IEEE International Conference on Dependable Systems and Networks (DSN'02)*, pages 399–408, 2002.

[5] B.-H. Lim and R. Bianchini. Limits on the performance benefits of multithreading and prefetching. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '96)*, pages 37–46, May 1996.

[6] 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.

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

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

[9] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 25–36, 2000.

[10] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings IEEE International Conference on Dependable Systems and Networks (DSN'02)*, pages 389–398, 2002.

[11] 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.

[12] N. Vaidya and D. 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.

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

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