Redundancy and Replication

Fault Tolerance on the System Level

Kurt Kanzenbach
Friedrich-Alexander University Erlangen-Nuremberg
ly80toro@cip.cs.fau.de

ABSTRACT

In the past decade the clock rates and densities of processors, enabled by new and smaller manufacture processes, increased. Due to this development current processors become less and less reliable and more vulnerable to physical effects such as the cosmic radiation, which can result in transient failures. One common approach to address hardware failures like these is to use redundant hardware. However, this strategy is expensive and not suitable in all cases. One alternative is to use redundancy and replication on the system level without exploiting extra hardware. Since software techniques may be implemented on different levels of the software stack, this seminar paper introduces and compares software-only approaches using redundancy and replication in order to detect and recover from hardware faults. In particular, two compiler-based, one process-based and one hypervisor-based techniques are discussed. The evaluation demonstrates that it is possible to detect and recover from hardware failures on the software or system level, whereas implementing fault tolerance on each software-level has its own advantages and drawbacks.

1. INTRODUCTION

Nowadays many of the technical systems we use in our everyday lives like airplanes, cars, high-speed trains, smart phones, medical devices, online banking and much more are software-controlled. Even if such software would be developed and tested perfectly it still would fail sometimes. One reason for that is, current hardware becomes less reliable. For instance transistors, used in current microprocessors, are more vulnerable to physical effects than a decade ago. Since the size of transistors decreases, the amount used in processors is steadily increasing. This leads to a higher error rate of the complete chip [13]. Errors caused by external events like cosmic radiation are called *transient faults*. These faults may manipulate stored values in the processor's registers or internal buffers which can result in an incorrect program execution.

One common technique for addressing hardware faults is to augment hardware with fault detection and recovery mechanisms or to use redundant hardware. However, this approach is expensive not suitable in all cases. Moreover additional hardware have a higher energy consumption and may have a higher weight. Hardware approaches are well known and deployed in many critical systems like space shuttles or nuclear plants. Applying hardware techniques to e.g. consumer systems is not meaningful, since it is simply to expensive. This is why software-only techniques are gaining popularity. First of all software schemes can run on off-the-shelve hardware without modifications and their fault coverage level may be dynamically configured [9], which make them really flexible. Second, software techniques can be applied after a system is in active use. Thus, software schemes are an attractive alternative to directly hardware-based fault tolerance.

Software-only approaches may be implemented in different software layers using different components for replication like instructions, processes or complete virtual machines. Implementing fault tolerance in each of these levels have their own advantages and drawbacks. This seminar paper introduces some of the most popular software techniques as examples and compares them to each other.

This seminar paper is structured as follows. Section 2 gives insight of the terms of redundancy and replication, the sections 3, 4 and 5 describe the chosen software fault recovery techniques and section 6 and 7 compare the approaches and show pitfalls and drawbacks of each.

2. BACKGROUND

One key concept of implementing software fault tolerance is redundancy and replication. To a system, every component is considered redundant, if it can work fully functional work without this component. This means, redundancy includes all resources which are not necessary for the functionality of a system like additional hardware or extra information [7]. Since redundancy can include a variety of different components, it can be classified into four major categories: *structural*, *functional*, *information* and *temporal* redundancy. Structural redundancy includes additional components like extra processors, memory and files. Evaluating functions of a system using different implementations, is called functional redundancy. Information redundancy includes all additional data used in a program. This is not limited to exact copies or error-detecting codes, but it includes implicit data as well. For instance if a variable contains a betterment tax money amount and a other one contains the same value without taxes, this is also considered as data redundancy. Temporal redundancy involves adding additional execution time to a

function or algorithm to overcome faults [7, 12].
The fundamental type of redundancy used for replication is structural redundancy. Redundant components of a system are used in a coordinated way to detect and mask faults transparently [18]. All the software fault tolerance techniques described in the next sections use replication in different ways. However, the basic scheme of replication is always the same. First of all the input for the replicas is duplicated, then all replicas execute the input and last the output of all replicas is compared to find one correct version. This concept is illustrated in figure 1 by using two redundant executions. The logical boundary of redundant execution is called *Sphere of Replication* (SoR) [17].

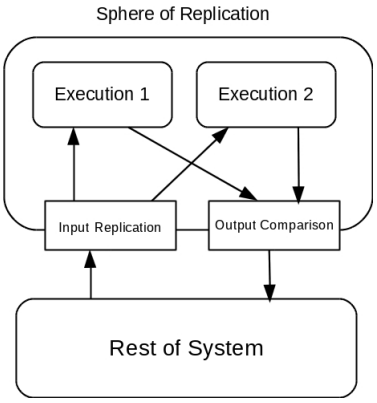

Figure 1: Basic Scheme of Replication [9]

However, using replication encounters own problems. In order for the output comparison to work correctly, it is necessary that all replicas produce the same output by a given input. Otherwise it is not possible to recover from hardware faults using majority voting. This characteristic is called *replica determinism* [11]. Depending on what is replicated, ensuring replica determinism can be a major effort. For instance, if a process is replicated, non-deterministic function calls like getting the time-of-day have to be consistent among all replicas. Moreover the size of the SoR influences the frequency of output checks and input replications heavily. These checks may degrade the performance and reliability. Furthermore the replicas should be isolated from each other, so that a faulty replica cannot manipulate the internal state of another replica.
The next three sections introduce examples of software tolerance techniques using replication, thereby section 6 evaluates how the described problems above are solved and what advantages and drawbacks can be derived.

3. INSTRUCTION-LEVEL FAULT TOLERANCE

This section describes two approaches to address transient processor failures by using a software-only fault detection and recovery mechanism. The compiler is used to replicate instructions as well as data stored in processor registers. Many techniques on the compiler-level like EDDI [10], Spot [14] or SWIFT [15] have been implemented and tested over the last decades. This is why this sections introduces two of the most popular ones in order to demonstrate the general functional principle.

3.1 Software-Implemented Fault Tolerance

SWIFT [15] is a compiler extension which replicates the instructions of a program. The duplicated instructions use different registers and are scheduled among the original ones. At certain points in the program's execution validation checks are inserted in order to detect faults. These checks compare the computed values with their redundant copies. If they do not match a fault is detected and a appropriate fault detection subroutine is called.
However, when do these validation checks need to be inserted? Since the visible impact of a wrong calculation or execution path is observed when a program performs output, the output has to be checked. A program has basically two options to accomplish output: The first one is to use system calls or external libraries like GNU C Library [2]. The second is to use memory mapped I/O via calls to `load` and `store` operations. Using memory mapped I/O the register operands for these two instructions must be checked right before using them. Though `load` and `store` operations are not duplicated, since the main memory is assumed to be ECC [1] protected. To replicate the result of a `load`, a `mov` instruction is performed to the shadow register.

```
                          1: br faultDet, r4 != r4'
                             ld r3 = [r4]
ld r3 = [r4]              2: mov r3' = r3
                             add r1 = r2, r3
add r1 = r2, r3           3: add r1' = r2',r3'
                          4: br faultDet, r1 != r1'
                          5: br faultDet, r2 != r2'
                             st [r1] = r2
st [r1] = r2
```

(a) Original Code (b) SWIFT Code

Figure 2: SWIFT Example [13]

For an example of the SWIFT transformation see figure 2. The registers denoted with `'` are the redundant copies of the original registers. The `add` instruction is duplicated. While instruction **1**, **4** and **5** are inserted to insure that the `loads` and `stores` will be executed correctly. Otherwise the `faultDet` subroutine is called. Instruction **2** duplicates the result of the previous `load`.
As mentioned earlier a program can also do output via external functions which are not necessarily protected by SWIFT. Therefore the only way to ensure correct computation is to check the input parameters for those functions. Parameters passed by memory do not need to be checked, because the appropriate `store` instruction is SWIFT protected. So only register parameters will be validated by SWIFT in the way as shown in figure 2. After a function returns, the result is replicated as well.
The SWIFT approach is also able to protect against wrong control flow execution paths. They are mostly determined by comparisons of registers. Thus the registers used in `branch` instructions are validated accordingly.
Whereas SWIFT is a reliable technique to detect transient processor failures, it is incapable of recovering. There is no way to determine whether the original value or the copy is the right one, if a fault occurs. This is why SWIFT got extended to SWIFT-R [13], where every instruction is replicated twice. This results in three copies of register values. If a single bit-flip is detected the original value can be restored by majority voting. Regarding to the example in figure 2

the instruction 4 would be replaced by `majority(r1, r1', r1")`, whereby `r1'` and `r1"` are the redundant copies of `r1`. While SWIFT-R can detect and recover from many faults, there are still limitations: First of all if a fault occurs between the validation and the use of a register, then the fault remains undetected. This is why the compiler tries to schedule the validation checks as closest as possible to the `load` and `store` operations. This cannot prevent the fault, however decrease the probability to a minimum. Furthermore the same timing issue exists before copying a value from a previous `load`. Moreover opcode bit failures which lead to changes in instructions can be a major problem. If a instruction is transformed into a conditional or unconditional jump then a wrong execution path may be taken. Also bit failures in the program counter or stack pointer register are not covered by the SWIFT-R approach.
Another compiler based technique is TRUMP which enables fault recovery by using only two copies. The next subsection introduces TRUMP in detail.

3.2 Triple Redundancy Using Multiplication

The TRUMP [13] approach uses single redundancy of instructions and registers. The duplicated copies are not directly taken, but rather encoded using AN arithmetic codes. This mathematical code allows to detect which version of a value, original or copy, is the uncorrupted one.
AN-codes are simple arithmetic codes used in TRUMP for error detection. A and N are positive numbers and A is constant. The code word is the original value multiplied by A. Therefore a valid code word must be divisible by A without remainder [16]. The AN-codes are preserved during arithmetical operations, as the following equations demonstrate [13]:

$$(Ax) + (Ay) = A(x + k) \quad (1)$$

$$(Ax) \cdot k = A(x \cdot k) \quad (2)$$

For the TRUMP technique $A = 3$ were chosen, which is big enough to detect single bit failures, since they lead to an addition or subtraction of 2^k. TRUMP is similar to SWIFT except the duplicated copy AN-encoded. In particular, the code word is built by multiplying the original value times three. The figure 3 shows the same example used in section 3.1 except it uses TRUMP to protect the code and allows recovery. The registers denoted with `t` store the AN-encoded version of the corresponding register. To replicate the result of the `load` instruction, the original value is multiplied by three.

```
ld r3 = [r4]

add r1 = r2, r3



st [r1] = r2
```

(a) Original Code

```
1: call recovery, 3*r4 != r4t
   ld r3 = [r4]
2: mul r3t = 3, r3
   add r1 = r2, r3
3: add r1t = r2t, r3t
4: call recovery, 3*r1 != r1t
5: call recovery, 3*r2 != r2t
   st [r1] = r2
```

(b) TRUMP Code

Figure 3: TRUMP Example [13]

The **recovery** subroutine is called, whenever the original value times three does not match the AN-encoded variant. The recovery itself is then a bit more complex than the simple majority voting as introduced in SWIFT-R. The figure 4 shows the recovery routine.

```
x = original copy;
y = AN-encoded version;
if (3*x != y) {
  if (y % 3 == 0) {
    x = y / 3;
  } else {
    y = 3 * x;
  }
}
```

Figure 4: TRUMP recovery [13]

This routine is able to detect which of the two values `x` and `y` is the faulty one and recovers appropriately. Basically a fault has occurred when the original copy times three is not equal to the AN-encoded version. If the encoded word is zero modulus three then a fault has changed the value of `x`. `x` can be restored by dividing `y` by three. For `y` it is the same by multiplying `x` by three.
The TRUMP approach is a good alternative for SWIFT-R, because it needs less resources, but it is not applicable in all instruction sequences. Registers usually have a fixed size of e.g. 32 or 64 bits. If the program calculates values near at the maximum size of a register, then the redundant register is to small to store the AN-encoded version. Furthermore the AN-encoded variants preserve through arithmetic operations, as shown above, but not through logical bit operations like `and` or `xor`. In both cases TRUMP cannot be applied to secure the instruction stream.
While SWIFT and TRUMP use replicated instructions in order to ensure fault detection and recovery, the technique described in the next section uses redundant processes.

4. PROCESS-LEVEL FAULT TOLERANCE

Implementing software fault tolerance using replicated instructions, as stated above, is only one approach to detect and recover from hardware failures. A second, more generic, approach is described in this section. It uses a set of redundant processes for an application. This technique is called *Process-Level Redundancy* (PLR) [17]. Alex Shye et al. have developed, implemented and evaluated PLR, the technique which is introduced here.
The Sphere of Replication of PLR includes the application and libraries itself. PLR intercepts the start up code of an application at the beginning and creates the redundant processes. One becomes the *master* and the others act as so called *slaves*. The generic scheme of PLR can be seen in figure 5.
Since redundant processes are created, all input has to be replicated and their output needs to be compared to ensure correct behavior. Input and output is usually done via system calls. Thus, PLR includes a system call emulation unit (see figure 5). The result of every system call which attempts to do input, like reading data from a file, is executed only once and the data is replicated and forwarded to all processes. On the other hand all parameters including buffers passed to write system calls are compared in order to detect

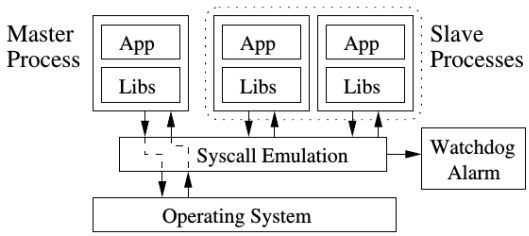

Figure 5: PLR with three redundant processes [17]

faults. Moreover the system call emulation unit also ensures, that non-deterministic system calls are executed only once, by the master, and the same result is propagated to all redundant processes by emulating the system call. Deterministic system calls which do not manipulate the system's state are executed by all processes. For input replication and output comparison PLR uses a shared memory segment between all redundant processes.

PLR is able to recover from transient hardware faults when a set of at least three redundant processes are used. The emulation unit compares the output of all processes and uses majority voting to correct failures. This is similar to SWIFT-R's recovery scheme. However, the system call and output comparison only takes place when all processes perform the same system call. For instance, if one process hangs in an infinite loop due to a transient fault in a variable, the progress of the entire application does not continue. Therefore a second component, the watchdog alarm, is deployed. The watch dog starts a timer, whenever the first process enters the emulation unit. If not all processes reaches the same system call in a specified amount of time, the faulty process will be killed and the recovery is invoked. The recovery involves duplicating one correct process e.g. by the `fork()`[1] system call on Linux. Furthermore it is possible that a process performs an illegal instruction like dereferencing a faulty pointer which results in a segmentation fault. Hence, appropriate signal handlers are registered at the operating system and an error is indicated, so that the next emulation unit invocation recreates the crashed process.

Like all approaches PLR has its own limitations. If a fault occurs while executing the PLR code in the emulation unit, it may result in an unrecoverable error. Moreover faults in the operating system are not covered by PLR and may lead to an incorrect execution as well. Also PLR does not support shared memory and multi-threaded applications currently.

The next section introduces an approach which uses replication on a larger scale. The hypervisor-based fault tolerance makes use of redundant virtual machines to mask hardware faults.

5. REDUNDANT VIRTUAL MACHINES

This section examines the implementation of software fault tolerance using a set of redundant virtual machines (RVMs). In order to create and run virtual machines, an additional software layer, called *hypervisor* or *virtual machine monitor* (VMM), is introduced. The exact structure of a system using redundant virtual machines can be found in figure 6.

The operating systems inside the virtual machines access all

[1] `http://linux.die.net/man/2/fork`

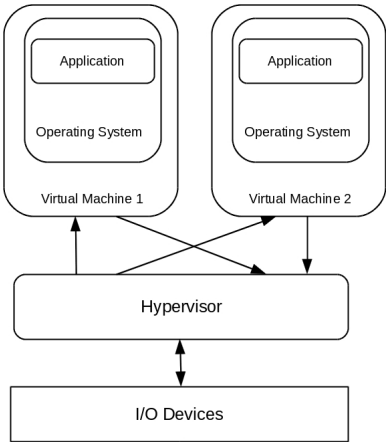

Figure 6: RVM-System with two Virtual Machines [9]

hardware devices indirectly through the hypervisor. Since the Sphere of Replication includes entire virtual machines, the hypervisor is responsible for input replication and the output comparison by evaluating the operating systems' instructions. The replication and output comparison can be complicated, because the hypervisor has to deal with non-deterministic instructions and asynchronous hardware interrupts. One protocol implementing a hypervisor-based fault tolerance is described by Bressoud and Schneider [5].

They separate the virtual machines into a primary one and several backup ones. If one virtual machines fails due to a hardware fault another one takes over transparently. This is quite similar to the PLR approach for processes.

The critical part in implementing redundant virtual machines, is to keep them in the exact same state. Therefore the machines have to execute the exact same sequence of instructions. However, non-deterministic operations can lead to different internal states of the VMs even while executing the exact sequence of instructions. This is why the hypervisor is used to catch and emulate those operations. Non-deterministic instructions are executed only once by the hypervisor of the primary virtual machine and the result is saved. If a backup VM attempts to execute the same non-deterministic instruction, the buffered result is returned by the backup's hypervisor. Deterministic instructions like arithmetic operations are directly passed through the hardware by the hypervisors.

The hypervisor is responsible for the input replication as well. The input for a virtual machine are basically hardware interrupts. Since interrupts have to executed by every single virtual machine at the same instruction in order to ensure replica determinism, the hypervisors intercept the interrupt delivery. Therefore the execution of the virtual machines are partitioned into epochs. An epoch usually consists of several thousand instructions. Interrupts received by the primary VM are buffered and at the end of an epoch they are replicated and delivered to the backup virtual machines. So all VMs receive the interrupts at the same instruction boundary. Interrupts itself are suppressed by the hypervisors in all backup VMs to ensure that each interrupt is only executed once. Also the output requests of all virtual machines have to be executed only once. This is done by the primary VM. Output requests at the backup virtual machines are simulated by their hypervisors, but not executed.

If the primary virtual machine fails, one backup VM is promoted to become to the primary. However, if the primary fails before it reaches the end of an epoch, buffered interrupts may be lost. This is why the backup hypervisor generates uncertain interrupts for all outstanding I/O operations. Therefore all hypervisors have to keep track of outstanding I/O operations. The backup system then results in a clean state and is able to act as primary.
Furthermore the clock time of all redundant virtual machines have to be consistent among all replicas. Since technical clocks can diverge fast, a synchronization has to take place. Otherwise, if the primary fails and a backup virtual machine takes over, the backup VM might be in the past which can lead to a wrong program execution. Therefore, the primary hypervisor sends the time of the primary to all hypervisors of the backup machines at the end of each epoch and the time is adjusted accordingly.
This section described a basic protocol by Bressoud and Schneider [5] to implement software fault tolerance by using redundant virtual machines. The next section discusses all introduced techniques by evaluating advantages and drawbacks depending on what is replicated.

6. DISCUSSION

Each of the presented software-only fault tolerance approaches can be categorized in one of the following three categories: instruction-level, process-level and redundant virtual machine fault tolerance. This sections examines what implications can be derived by implementing fault tolerance on different software levels using different replicas.
The instruction-level based techniques like SWIFT-R or TRUMP duplicate instructions. Every single operation is executed at least twice. Thus, the Sphere of Replication includes the complete processor. Therefore every data entering the processor is replicated and every output is compared in order to ensure correct program behavior. This results in a lot of replications and comparisons, since every `load`, `store` and function call is checked by instruction-level based schemes. Due to this fact, the performance is degraded heavily by those techniques. For example, Reis et al. [13] have implemented SWIFT-R and TRUMP into GNU GCC compiler[2] for the PowerPC 970 architecture. Using a variety of benchmarks, they evaluated that SWIFT-R is about 99 percent slower than the original program. For TRUMP they measured a performance overhead of only 36 percent. However, as stated in section 3.2, TRUMP cannot protect all code sequences and is therefore not as reliable as SWIFT-R. Whereas the reliability of instruction-level techniques is in general good, they also detect faults which have not to be destructive necessarily. Not every fault occurring on the instruction-level is leading to a wrong program computation or program crash. So it might not be not necessary to detect that many faults. But, instruction-level schemes cannot do that, because they have not enough insight into the application to decide which instruction needs to protected and which not. Therefore it might be a good idea to protect only critical parts of a program using one of the approaches. Borodin et al. [4] have evaluated by doing so, the performance can be increased heavily. By their approach a programmer is able to specify which parts of a program are critical and therefore protected and which are not.

[2] `https://gcc.gnu.org/`

As described in section 2 using replication encounters problems like replica determinism. By replicating instructions replica determinism is actually no problem, since all instructions are executed multiple times. This is why there is no additional overhead for ensuring replica determinism. On the other hand there is no isolation between the replicas. A faulty instruction may manipulate the internal state of another instruction or register which can result in a program failure. However, many faults can be detected and recovered easily by restoring e.g. a register value.
Using instruction-level schemes in practice may be a problem, since for many techniques the source code of a program has to be available. That is not the case for legacy software. There are also instruction-level schemes operating on binaries. But, they most likely degrade performance even more, as the compiler is not able to do optimizations anymore.
Another approach providing software fault tolerance is by using redundant processes. The Sphere of Replication for those techniques include the application and libraries itself. This is why the input replication and output comparison involves system calls, since an application may gather input and produce output only by these. Every time the application attempts to make a system call it has to be intercepted. The input of system calls which read data like `read()`[3] is replicated. The replication of such functions can involve complete buffers. The same holds true for system calls which produce output like `write()`[4]. There, complete buffers may have to be compared from each redundant process. As a result the input replication and output comparison using redundant processes takes longer than in instruction-level schemes, where only registers have to be replicated or compared. However, the amount of system calls in an application is usually lower than the amount of `load` and `store` operations. So the system call emulation unit is not invoked that much. Regarding to Shye et al. [17] the average performance overhead for recovery (using three processes) is about 41.1 percent using their PLR technique. The performance is better than e.g. SIWFT-R's, because using redundant processes allows to ignore all faults, that do not result in an incorrect program execution. However, the performance of memory intensive applications may be degraded much more, because by using redundant processes the memory bandwidth may be easily saturated.
Ensuring replica determinism by using process-level schemes involves emulation of system calls, as there are non-deterministic system calls like `gettimeofday()`[5]. The mechanism has to guarantee that all processes get the same result from such system calls which include additional time. The isolation between the replicas is much better compared to instruction-level techniques, but is still not perfect. In PLR the system call emulation unit uses a shared memory segment to share data within all processes. A fault which results in manipulating this area can still result in an incorrect program execution. Furthermore the recovery of a process due to a fault is way more expensive compared to instruction-level techniques, since a complete new process has to be created. One advantage from process-level techniques over instruction-level schemes is, that e.g. PLR can be applied to binaries without modification of the application or operating system.

[3] `http://linux.die.net/man/2/read`
[4] `http://linux.die.net/man/2/write`
[5] `http://linux.die.net/man/2/gettimeofday`

PLR uses the Intel Pin Framework [8] to manipulate the binary in order to patch the start up code and the system call invocations.
Replication can be used to run redundant virtual machines as well. The Sphere of replication for RVM-based techniques involves entire systems including operating systems and applications. Hence, the input replication is mainly determined by interrupts and the output by instructions and I/O requests. Therefore the hypervisor has to replicate every single interrupt arriving to an virtual machine. Those interrupts need to be buffered and delivered to hypervisors of all backup virtual machines. Moreover the output requests have to be intercepted and executed only once which results in even more additional overhead. Bressoud and Schneider [5] implemented a prototype for their protocol described in section 5. They used a system containing two HP-9000/720 PA-RISC processors connected via Ethernet. For the performance and reliability testing, two virtual machines, one primary and one backup, were used. A set of CPU intensive and I/O intensive benchmarks were tested. The performance decrease for the CPU intensive benchmarks is about 84 percent by using an epoch length of 32000 instructions. For I/O intensive benchmarks which read data from disks the decrease is roughly 98 percent, for writing data it is about 64 percent. This is due to the fact that the protocol involves a lot of communication, buffering of all interrupts, their corresponding data and emulating instructions. However, their prototype was not fully optimized. There are actually protocols, based on Bressoud and Schneider's, which are used in current hypervisors. One of them is Remus [6] which is based on the open source hypervisor Xen [3].
The protocol of ensuring replica determinism for RVM-based systems is more complex than in PLR, since all virtual machines have to execute the same sequence of instructions. Therefore, interrupts have to be buffered and delivered to all other machines at the same point in execution. This may by tricky, as each hypervisor has to get control of the execution of each virtual machine at the same instruction. Bressoud and Schneider used the recovery register of the PA-RISC architecture to accomplish this. Other major architectures like Intel x86 have similar mechanisms. However, not all processors have such registers which makes it more difficult to ensure interrupt determinism. In addition non-deterministic hardware instructions have to be executed only once. The result is saved and delivered to all other VMs accordingly. The complete protocol for running redundant virtual machines is complex. But, implementing it once for a specific hardware is enough, as the hypervisor is completely independent from the operating systems and the application. Also using a hypervisor provides nearly perfect isolation between the virtual machines. A failure of one of them does not interfere with the others. On the other hand recovering from a hardware failure may take some time, since the complete virtual machine has to be restarted.
A summary showing the characteristics of implementing redundancy and replication on different layers can be found in table 1. It shows what is replicated, the SoR, when input replication and output comparison is done, the effort for ensuring replica determinism, performance and reliability. All software techniques have in common that they degrade performance and cannot protect against all occurring hardware faults, whereas they differ in other characteristics as shown in table 1.

This section evaluated the different techniques and showed advantages and drawbacks of using different levels of replication. Section 7 draws a conclusion to using software based fault tolerance using redundancy and replication.

	Instruction-FT	**Process-FT**	**RVM-FT**
Replicas	instructions	processes	virtual machines
SoR	processor	application	system
Checks	stores	system calls	I/O operations
Input-Repl.	loads	system calls	interrupts
Isolation	weak	medium	good
Determinism	easy	medium	complex
Reliability	medium	medium	medium
Performance	weak	medium	weak-medium
Recovery	fast	medium	slow

Table 1: Comparison of different software fault tolerance techniques using different replicas

7. CONCLUSION

Since hardware based approaches are too expensive and inflexible to deploy in all technical systems, software-only recovery techniques are gaining popularity. Software schemes are able to detect and recover from many hardware failures, whereas their inherent overhead degrades the performance. Depending which technique is actually used, the performance, reliability and fault coverage level may be different.
All schemes presented in this seminar paper use the key concept of replication and redundancy. Thereby, the basic scheme is always the same: the input is replicated, then the execution is done and last the output is compared in order to use majority voting to detect and recover from hardware faults. However, depending on what is replicated each software technique has its own advantages and drawbacks. Implementing software fault tolerance on a higher level of the software stack tolerates more faults, since faults from lower levels are also covered. Furthermore the flexibility of approaches in higher levels increases and the frequency of output checks is typically lower. On the other hand implementing software fault tolerance on a lower level has a shorter recovery time, as the fault cannot propagate through higher levels. Moreover the effort to ensure replica determinism differ dramatically depending on what is actually replicated. Also which faults are detected and recovered are not the same.
In general, no specific approach is better than the others. It always depends on the actual use case, which variant suits better. For instance, if a complete system should be protected, it might be a good idea to run RVMs on two different servers. If one fails, the other can take over transparently. But, if a single application needs protection, SWIFT-R or PLR might be a better solution.
To put in a nutshell, software centric approaches are the solution whenever it is too costly to apply hardware techniques and reliability up to a high rate is enough.

8. REFERENCES

[1] ECC Memory. `http://en.wikipedia.org/wiki/ECC_memory`, 2014. Online; accessed 27-Oct-2014.

[2] The GNU C Library. `http://www.gnu.org/software/libc/`, 2014. Online;

accessed 26-Oct-2014.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, Oct. 2003.

[4] D. Borodin, B. H. Juurlink, S. Hamdioui, and S. Vassiliadis. Instruction-level fault tolerance configurability. *J. Signal Process. Syst.*, 57(1):89–105, Oct. 2009.

[5] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst.*, 14(1):80–107, Feb. 1996.

[6] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association.

[7] K. Echtle. *Fehlertoleranzverfahren*. Springer-Verlag, 1990.

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[9] S. Mukherjee. *Archtitecture Design For Soft Errors*. Morgan Kaufmann Publishers Inc, 2008.

[10] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *Reliability, IEEE Transactions on*, 51(1):63–75, 2002.

[11] S. Poledna. *Fault-Tolerant Real-Time Systems - The Problem of Replica Determinism*. Kluwer Academic Publishers, 1996.

[12] L. L. Pullum. *Software Fault Tolerance - Techniques and Implementation*. Artech House Publishers Inc, 2001.

[13] G. A. Reis, J. Chang, and D. I. August. Automatic instruction-level software-only recovery. *IEEE Micro*, 27(1):36–47, Jan. 2007.

[14] G. A. Reis, J. Chang, D. I. August, R. Cohn, and S. S. Mukherjee. Configurable transient fault detection via dynamic binary translation. In *IN: PROCEEDINGS OF THE 2ND WORKSHOP ON ARCHITECTURAL RELIABILITY*, 2006.

[15] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 243–254, Washington, DC, USA, 2005. IEEE Computer Society.

[16] U. Schiffel. *Hardware Error Detection Using AN-Codes*. PhD thesis, University of Dresden, 2011.

[17] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, pages 297–306, Washington, DC, USA, 2007. IEEE Computer Society.

[18] D.-I. P. Ulbrich. Redundante Ausführung. `https://www4.cs.fau.de/Lehre/SS14/V_VEZS/Skript/04-Redundanz_hoA4.pdf`, 2014. Online; accessed 02-Nov-2014.