ANALYSIS OF SOFTWARE-BASED RECOVERY SCHEMES FOR SMT PROCESSORS

Lukas Beyer, Bernhard Fechner, Jörg Keller
Department of Computer Science
University of Hagen
58084 Hagen, Germany
email: {lukas.beyer |bernhard.fechner |joerg.keller} @fernuni-hagen.de

ABSTRACT
Today's microprocessors are prone to transient hardware faults caused by e.g. ionizing particles. The usual method to detect and correct such faults is to use duplex systems in software. Fault detection and correction can be accelerated by taking advantage of logical processors available since the introduction of commercial SMT systems, e.g. by performing a simultaneous retry and roll-forward on different logical processors. We derive four different recovery schemes ($\{probabilistic, deterministic\} \times \{pessimistic, optimistic\}$), each of which can be applied after an error has been detected. The recovery software is modular and requires only minor extensions to existing code to provide protection. The schemes are tailored to be executed on an SMT processor. Their execution times are measured under the influence of transient faults, injected at rates of 10^{-5}, $\frac{1}{4} \cdot 10^{-5}$ and 10^{-6}. Depending on fault rate, checkpoint distance and the probability to correctly guess correct versions, we make recommendations about which variant to choose. An important insight is that a high rate of successful guesses p is needed for the probabilistic schemes to provide significant advantage over the deterministic ones. When randomly choosing the version to roll-forward ($p = 0.5$), the optimistic deterministic variant is faster than the optimistic probabilistic one. With $p = 0.7$, the optimistic probabilistic variant begins to perform better than its deterministic counterpart. The comparison of pessimistic schemes yields similar results.

KEY WORDS
Virtual Duplex System, SMT, recovery, roll-forward, roll-back

1 Introduction

Transient faults in the form of single event upsets can arise from ionizing particle radiation. The high-energetic particles come from different sources, e.g. radioactive decay and cosmic radiation. They ionize while passing through an electronic circuit. Thus, the charge of the circuit [1] can be modified temporarily, introducing bit-flips in memory elements. Fault detection can be achieved by temporal or structural redundancy i.e. multiple processing units. In the past, fault tolerance was only attractive for mission-critical applications, since classical fault-tolerance mechanisms like triple modular redundancy (TMR) systems sacrifice the performance of several processors to locate and detect faults, including huge additional energy and spacial requirements.

As future processors will have smaller feature sizes, reduced voltage levels and higher on-chip transistor counts, the frequency of transient faults will increase. Hence, future computing systems will have to detect those faults and be able to recover quickly from them. Simultaneous multithreading (SMT) [4] is a technique that allows multiple instruction streams to execute in parallel by using different functional units of a superscalar microprocessor. SMT processors can be used for fault detection and recovery by running processes or threads on independent logical processors.

This paper contributes the following:

- It presents a performance analysis of four recovery schemes under the influence of different fault rates.
- It makes recommendations about which recovery scheme performs best under different parameter variations.

The paper is organized as follows: Section 2 presents related work. In Section 3, we describe the modeling of recovery schemes in software. The simulation methodology is presented in Section 4. Finally, Section 5 presents the simulation results followed by a conclusion in Section 6.

2 Previous and Related Work

To achieve fault tolerance, a computing system requires redundancy. In duplex systems, two versions of a program are executed on two processors, fed with the same input. To detect faults, the states of the versions are compared at regular time points. Virtual duplex systems [7] avoid the duplicated hardware resources associated with traditional duplex systems, requiring only a single processor. Instead of executing the versions in parallel, two software processes use used to execute the versions. The processes are scheduled in a round-robin fashion, thereby using temporal redundancy instead of structural redundancy. If both versions execute the same code, a permanent hardware fault in a virtual duplex system will affect both versions in the same way. Therefore the fault will not be detected. To support the detection of permanent faults by virtual duplex systems, systematic and design diversity can be applied. To achieve design diversity, the versions have to be developed by independent individuals or groups, following the same initial specification [3]. Systematic diversity techniques modify the code at assembler language level [5]. The goal is diversifying the use of hardware in order to minimize the probability of identical erroneous results. Reinhardt and Mukherjee [10] used an SMT processor to achieve fault detection by running two identical versions cycle-by-cycle lockstepped to reduce detection time to a minimum. The implied virtual duplex system follows a variant of the roll-forward checkpointing scheme described in [11] and [12]. The different recovery schemes in this work follow the classification and unified modeling introduced in [8]. Preceding theoretical results are provided in [9]. There it was forecast that running a virtual duplex system on an SMT processor would yield the same speedup over conventional processors as other applications in the fault-free case, and that average performance during roll-forward would be between 70 and 80% of that value, with a ranking between three of the four schemes presented in Section 3.2.

3 System Modeling

Our basic model relies on the general concept of an abstract state of a version and mapping functions from [2]. The fault model assumes transient faults in the datapath of the underlying processor. If the versions are developed independently by distinct programmers, they may well choose different representations for the states of the versions. For example, a graph may be represented by an adjacency list in one version and by an adjacency matrix in another. The concrete representation of a version's state is called *internal state*. For the recovery of versions and state comparison a common representation has to be used - the *abstract state*. It has to contain enough information to recover the state of any version. The implementation consists of two core components:

- The versions which perform the actual functionality of the application.
- A software component that is independent from the concrete application, termed *controller*.

The controller assigns versions to the available logical processors and compares the versions' states by computing hash values. Furthermore it monitors the results from the error detection and controls the selection of different recovery schemes. The controller binds the processes or threads that execute the versions to different logical processors of the underlying SMT processor. The execution of the versions is realized in rounds. In every round, a state transition function is invoked to calculate the next internal state.

Five functions reside within every version besides the application code itself:

- A *write* function for transforming the internal state of a version to an abstract state and to write it to a file or pipe.
- A *recover* function to restore the internal state of a version from an abstract state. It reads the abstract state from a specified file or pipe, transforms it to an internal state and returns the recovered state.
- A function for releasing the internal state. The function takes the internal state and frees occupied memory.
- A function calculating the hash value over the abstract state.
- A state transition function that takes the current internal state of the version and returns the next internal state. This function calls the application code. An upper limit can be set on the time permitted to execute the function. To prevent permanent faults from affecting both versions in the same way, the implementations of the state transition functions should be diverse.

The *write* and *recover* functions serve two purposes. The first is to store checkpoints on the stable storage and recover from them in case of a detected fault. Checkpoints are written after s rounds. The second is to clone the fault-free state of a forwarded version. After a successful roll-forward a pipe is opened and the forwarded version is requested to write its abstract state to the pipe. Concurrently, the other version is asked to recover its state. Thereafter, both versions are in the same state and can be executed for further fault detection.

The controller uses a simple request/response protocol to supervise the execution of versions. It sends a request message to a version to execute a certain function using the supplied arguments. After the requested function is executed, the version calculates a hash value over the new state. The hash value is wrapped into a response message and returned to the controller for state comparison.

Due to the modular design, the implementation of the software fault-detection and recovery schemes requires only minor changes to the application code.

3.1 Fault Detection

For fault detection, two versions $v1$ and $v2$ are executed simultaneously within processes or threads, scheduled on the two logical processors of an SMT platform. After each round, the controller compares the hash values returned by the versions.

If the hash values are not equal, it can deduce that at least one of the threads is faulty. To decide which version is fault-free, a third hash value is needed for a majority vote.

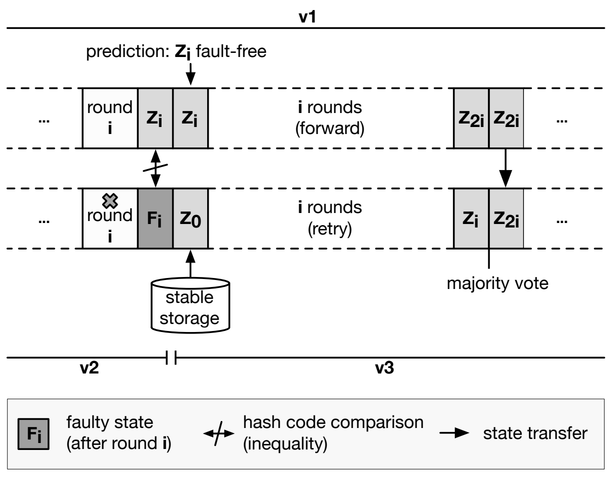

Figure 1. The optimistic probabilistic recovery scheme with a correct guess of the fault-free version.

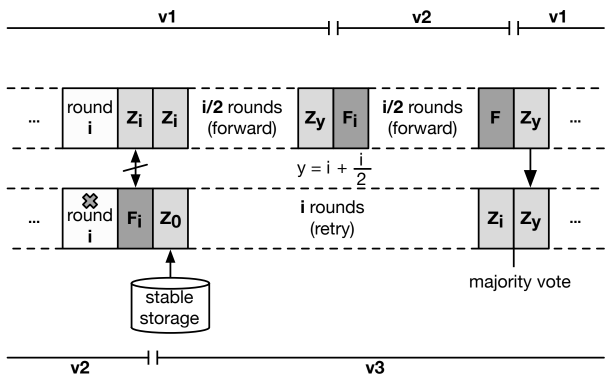

Figure 2. The optimistic deterministic recovery scheme.

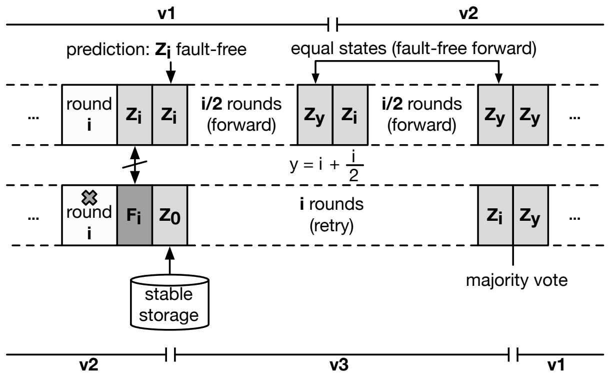

Figure 3. The pessimistic probabilistic recovery scheme. The fault-free version is correctly guessed.

If a fault is detected at the end of round i after the last checkpoint, the state of the third version $v3$ is recovered from the most recent checkpoint. Therefore the controller issues a *recover* request to $v3$. Then $v3$ is executed by the second logical processor until round i inclusively. With the hash value calculated by $v3$, a majority vote can be taken by the controller. If another fault occurs during the retry, it will normally lead to three different states so that the whole system has to be reset to the last checkpoint. Simultaneously to the retry, a roll-forward is executed on the first logical processor to avoid a performance loss during the retry.

3.2 Fault Recovery

The four implemented recovery schemes employ different forwarding strategies. All have in common, that the roll-forward thread may not pass the next checkpoint. Hence, the number of rounds to be forwarded is always trimmed down to $s-i$, although we do not mention that in the sequel.

In the **optimistic probabilistic** scheme (Figure 1), the controller guesses which of the versions is fault-free, binds it to the first logical processor and forwards it for i rounds. If the majority vote after the retry proves that the guess was correct, the state of the forwarded version is copied to $v3$ via a pipe by issuing *write* and *recover* requests. The execution of both versions is continued starting from the forwarded state, achieving a progress of i rounds, i.e. there is no performance loss. If the guess was not correct, the execution of the fault-free version and $v3$ is continued and no progress is made. Obviously, the probability to correctly guess the fault-free version is at least 0.5. If the system gives any clue as to which version is fault-free (for example, by providing information about cache-faults or timeouts), then the probability of a correct guess can be substantially higher.

In the **optimistic deterministic** scheme (Figure 2), the versions $v1$ and $v2$ are both forwarded for $i/2$ rounds on the first logical processor. After the retry is completed, the state of the version which was correct after round i is transferred to $v3$. A progress of $i/2$ rounds is reached in any case.

Note that the optimistic schemes cannot detect faults affecting the first logical processor during the roll-forward. To detect further faults, the pessimistic variants run one or two virtual duplex systems in the roll-forward thread.

In the **pessimistic probabilistic** variant (Figure 3), the controller tries to predict the fault-free version. The state of that version is then copied to the other version. As a result, versions $v1$ and $v2$ are in the state which was guessed to be fault-free. Then both versions are executed as a virtual duplex system on the first logical processor, however without switching context after every round. After the roll-forward has been completed, the hash codes of the forwarded versions are compared. If they differ, a fault has affected the roll-forward thread and execution is continued starting from the state of $v3$. If the prediction was correct and no fault occurred during the forward, a progress of $i/2$ rounds is attained. Otherwise no progress is made.

In the **pessimistic deterministic** algorithm (Figure 4), two virtual duplex systems are executed in the roll-

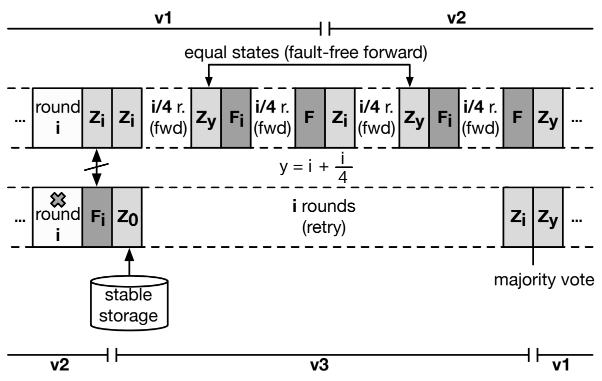

Figure 4. The pessimistic deterministic recovery scheme.

forward thread - one forwarding the state of $v1$ and the other one forwarding the state of $v2$. Hence, four processes are scheduled on the first logical processor. If no fault influences the roll-forward of the fault-free version, a progress of $i/4$ rounds is made.

4 Simulation Methodology

To compare the performance of the described recovery schemes we did simulation studies on an Intel Pentium 4 Hyper-Threading [13] machine with 1GB of RAM, running Linux kernel 2.6. We did five software-based fault-injection runs and averaged over the measured runtimes (relative avg. deviation 2.17%). In each fault-injection run two million rounds were executed. The versions did a matrix multiplication of two (5,5)-matrices. Hash values over the resulting matrices were calculated and compared.

Before issuing a message, the controller used a random number generator to determine whether a fault should be injected. The outcome of the test was added to the request message as an additional parameter. If the value of the parameter was true, the resulting matrix was altered by the version. Hence, a fault was detected by the controller and the recovery algorithm will be executed.

The parameters used in our measurements are defined as follows:

- The number of rounds between checkpoints, s. For example, if the value of s is 100000, one checkpoint is stored on the stable storage every 100000 rounds. The values chosen for s are: 75000, 100000, 125000, 150000 and 200000.
- The probability of a fault per round, f. We used three different values of f: 10^{-5}, $\frac{1}{4} \cdot 10^{-5}$ and 10^{-6}. For example, a value of 10^{-5} means that one fault occurs every 10^5 rounds on average.
- The ratio of correct guesses for the probabilistic recovery schemes, p. For example, if p is 0.7, 70% of the predictions on which version to forward are correct.

5 Results

The execution of the versions on two logical processors yielded an average speedup of 34.1% in comparison to a mapping of processes/threads to one logical processor. This is in line with the observations of [6] and the ranking from [9]. We found no significant difference in the execution times of the thread and process variant. This could be explained by the fact, that the Linux thread library creates threads corresponding to a process in the kernel. Figure 5 shows the execution times of the four recovery schemes for a mapping on one or two logical processors, respectively. Execution times were measured in milliseconds, those of the probabilistic schemes resulted in calculating the arithmetic mean for p=$\{0.5, 0.7, 0.9\}$.

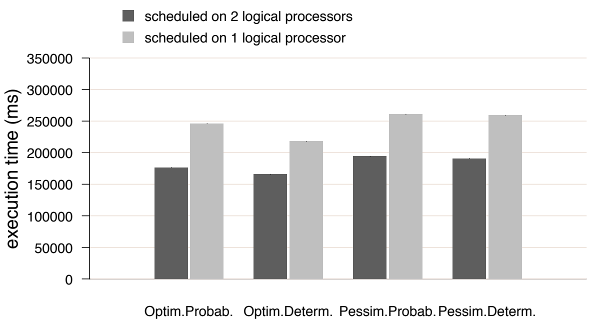

Figure 5. Hyper-threading performance gain for different recovery strategies.

Table 1 shows the execution times of the optimistic deterministic variant divided by those of the optimistic probabilistic variant. For $p = 0.5$, the optimistic deterministic variant was always faster than the optimistic probabilistic variant. Even with 70% correct guesses ($p = 0.7$), the maximum performance loss of the optimistic deterministic variant was merely 3.63%. For $p = 0.9$ the gains of the optimistic probabilistic variant highly depend on f and s. For maximum values of f and s ($f = 10^{-5}$, $s = 200000$), the optimistic probabilistic variant was 16.9% faster than the optimistic deterministic variant.

A high ratio of correct guesses was required to let the probabilistic schemes outperform the deterministic schemes significantly. For $p = 0.5$ and $p = 0.7$, differences between the schemes became small. Comparing the pessimistic deterministic with the pessimistic probabilistic scheme leads to similar results. Figure 6 plots the slowdown of the pessimistic deterministic variant for $p = 0.7$, $p = 0.9$. For $p = 0.5$, the performance of the pessimistic deterministic and pessimistic probabilistic schemes showed no difference. For $p = 0.7$ and $f = 10^{-5}$, the average performance advantage of the optimistic probabilistic scheme was no better than 4.4%. To let the optimistic probabilistic scheme gain a significant advantage over the optimistic deterministic scheme, 90% of the guesses had to be cor-

		s		
f	p	75000	150000	200000
10^{-5}	0.5	0.9862	0.9384	0.9760
	0.7	1.0176	1.0305	1.0363
	0.9	1.0671	1.1217	1.2030
$\frac{1}{4} \cdot 10^{-5}$	0.5	0.9614	0.9878	0.9895
	0.7	0.9861	0.9852	1.0121
	0.9	1.0193	1.0580	1.0666
10^{-6}	0.5	0.9905	0.9891	0.9816
	0.7	0.9969	0.9911	1.0291
	0.9	1.0092	1.0009	1.0131

Table 1. Execution time ratio of the optimistic deterministic to the optimistic probabilistic scheme. Values greater than 1 imply that the deterministic scheme is slower.

rect ($p = 0.9$) and a large value had to be chosen for s ($s = 200000$).

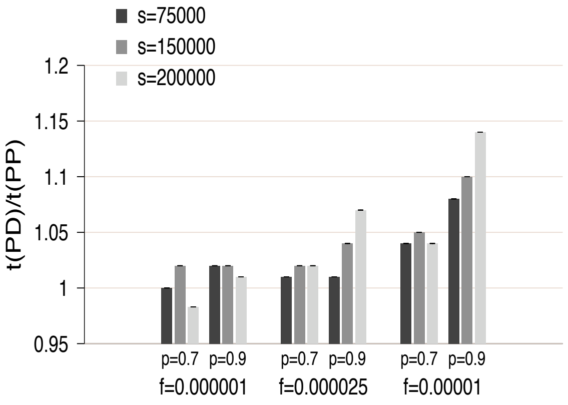

Figure 6. Execution time ratio of the pessimistic deterministic to the pessimistic probabilistic scheme. Values greater than 1 imply that the deterministic scheme is slower.

Performance differences between the optimistic and pessimistic schemes are highly dependent on s and f, but hardly on p. Figure 7 shows the slow-down of the pessimistic schemes compared to their optimistic counterparts. The values plotted for the probabilistic scheme apply to $p = 0.7$. For moderate values of f and s, both schemes performed similarly. For $f = 10^{-5}$ and $s \geq 100000$, the optimistic schemes surpassed their pessimistic counterparts by more than 10%.

Figures 8 and 9 show the execution times of the four recovery schemes for $f = 10^{-5}$. The probabilistic schemes are illustrated for $p = 0.7$ and $p = 0.9$. We see that for $p = 0.9$ the optimistic probabilistic scheme requires a large checkpoint interval to make good use of the high probability for a correct guess.

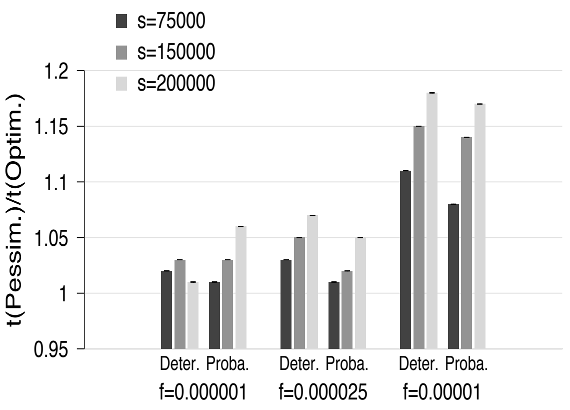

Figure 7. Execution time ratio of the pessimistic schemes to the corresponding optimistic schemes. Values greater than 1 imply that the pessimistic scheme is slower. The relations between the probabilistic schemes apply to $p = 0.7$.

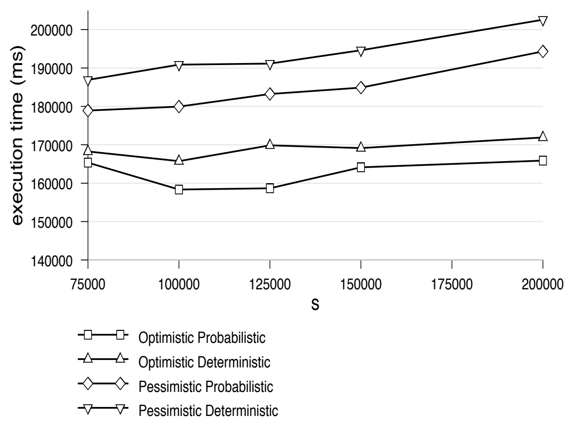

Figure 8. Execution times of the recovery schemes ($p = 0.7$,$f = 10^{-5}$).

6 Summary and Conclusion

We presented a performance analysis of four different software-based recovery schemes, tailored for a contemporary SMT-system under the influence of faults. The execution of the versions on two logical processors gave an average speedup of 34.1% in comparison to a mapping of processes/threads to one logical processor. No significant difference in the execution times of the thread and process variants was measured. For $p = 0.5$, the optimistic deterministic variant was faster than the optimistic probabilistic variant. With $p = 0.7$, the optimistic probabilistic variant began to perform better than its deterministic counterpart. For maximum values of p, f and s ($p = 0.9$, $f = 10^{-5}$, $s = 200000$), the optimistic probabilistic variant was 16.9% faster than the optimistic deterministic

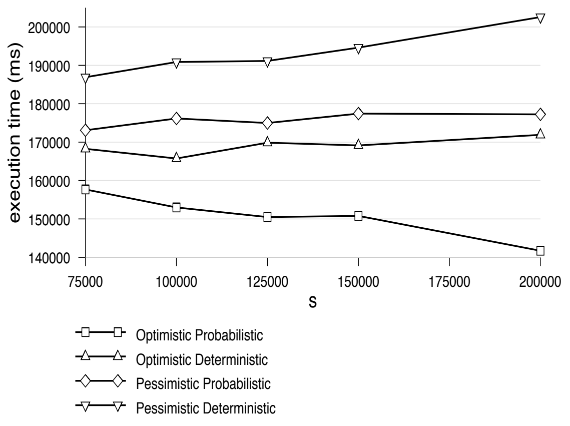

Figure 9. Execution times of the recovery schemes ($p = 0.9$, $f = 10^{-5}$).

one. Thus, a high ratio of correct guesses is required to let the probabilistic schemes outperform the deterministic schemes. The comparison of pessimistic schemes yielded similar results. The optimistic probabilistic scheme with $p = 0.9$ required a large checkpoint interval to make good use of the high guess precision.

[1] J.F. Ziegler et al., IBM experiments in soft fails in computer electronics (1978-1994), *IBM Journal of Research and Development*, 40(1), 1996, pp. 3-18.

[2] A. Romanovsky, On version state recovery and adjudication in class diversity, *International Journal of Computer Systems Science and Engineering*, 17(3), 2002, pp. 159-168.

[3] A. Avizienis, The Methodology of N-Version Programming, in Lyu (Ed.) *Software Fault Tolerance*, (Hoboken: John Wiley and Sons Ltd, 1995), pp. 23-46.

[4] D. Marr et al., Hyper-Threading Technology Architecture and Microarchitecture, *Intel Technology Journal*, 6(1), 2002, pp. 4-15.

[5] T. Lovric, *Fault detection by systematic diversity in design-diverse and temporal redundant computer systems and their evaluation through fault-injection (in German)* (Berlin: Logos Verlag, 1997).

[6] J. Bulpin and I. Pratt, Multiprogramming Performance of the Pentium 4 with Hyper-Threading, *Third Annual Workshop on Duplicating, Deconstruction and Debunking*, 2004, pp. 53-62.

[7] K. Echtle and B. Hinz and T. Nikolov. On Hardware Fault Diagnosis by Diverse Software, Proceedings of the 13th International Conference on Fault-Tolerant Systems and Diagnostics, pp. 362-367, 1990

[8] P. Sobe, B. Fechner, J. Keller. Classification and Unified Modeling for Duplication-based Recovery. In Proc. Fifth European Dependable Computing Conference (EDCC-5), 2005.

[9] B. Fechner, J. Keller, P. Sobe. Performance Estimation of Virtual Duplex Systems on Simultaneous Multithreaded Processors. In Proc. 9th IEEE Workshop on Fault-Tolerant Parallel, Distributed and Network-Centric Systems, 2004.

[10] 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.

[11] D. Pradhan, D. Sharma, and N. Vaidya. Roll–Forward Checkpointing Schemes. In M. Banatre and P. Lee, editors, Hardware and Software architectures for Fault-Tolerance, No. 774 Lecture Notes in Computer Science. Springer, 1994.

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

[13] M. Withopf. Virtual tandem: Hyperthreading in the new Pentium 4 with 3 GHz (in German). ct, 24:120ff, 2002.