Efficient Fault Tolerance for Operating System Data Structures

Bachelorarbeit im Fach Informatik

von

Robby Zippel

geboren am 11.10.1987 in Gera

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreut durch:

Dipl.-Ing. Martin Hoffmann
Dipl.-Inf. Isabella Stilkerich
Dr.-Ing. Daniel Lohmann

Beginn der Arbeit: 01. Juli 2011
Ende der Arbeit: 23. November 2011

Hiermit versichere ich, dass ich die Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich durch Angabe der Quelle als Entlehnung kenntlich gemacht.

Erlangen, den 23. November 2011 ______________________________

Abstract

Fault tolerance is getting an important field due to the increasing soft error rate of circuits. In the operating system context, the data integrity and the correctness of data structures need to be protected to guarantee a properly running system, if errors occur in the main memory. Simple solutions, such as straightforward checksum algorithms and basic double-linked lists, can give fault tolerance to data, but if a more robust system is required, more complex mechanism with as little overhead as possible must be developed. Therefore, fault tolerance techniques such as an additional pointer and mechanisms to detect and correct errors are added to a double-linked list, which is employed in an embedded configurable operating system for any list operation, which are used, for example, in the scheduling process. That implies a data structure, which is able to detect and correct two changes in its structure caused by errors and, however, has just a minor overhead in operating system case of application like the scheduling process. In the end, the overhead to make operating systems fault tolerant in the scope of this work is very small, so that fault tolerance mechanisms and data structures are a good choice to use in operating systems to be ready for the upcoming problems with transient faults.

Zusammenfassung

Da die Häufigkeit von transienten Fehlern in Schaltungen steigt, wird Fehlertoleranz ein immer wichtigeres Forschungsfeld. Im Kontext von Betriebssystemen muss die Datenintegrität sowie die Korrektheit der Datenstrukturen sichergestellt werden, um einen ordnungsgemäßen Ablauf des Systems beim Eintreten von Fehlern im Hauptspeicher zu garantieren. Fehlertoleranz kann mit einfachen Mitteln, wie einem unkomplizierten Prüfsummen-Algorithmus oder grundlegenden doppelt verketteten Listen, für Daten sichergestellt werden, sobald jedoch ein robusteres System benötigt wird, müssen komplexere Mechanismen, welche aber möglichst wenig Mehraufwand verursachen, entwickelt werden. Aus diesem Grund wurden Fehlertoleranztechniken, wie ein zusätzlicher Zeiger, und Mechanismen um Fehler zu erkennen und zu korrigieren, zur doppelt verketteten Liste hinzugefügt, welche in einem eingebetteten konfigurierbaren Betriebssystem für jede Listenoperation, die zum Beispiel für den Einplanungsprozess verwendet werden, integriert sind. Dies impliziert eine Datenstruktur, die imstande ist, zwei durch Fehler verursachte Änderungen in ihrer Struktur zu erkennen und zu korrigieren, jedoch nur, abhängig vom Anwendungsfall, wie dem Einplanungsprozess, einen geringen Mehraufwand im Betriebssystem verursacht. Letztendlich ist der Mehraufwand für Fehlertoleranz, in dem Rahmen, wie sie in dieser Arbeit entwickelt wurde, in Betriebssystemen, sehr gering. Daher ist es eine gute Wahl solche Fehlertoleranzverfahren und fehlertolerante Datenstrukturen in Betriebssystemen zu verwenden, um auf die kommenden Probleme durch transiente Fehler vorbereitet zu sein.

Contents

CHAPTER 1

Introduction

Every day we are surrounded by technology that affects our lives. Examples are entertainment systems like TVs and cell phones, or "invisible" techniques, which ensure our safety such as the air-bag computer in our cars or embedded medical systems like a pace maker. Especially safety-related technologies are important to be reliable, because if a system fails there may be damage or danger of life, which must definitely be prevented. But electronic systems are vulnerable to environmental conditions such as electronic noise, changes in temperature, and, in particular, radiation. These conditions influence the physical circuits that modern systems consist of. Those are so called *single event effects* which are "undesired effects caused by single energetic particles passing through a given medium." [Che] That means that an energetic particle hits a transistor, which subsequently changes its state. Years ago this topic barely mattered, but with device scaling and super-pipelining as the two major trends of microprocessor technology, the soft error rate increases due to the susceptibility to physical influences [Shi02]. This is also drafted in figure 1.1. *Failures in time (FIT)* describe the rate of soft errors with measuring the number of failures per 10^9 hours of operation. While the soft error rate for SRAMs has always been high, it has been increasing of the order of nine since the last twenty years for circuit logic. Accordingly this affects the number of transient errors the system has to handle.

Such for hardware non-detrimental, unpredictable physical influences can lead to a transient hardware fault, which is a change of the state of a single bit (so called *bit flips*) in the affected electronic component. In this thesis the focus is on bit flips in the system memory and the question how to guarantee that the system will work properly. So there are two cases of possible errors:

On the one hand, it is important to ensure data integrity. Memory can hold important interim results which can, in case of an error, lead to undefined behaviour or even system failure, if the system uses wrong data. The context switching process in an operating system is an excellent example for the necessity of data integrity. If something goes wrong here, the system may crash.

On the other hand, it is important to assure the correctness of data structures which means that the organization structure which is stored in the main memory must be protected. This includes the pointers to nodes in the data structure. If they are changed by a bit flip, this can destroy the correctness of the data structure. One case in which it is

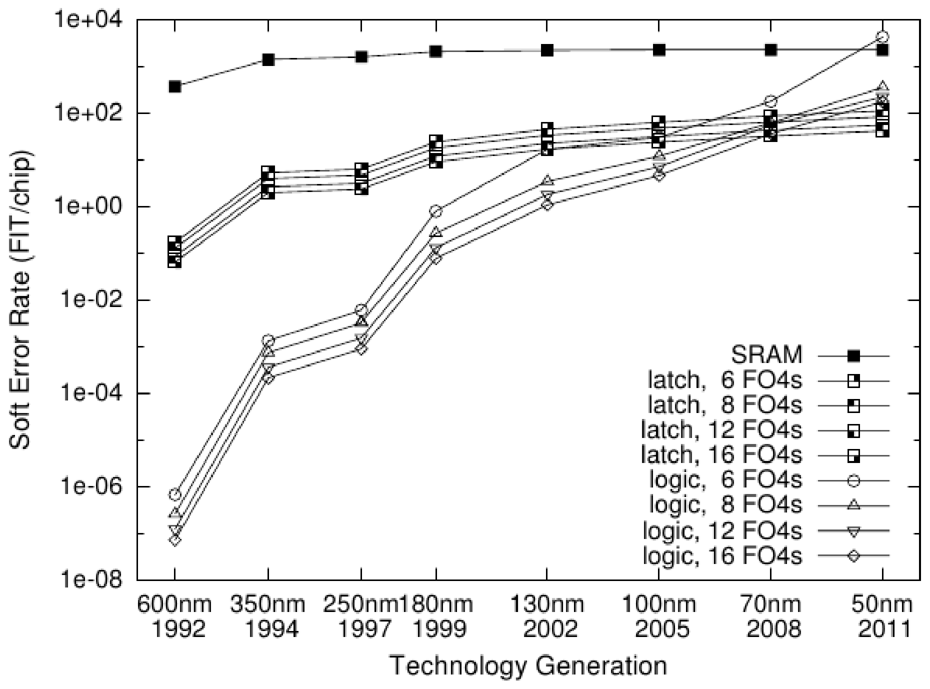

Figure 1.1: Soft error rate per chip for different technology generations [Shi02]

inevitable to have a correct and error-free data structure is the scheduling process in an operating system. If, for instance, an important process gets lost, the system cannot run properly or essential functions will not be executed again until the system is restarted.

The error model applied in this thesis comprises transient hardware faults in memory. Their effects, however, are two-fold: In the data integrity part one or more bit flips can occur and change the data value in just one bit. Yet in the data structure part the point of view changes, as with having the organization address pointers to stay correct, the modification of a pointer is of current interest, independent how many bit are flipped within the pointer address.

Therefore, fault-tolerance methods for data integrity and data structures will be researched based on a real-time operating system. So for a case study fault tolerance techniques are applied to context switching and scheduling process of the embedded configurable operating system (eCos).

CHAPTER 2

Development Environment

Developing fault-tolerance mechanisms for embedded systems, there is the need to design, test, and evaluate on a special purpose operating system. In this thesis, the embedded configurable real-time operating system (RTOS) eCos is used on a special purpose system, based on Intel's Atom architecture. The field of application is the click soft-router, which should be provided with the fault-tolerance property. Click is a modular software router, which puts emphasis on being extensible and highly configurable. Further information is given in [Koh00].

The developed strategies to ensure data integrity and the correctness of data structures are compared in micro-benchmarks. Appropriate testing environments are the context switch and scheduling process of an operating system, in this work the threads of eCos. The operating system will be described in the following section including the operating mode of the context switching and scheduling process.

2.1 The Embedded Configurable Operating System (eCos)

Embedded devices are often limited in their hardware components, particularly the main memory. In this case they need special operating systems such as eCos [Mas03]. A lot of RTOSes, however, bring a full support of different features along, which may not be required depending on the application. This makes the software more complex and even bigger than it has to be, which may require larger and more expensive devices than necessary. With eCos, an open source RTOS for embedded systems, the developer can control run-time components and remove unnecessary ones easily by using the system configuration tool. Required components can be bound from the component repository. So the software's size can be scaled this way from a few hundred bytes up to hundreds of kilobytes depending on the developer's needs.

eCos uses the Component Definition Language (CDL), which is an extension of the existing Tool Command Language (Tcl) scripting language, to make the components configurable. Each section of the repository (hal, kernel, etc.) has at least one CDL script file, describing the package. By editing the script files, it is easy to add own modules, which were included to the repository by the developer, to the configuration tool. The result of the configuration is stored in a file with an .ecc extension. Depending on this file the eCos-tree is built. Furthermore, it is not only possible to pick different packages, but also to choose between various implementations. The kernel scheduler, for example, can be selected as multilevel queue or bitmap scheduler. There are also a lot of adjustments to make such as the number of clock ticks between time slices for the scheduler.

In this way, the developer has full control about the components to use from the repository and their options. Additionally, it is possible to easily add own configuration options for own modules using the CDL scripting language. These are great advantages of eCos towards other RTOSes. This is the reason why it is used in this work and so there is the demand to take a closer look at the context switching and scheduling process in eCos.

2.1.1 Context switching - The significance of data integrity

The ability to run more than one process is a must in modern operating systems. Examples for important use cases are the feature to handle interrupts and the power to let processes wait for an event without occupying the processor uselessly, as the process needs the result of the event to continue. If there are enough processors for all the processes and interrupts, there won't be any problem as every process can run in parallel. But nowadays there are more processes than processors. Especially in embedded systems there is the need for economical components. The manufacturers try to use hardware just as good as really necessary to produce cheaply. Due to the fact that it must be possible to be able to handle as much processes as needed, there is the necessity to change between processes which are interruptible and continuable. Because of this they have to store their current state when interrupted to resume on the intermitted position with the correct condition. The state which needs to be saved contains at the minimum the program counter. If this is all being stored only cooperative processes on the same stack are working correctly. Preemptive systems, in which every process has its own stack, have to save the stack pointer and the working registers as well. One possibility to do this is to push the register values to the stack, save the stack pointer in the thread structure, and switch the context by setting the stack pointer of the processor to the loaded address of the next thread which is about to run. After the switch the state of the next thread must be restored. So the registers are popped from its stack and stored in the processor's registers.

This process of being outsourced and loaded again is transparent for the processes. They keep on working on the values which were buffered in the main memory during their outsourced period. But this is completely built on the correctness of data integrity. If an error occurs to the data on the main memory, the process gets this wrong value and tries to execute the next instruction as it does not know about the interruption. The consequence can be fatal in different ways. If a value changed in a register holding interim results, the final outcome may get wrong during the next computations. The magnitude of this mistake depends on the case of application. It is annoying if a hand-held calculator gives the wrong result, but this is not as serious as if for instance the automatic pilot system on a plain is faulty. But at least the system is still running (assumed there was no division by zero which would end up in an exception) and has the chance to recognize the error on his own its following execution. Nevertheless, there are absolutely essential register values which in case of an error lead to a system crash. The instruction pointer for example holds either the address of the next or currently executed instruction, depending on the type of system. A change of this pointer results in a wrong instruction or to something anywhere in the memory. If this happens the system is not able to continue its work.

With both cases (wrong results and system crashes) being undesirable and must be prevented, methods to ensure data integrity are required and will be considered in the following chapter.

2.1.2 Scheduling - Dependence of correct data structure

Before the system can switch the context, it has to know which process will be next. Generally all threads which are not running at the moment are stored in a list and the head of this list is picked to be the next thread. The strategy of the sort order of the list is determined by the scheduler. It decides what process is next by controlling the threads' positions in the list. Furthermore it provides mechanisms for synchronization during the scheduling and context switching process. In eCos there is a bitmap and multilevel queue scheduler. The last one was used in this thesis, so it will be described in the following paragraph.

Multilevel queue means the ability to have a certain number of priority levels. The number of levels is configurable from 1 to 32 with level 0 being the highest and level 31 the lowest priority. For each level there is one queue stored in an array of all scheduling queues. So threads get their priority and are hold in the appropriate list. If preemption is allowed a thread with lower priority can be displaced by one of a higher level. Within a priority level the scheduler allows timeslicing. That means that a thread can execute for a special amount of time, which is also configurable by the developer. In figure 2.1 an execution process of three threads (one with low (*Thread C*) and two with high priority (*Thread A* and *Thread B*)) is shown. The current executing process Thread C is displaced by Thread A, which runs until its timeslice ends. After that the process within the highest prioritized not-empty queue is chosen to run next, which is Thread B in this case. After descheduling Thread B, Thread A is the highest prioritized process to run because it was rescheduled after its timeslice ended. When Thread A is also descheduled Thread C is going to be next. Although it is the process with the lowest priority, it is the only thread which can be scheduled at the moment as no other thread is in any queue.

Double linked circular lists are used for the scheduler's queue implementation. As it will be shown in the following chapter this is already a fault-tolerant data structure. But there is the need for more robust data structures as soft error rate will increase in future as seen in the introduction part.

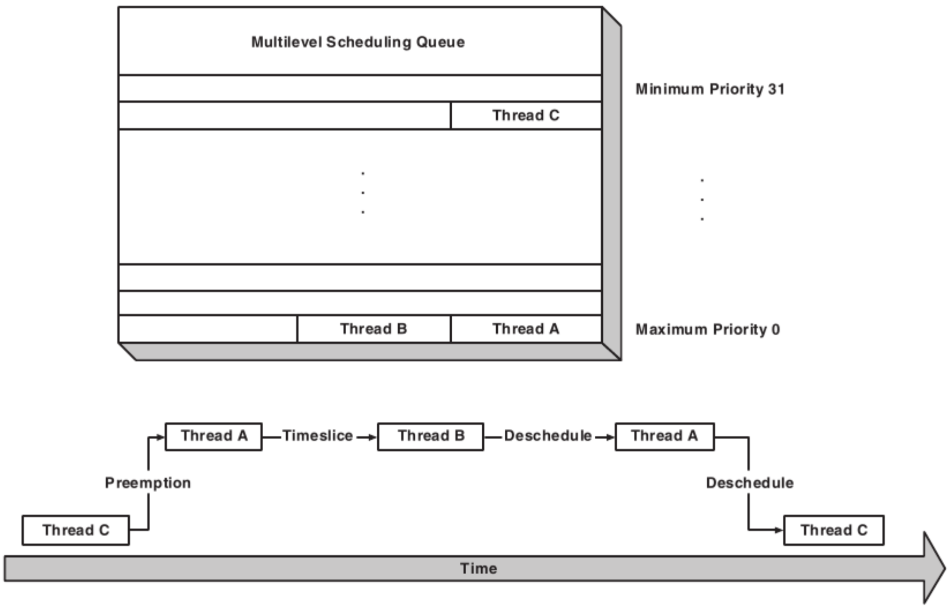

Figure 2.1: Thread executions with multilevel queue scheduling [Mas03]

CHAPTER 3

Fault-tolerant data integrity and data structures

Fault-tolerance mechanisms ensuring data integrity and the correctness of data structures are compared and analysed against newly developed and already existing implementations. First, data content is to be protected against transient errors using various methods, and, secondly, a comparison between these mechanisms applying the eCos context switching is made. Afterwards the main subject of this thesis, the fault-tolerance of data structures, is researched. The question, how an additional pointer can improve fault-tolerance and how this influences the overhead, will be resolved.

3.1 Data integrity

The focus in this section is on the integrity of data content. The aim is to get a transparent fault checking and, if necessary, correcting, which has as little overhead as possible. Therefore, different checksum implementations are compared against each other and also against an error correcting code.

3.1.1 Checksums

One possibility to detect errors is checksums. These are no correcting codes meaning the redundant data must include the entire content, because if an error occurs there is no way to identify, neither in which word the error occurred nor which bit flipped. After checking the data, there is only the guarantee if an error arose or not. Faulty data implies the whole replacement of the concerned section by the redundant data. Consequently this means that the storage overhead is at least 100%. With the result of the checksum has to be stored as well, the overhead is even higher.

In this work two kinds of checksums are discussed: On the one side one straight forward implementation of a simple checksum and on the other side cyclic redundancy codes.

Add up - a simple checksum

The algorithm generates a sum over the stored content in scope. To say for sure that there was an error only one bit error over the whole context is acceptable, because if in one word a bit changes from 0 to 1 and in another word the same bit flips vice versa, the result is the same checksum and, by implication, the error remains undetected. But in best case all bit flips occur in different bits. In this case the algorithm can detect thirty-two bit errors.

Cyclic Redundancy Check

Using cyclic redundancy codes is a common method to detect errors. As many chipsets have no special CRC generation circuits, software-based CRC generation is important. A lot of work has been done in the past to accelerate the CRC generation process. [Kou] Today the most frequently used algorithm is the implementation by Sarwate [Sar88]. Intel developed a faster implementation called *slicing-by-8* [Kou]. In Intel's Nehalem-based Intel Core i7 the *SSE4.2* instruction set was first implemented. It includes a *CRC32* instruction that speeds up the checksum generation by using hardware support. These three different implementations are compared to each other.

Basics

"CRC algorithms treat each bit stream as a binary polynomial $B(x)$ and calculate the remainder $R(x)$ from the division of $B(x)$ with a standard 'generator' polynomial $G(x)$." [Kou] This calculation is done in module-2 arithmetic. In other words the additions and subtractions are equal to the exclusive OR (XOR). The calculations are free of carries which simplifies the technical realization as only XOR gates are necessary. Additionally this increases the speed of the computation. [Her06] "The remainder that results from such long division process is often called CRC or CRC 'checksum'" [Kou]. The receiver[1] verifies the remainder by calculating it using the bit stream and performing an exclusive OR against the transferred remainder. If the result is null, the bit stream is valid.

Working on one bit of the dividend at the same time is very slow. So, effective methods to perform the long division were developed: The current remainder that results from a group of bits is pre-calculated and stored in a table. At run time all possible remainders are already pre-computed. So one table lookup step can replace several long division steps [Kou].

The Sarwate Algorithm

First the CRC value is initialized to a given value depending on the implementation. For instance for *CRC32c* this number is 0xFFFFFFFF. In listing 3.1 the initial value is *INIT_VALUE*. Before performing the CRC algorithm a 256 entry lookup table with the results in it was pre-calculated. The algorithm now walks through every single byte of the input and XOR-s the read byte from the input stream with the least significant byte of the CRC value. The result is 8 bits which give the index of the lookup table to get the result. After that a XOR operation between this value and 24 most significant bits of the CRC value, after shifting them 8 bits to the right, is accomplished. The outcome is the CRC value which will be used for the next step of the loop. The calculation is finished when all bytes of the input have been taken into account [Kou]. At the end the calculated CRC value is XOR-ed with a final value.

1 In our case the receiver is the same system, even the same class. The content has to be checked against soft errors occurred while outsourced.

Listing 3.1: The Sarwate Algorithm [Kou]

```
crc = INIT_VALUE;
while(p_buf < p_end)
  crc = table(crc ^ *p_buf++) & 0x000000FF) ^ (crc >> 8);
return crc ^ FINAL_VALUE;
```

[Sar88] contains detailed information about the developing of this CRC algorithm.

Intel's *slicing-by-8*

Intel developed a faster implementation of a CRC algorithm and presented it in 2005. It takes 2.2 cycles/byte [Jog05] instead of Sarwate algorithm's 7 cycles per byte[Kou]. The algorithm works on 64 bits at a time, while standard implementation only takes eight bits, by using eight lookup tables. With modern processors having larger cache units they are able to handle moderate size tables. The slicing-by-8 algorithm has a total space requirement of eight kilobytes. So it fits into the processor cache and results into faster query results, especially by having eight queries per loop which is the reason for the increase of speed.

The algorithm starts like the Sarwate algorithm with a 32-bits initial value. This value is XOR-ed with 32 bits of the input stream and the two least significant bytes of this result are both indexes for two different tables, of these two values are picked and an XOR operation is performed between them. This is done again with the two most significant bytes but in turn with different tables. So, four different tables are used by now. At this moment there are two results which must be XOR-ed again. After that another XOR operation must be executed. But at first the second value of this XOR operation must be calculated. To do this the next 32 bits of the input stream are used without XOR-ing them with the initial value. Same story as above: all four bytes are looked up in a table, but this time yet other four tables than used for the first four bytes, and then XOR-ed. Following that, an XOR operation is performed between this interim result and the outcome of the first four bytes.

This is done in a loop until all bytes are read and calculated into the interim CRC value. After the loop this number is again XOR-ed with a final value. After all the CRC computation is done.

SSE4 instruction *crc32*

The *streaming SIMD extension* (SSE) is an extension of the x86 instruction set. SSE in version 4.2 was first implemented on Intel's Nehalem architecture and with this a *crc32* instruction [Sin].

3.1.2 Comparing redundant data

A very fast solution is the direct comparison between original and redundant data. The actual data is stored redundantly without any checksum or correction bits. When checking if an error occurred the algorithm passes every word and compares them. In the case of inequality an error is detected. But the big disadvantage and killing stroke is the lack of knowledge which of the two words is incorrect. So this proceeding is useless.

3.1.3 Hamming code - an error-correcting code

The hamming code is an error-correcting code developed by Richard Hamming. With a hamming distance[1] of three it can detect two bit errors and correct single bit errors. The *(7,4)-Code* is the simplest hamming code. It is a length of seven, of which three bits are correction information, the so called parity bits.

Implementation

In case of guaranteeing data integrity of a scope of 32 bit words, a hamming code for every word in this range needs to be generated. With having 32 bit words, six parity bits are needed to get a hamming distance of three to be able to correct single bit errors per word. It is also possible to recognize two bit errors, which leads the system to throw an exception, which can be handled by the user.

The output is a 38 bit word. Therefore, the memory has to be restructured after building the hamming code to store the generated word. The reason is that the system is 32 bit aligned[2]. The consequence would be high effort on restructuring associated with a waste of memory, as 28 filling bits must be stored per word because of the alignment. As a solution the calculated parity bits are stored separately from the data words.

To build the parity bits, the '1's at special positions in a word must be counted. This is done by performing a bitwise AND with the associated mask (e.g. p[0] = (v & 0x56aaad5b) for the first parity bit). In figure 3.1 the source of the associated mask is illustrated.

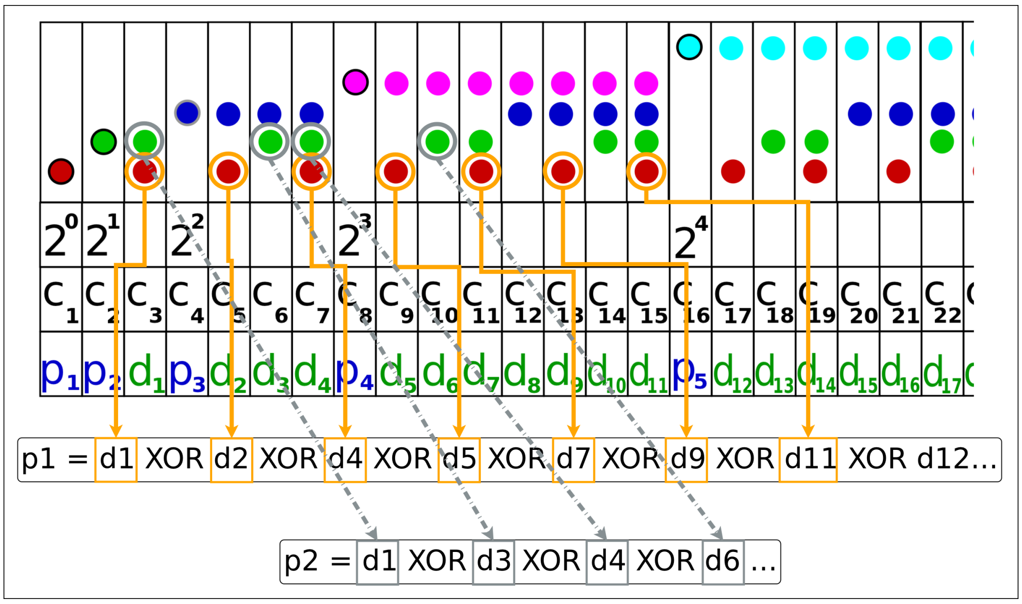
2^0 2^1 2^2 2^3 2^4
C_1 C_2 C_3 C_4 C_5 C_6 C_7 C_8 C_9 C_{10} C_{11} C_{12} C_{13} C_{14} C_{15} C_{16} C_{17} C_{18} C_{19} C_{20} C_{21} C_{22}
p_1 p_2 d_1 p_3 d_2 d_3 d_4 p_4 d_5 d_6 d_7 d_8 d_9 d_{10} d_{11} p_5 d_{12} d_{13} d_{14} d_{15} d_{16} d_{17}
p1 = d1 XOR d2 XOR d4 XOR d5 XOR d7 XOR d9 XOR d11 XOR d12...
p2 = d1 XOR d3 XOR d4 XOR d6 ...

Figure 3.1: Building parity bits

1 The hamming distance is the smallest number of binary digits which sets one word apart from another.
2 Every n bit system has the same problem because the resultant words are of $n + p > n$ length.

Optimizing the population count

The population count of a word is the number of one bits in the value. The simplest type to perform this is a loop checking every bit successively. And this *ones count* slows the hamming code strongly down. This is shown in table 3.4: Just changing the population count function affects the runtime. Therefore, additional optimizations on this instruction were implemented and tested.

SWAR population count algorithm

The following algorithm was developed by *The Aggregate*, a group of researchers based at the University of Kentucky. The algorithm executes a tree reduction adding the bits in a 32 bit value. Further information is given in [Die].

Listing 3.2: Population Count using a variable-precision SWAR algorithm[Die]

```
unsigned int ones32(register unsigned int x)
{
  x -= ((x >> 1) & 0x55555555);
  x = (((x >> 2) & 0x33333333) + (x & 0x33333333));
  x = (((x >> 4) + x) & 0x0f0f0f0f);
  x += (x >> 8);
  x += (x >> 16);
  return(x & 0x0000003f);
}
```

Array with precomputed count of ones

Another possibility is the pre-computation of ones in a word. The word is separated in 8 bits. Getting the population count for 8 bits is done in $O(1)$.

Listing 3.3: Precomputed ones for 8 bits words in a table.

```
unsigned int ones32(register unsigned int x)
{
  unsigned int count = 0;
  for(int q = 0; q < 32; q += 8) {
    count += __popcount_tab[(x >> q) & 0xFF];
  }
  return count;
}
```

Sparse Ones

This algorithm lasts the number of ones in the word loops.

Listing 3.4: Spares Ones algorithm.

```
unsigned int ones32(register unsigned int x)
{
  unsigned int count = 0;
  while (n) {
        count++;
        n &= (n - 1);
    }
    return count;
}
```

SSE4.2 instruction *popcnt*

The SSE4.2 instruction set, first implemented in Intel's Nehalem architecture, includes the *popcnt* instruction which counts the number of ones in a word.

3.1.4 Micro-benchmarking fault tolerant eCos context switching

Including fault-tolerance mechanisms slows systems down, because additional checks have to be made and extra memory is used for redundant data and checksums or parity bits, depending on the fault-tolerance strategy. The question is, what about the time and memory overhead, if it is necessary to use fault-tolerance to ensure data integrity. This will be figured out in the following.

Using an Intel Atom system, Intel's read time-stamp counter instruction (RDTSC) can be used for time measurements for evaluations. The basics and implementation is described in [Coo97]. A very important thing to know is that Intel's RDTSC measures cycles and not time. This is working with Intel processors as it is possible to access their time-stamp counter, which is a 64-bit model specific register incrementing every clock cycle. To calculate the time in seconds the measured process took is the number of cycles divided by the frequency of the processor in Hz.

Applying this technology, it is not possible to use the SSE4.2 instructions. Nevertheless, these instructions were also measured on an Intel i7 870 CPU with SSE4.2 support to get to know what is potential when using hardware supported commands. After the scheduler made the decision which thread is next, the context is going to be switched. For the test case of one bit errors, fault injection was done statically after the checksum generating process by flipping a bit of the instruction pointer stored in the main memory. The context switching process is to be measured using the RDTSC instruction. Average is the mean value over a thousand measurements. To calculate the overhead the reference is the average value of *without FT* with *O3* optimizing. The results are shown in the tables 3.2 to 3.5 and will be discussed now.

First of all the additional overhead for storing and restoring data redundancy was measured. It is listed in the table 3.1. The results without optimization are not worth mentioning. As the hamming code was developed without the optimization to store the parity bits consecutively instead of the six parity bits per storage location of four bytes, it is the same overhead as the checksum algorithms.

The simple checksum (in the tables called *FTChecksum*) benefits from its simple algorithm. This makes it the fastest one in the tests. While having only 38% overhead in running time when no error occurs, it increases slightly to 47% when a bit flipped. But this algorithm has disadvantages. The one thing is, that, and this is a problem of all checksum algorithms, the memory overhead is more than 100%. With checksums not being correcting codes, it is necessary to make the whole context, which must be

System	Operation	Optimization	Average
i7	store	O0	282
		O3	71
	restore	O0	264
		O3	71
	hamming-repair	O0	79
		O3	51
Atom	store	O0	401
		O3	136
	restore	O0	317
		O3	124
	hamming-repair	O0	127
		O3	103

Table 3.1: Overhead of storing and restoring redundant data and for the repairing process of the hamming code.

Algortihm	Compiler	Minimum	Maximum	Average	Overhead
without FT	-O0	400	960	412	0%
	-O3	400	540	412	0%
FTChecksum	-O0	970	1100	984	139%
	-O3	550	680	570	38%
FTCompare	-O0	1070	1270	1086	164%
	-O3	580	1570	616	50%
FTHamming	-O0	98610	99570	99057	23943%
	-O3	36170	36800	36307	8712%
	-O3 -DHAMMOPT	11480	12340	11553	2704%
	-O3 -DHAMMOPT1	26630	27750	26767	6397%
	-O3 -DHAMMOPT2	15110	17850	15257	3603%
FTSarwate	-O0	3120	3320	3138	662%
	-O3	1360	1640	1385	236%
FTsb8	-O0	2200	3240	2229	441%
	-O3	1120	2010	1149	179%

Table 3.2: Switching context without errors on Intel Atom - performance test.

Algortihm	Compiler	Minimum	Maximum	Average	Overhead
FTChecksum	-O0	1250	1900	1264	207%
	-O3	590	1150	605	47%
FTCompare	-O0	1060	1310	1077	161%
	-O3	580	1240	609	48%
FTHamming	-O0	98850	99810	99131	23961%
	-O3	36210	36950	36336	8719%
	-O3 -DHAMMOPT	11550	12530	11625	2722%
	-O3 -DHAMMOPT1	26710	27890	26839	6414%
	-O3 -DHAMMOPT2	15130	18010	15324	3619%
FTSarwate	-O0	3420	3780	3448	737%
	-O3	1390	1690	1418	244%
FTsb8	-O0	2480	3260	2505	508%
	-O3	1160	2270	1205	192%

Table 3.3: Switching context with 1 bit errors on Intel Atom - performance test.

protected, redundant. Additional to this the calculated checksum has to be stored as well. In this thesis a special purpose system is used which has no problem with making a lot of redundant data due to the fact that it has enough memory. But the memory overhead can get a disqualifier in small embedded systems. The other one, as already shown, is that only one bit of the entire area, which must be checked, may change to say for sure that an error occurred.

In more complex checksum algorithms like the Sarwate Checksum the last problem is solved by generating a checksum for every word and using this one again for the next checksum value. But this algorithm is slower, having a runtime overhead of 236% in case of no error and even worse when an error occurs. That is why another try was made, to speed up a better checksum algorithm than the simple add up. Therefore, the *slicing-by-8* algorithm was measured. Indeed it is much faster than Sarwate's version, but with 179% overhead still much slower than the simple checksum algorithm.

Finally there is the question whether to use a very fast implementation with the known disadvantages or a more robust one with the consequence of a slower system. The advance towards the fast implementation is justified by the knowledge of transient errors being very rare and most of the time the system does not need any fault-tolerance mechanisms. So it is required to get fast through the generating and checking process. Another reason for the add up implementation is that two errors occurring is even less frequently.

To get rid of the memory overhead problem, an error correcting hamming code can be used. Only six bits per word must be stored which means an 18.75% memory overhead. But in result table 3.2 shows hamming codes to be very slow. The problem is the population count, which was already mentioned in subsection 3.1.3, and also table 3.1 shows that the

Algortihm	Compiler	Minimum	Maximum	Average	Overhead
without FT	-O0	128	528	396	2%
	-O3	120	544	388	0%
FTChecksum	-O0	476	812	743	91%
	-O3	208	552	480	24%
FTCompare	-O0	528	920	805	107%
	-O3	188	668	458	18%
FTHamming	-O0	58720	61424	60353	15455%
	-O3	18788	19712	19274	4868%
	-O3 -msse4.2	**796**	**1180**	**1074**	**176%**
	-O3 -DHAMMOPT	5648	6252	5950	1434%
	-O3 -DHAMMOPT1	11236	12236	11566	2881%
	-O3 -DHAMMOPT2	7656	9896	8059	1977%
FTSarwate	-O0	1692	2044	1955	403%
	-O3	896	1236	1166	201%
FTsb8	-O0	1088	1804	1344	246%
	-O3	452	1152	715	84%
FTcrcSSE	-O0 -msse4.2	760	1080	1012	161%
	-O3 -msse4.2	220	620	484	25%

Table 3.4: Switching context without errors on Intel i7 - performance test.

correction process is very fast. Even software optimizations for the population count do not result in acceptable overhead.

Measuring the error correcting hamming code with different population count optimizations is given in the tables as compiler flags *-DHAMMOPT*. The *HAMMOPT* flag leads to variable precision SWAR algorithm, the *HAMMOPT1* flag to algorithm with the precomputed array of counted ones, and finally the *HAMMOPT2* flag to spare ones algorithm.

It is necessary to use hardware based instruction in terms of the SSE4.2 *popcnt* command to get a runtime which is comparable to the one of the Sarwate checksum. The measurements for this instruction were made on an Intel i7 architecture and as table 3.4 shows, it is possible to get an error correcting code with only 176% overhead. Unfortunately this instruction is not usable on smaller CPUs like the Atom architecture, so own hardware support of the population count for optimization can be considered.

A second hardware based instruction in the SSE4.2 instruction set is the CRC command to build a checksum. More complex checksum algorithms are quite slow as indicated above, but when the SSE instruction is used, the same speed can be reached as the simple

Algortihm	Compiler	Minimum	Maximum	Average	Overhead
FTChecksum	-O0	696	1036	976	152%
	-O3	256	548	501	29%
FTCompare	-O0	540	884	830	114%
	-O3	192	636	462	19%
FTHamming	-O0	593336	61744	60650	15531%
	-O3	19028	20280	19595	4950%
	-O3 -msse4.2	**808**	**1260**	**1078**	**178%**
	-O3 -DHAMMOPT	5760	6360	6002	1447%
	-O3 -DHAMMOPT1	11532	12560	11882	2962%
	-O3 -DHAMMOPT2	7984	9776	8291	2037%
FTSarwate	-O0	1904	2244	2174	460%
	-O3	916	1256	1188	206%
FTsb8	-O0	1316	2404	1564	303%
	-O3	468	1244	732	89%
FTcrcSSE	-O0 -msse4.2	996	1300	1234	218%
	-O3 -msse4.2	268	1364	843	117%

Table 3.5: Switching context with 1 bit error on Intel i7 - performance test.

checksum. Only when a bit flips and correction must be done, the algorithm using the SSE CRC command is very slow (117% overhead). Mostly, however, no error occurs, so this instruction is an alternative solution to the simple checksum with fewer disadvantages. Regrettably this instruction is also not available on Atom CPUs.

Finally, for the goal to ensure data integrity while being robust against single bit flips, a very simple checksum implementation is sufficient as this is a very fast solution. Utilizing more robust implementations or even an error correction code slows down the system so much that it gets impractical. Hardware based implementations give a solution to this problem, but are not available on all processors, which may lead to the consideration of building own hardware support. This may be a topic of future work to optimize the data integrity process.

3.2 Correctness of data structure

With the view to making data structure more robust against soft errors, the first idea is to increase the number of pointers. Indeed, this results in a more detectable and correctable data structure. The question, however, is on the overhead, which will be examined in this section. Furthermore, it will be considered if just adding pointers is the best solution to get more fault tolerance or if it gets too unprofitable and if other solutions might be better.

3.2.1 Types of errors

The focus is on the pointers and identifier fields of a data structure and not on its content. With changing a pointer due to single event upsets, different types of errors can occur.

Changing the identifier field is causing the system not to recognize a valid instance and may mislead to the assumption that a pointer was modified to a memory address no instance is stored. If it is possible to detect the identifier field as wrong, it is easily repaired by setting the correct value.

If a pointer is changed, two possible occurrences can arise. On the one hand, the modified pointer can lead to a valid, but different instance. In this case there must be enough redundant data to reconstruct the real structure. Otherwise one or more nodes can get lost like it is shown in figure 3.2. On the other hand, the memory address does not contain valid instances or the pointer even leads to an invalid address. This can provoke a trap if there is a memory protection unit (MPU) in the system. If so the reconstruction activity has to be done in the exception handler, as the same instruction is executed again once the handler returns. Another possible trap is thrown if the system requires aligned addresses and the pointer's last bits are changed to an unaligned address. If the system is not throwing any traps, an undefined word has been dereferenced and the data structure needs to be recovered.

3.2.2 Detectability and correctability

To determine the theoretical detectability and correctability of data structures some propositions are necessary.

"A k-determined storage structure is (at least) (k-1)-detectable"[Tay80b]. To use this theorem it is important to define that "a data structure is k-determined if the pointers in each instance of the storage structure can be portioned into k disjoint sets, such that each set of pointers can be used to reconstruct all counts, identifier fields, and other pointers" [Tay80b].

In theorem 7, the general correction theorem, it is said: "If a storage structure employing identifier fields is 2r-detectable and there are at least r + 1 edge-disjoint paths to each node of the structure, then the storage structure is r-correctable" [Tay80b].

3.2.3 Basic double-linked list

A simple linear list, in which each node has a pointer to its next node, is 0-detectable and 0-correctable [Tay80a]. The detectability can be incremented by adding an identifier field to each node and replacing the null pointer at the end of the list by a pointer to the head. Additionally a count field, which contains the number of nodes in the list, can be added. But adding this features has no effect on the correctability, which is still nought [Tay80a].

In double-linked list each node has an additional pointer to its predecessor. Now we have a 2-detectable and 1-correctable data structure, which has (as a disadvantage of adding organization structure to the list) an increased number of changes for manipulating the list. So inserting costs six changes (two forward pointers, two backward pointers, an identifier field, and the count) instead of two (the pointer of the inserted and the previous node), the simple linear list has to proceed [Tay80a].

3.2.4 DanceList - A triple-linked list

The comparison for our DanceList is a double-linked list as described in section 3.2.3. With the need to handle single event upsets, it is necessary to check the data structure every time we change it, e.g. inserting a node, because an error can occur any time and so a cyclic validation is insufficient, as an error may arise right after the validation. Therefore, it is very important that the time required for the checking process and manipulation is kept minor, while increasing fault tolerance.

The first step is the removal of the count field. To validate the count means running through the whole list and compare the number of nodes against the count. This is very costly when it's done every time the data structure is changed; additionally with regard to the fact that mostly no error occurred between two changes. The result is the decrement of the detectability by one. This fact is simply proven in figure 3.2. If two pointers change so that a node is not linked anymore, and the changes lead to a correct data structure, the fault is not recognizable. But, furthermore, it decreases the number of changes to five.

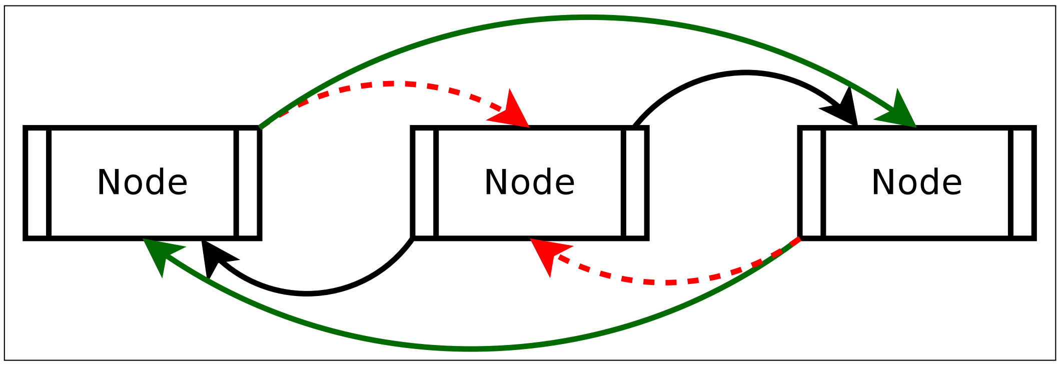

Figure 3.2: Double-linked list with two pointers (red ones) are changed to the green ones. The node in the middle is unrecognisably lost

In [Shi02] it is shown that the soft error rate increases with smaller chip designs. With adding an additional pointer to every node that points to the node after next, we want to get better results in detectability and correctability.

Detectability

In the DanceList the forward, the backward, and the pointer to the after-next node may be used to reconstruct an instance. Figure 3.3 shows the three disjoint sets that can be used to reconstruct the list. Therefore, it is 3-determined and with no stored count the DanceList is 2-detectable.

With the admission of more than one simultaneous error, detecting and correcting the data structure becomes more complex. A small example is the checking of the next pointer: To verify that the previous node of the next instance is the starting node again is insufficient as both pointers can be changed as in figure 3.2. More tests are necessary to encircle the correct broken pointers.

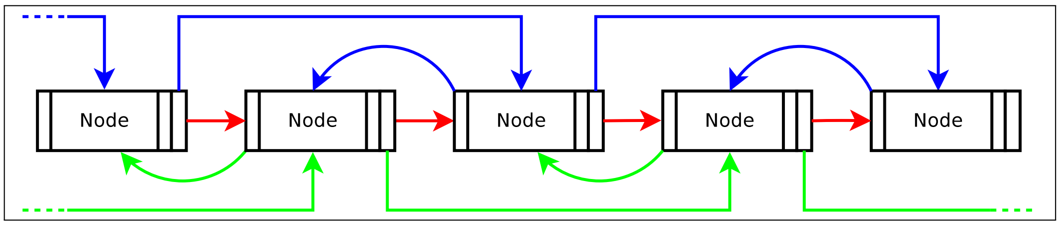

Figure 3.3: Portioning the DanceList in three disjoint sets of pointers. The green, red, and blue ones can be used to reconstruct the entire data structure. So the DanceList is 3-determined.

Detecting a broken pointer

When a broken pointer is leading to undefined memory[1] it is simple to detect as checking the identifier field will show up that the pointer doesn't point to an instance of the data structure. On the other side the next pointer can lead to another node. It cannot be said for sure if it is really the following one. Furthermore it is uncertain if one or two errors have occurred. There is the need to check all possible error constellations.

To do this there is the opportunity to go through a complex path of *if* and *else* instructions to enclose the erroneous pointers. The idea of the beginning of the detecting procedure as one example for the many possibilities is described in the following:

At first it is checked if the previous pointer of the following node is pointing back to the node checked. If the test is true, it is still possible that both pointers changed as in figure 3.2. Therefore, an additional check is needed: Is the after-next node's previous pointer leading to the observed instance's next one? In case yes, the four checked pointers are correct, but the previous pointer of the watched node, however, is uncertain. If not it is to be clarified which pointers were faulty. So, in case of an error, it is known now that there has been an error, but it is neither clear which pointer is broken nor if there were two pointers changed. The other possibility is that the check above (previous pointer of following node pointing back) returns false. There are a lot more options of error constellations. Unsure if the next pointer, the previous pointer of the following node, or even both are broken, it is indeed possible that another pointer than that mentioned were changed. Finally, there is a great number of comparisons to find out which pointer was or which pointers were changed.

This will end up in a lot of compare and jump instructions on assembly level. As a consequence a different kind of proceeding is to be used. Including three pointers per node, the number of possible errors is bounded above. The main idea is to check the data structure within several comparisons and get the changed pointers as the result. Optimally all errors are reliably predictable with as few tests as possible.

Not all combinations of faulty pointers are of interest. Specific pointers are needed to correct the data structure while others are irrelevant, because firstly they are not used for correction and secondly it doesn't matter if they are wrong as they will be set - if a new node is inserted - to another node. So the old (wrong) value of the node is negligible. After all it depends on the strategy of the correcting function what fault combinations are

1 Still on the assumption there are no traps when touching undefined memory.

relevant.

Correctability

As shown above the DanceList is 3-determined, and so the correctability is one consequently. This is the general correction theorem's result, which was introduced in section 3.2.2. But the following drafts will prove that the DanceList is indeed 2-correctable. To get rid of one of the two errors is covered below. After that step there is just a single error, which can be corrected in a next step.

Correcting the next pointer If the next pointer is faulty, there are two ways to correct it, depending on the second incorrect pointer. The two cases are visualized in figures 3.4 and 3.5. The red solid arrow demonstrates the broken next pointer. Pointers, which can also be corrupted, are red dotted. Finally, the correction of the next pointer is done by the green path.

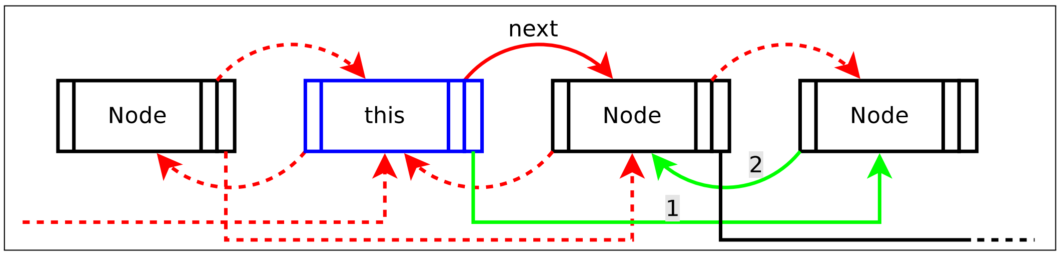

Figure 3.4: Way to correct the next pointer (red solid arrow) with the after next and the previous pointer of the after next node are faultless.

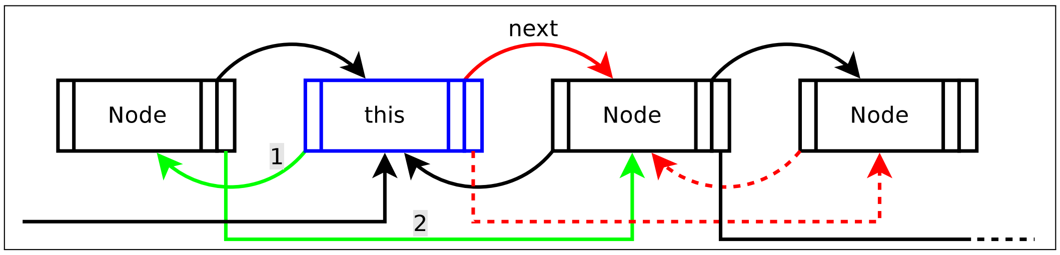

Figure 3.5: Way to correct the next pointer (red solid arrow) if the after next or the previous pointer of the after next node is faulty.

So in the detecting procedure it is important to figure out if in addition to the next pointer the after-next or the previous pointer of the after-next node is broken. In this case the next pointer has to be corrected like in figure 3.5. As the after-next node's previous pointer is irrelevant for the list's manipulation process at the observed node's location, it does not have to be corrected. But not all pointers can be left unappreciated. Specific pointers are important for the insertion and removal process. So the previous node's after-next pointer has to be set correctly when processing the list. That is why an error-free reference to the previous node is needed. A faultless pointer to the next node is necessary, because, first of all when inserting nodes, the next node's pointers will be changed and this node is also a reference for pointers of the inserted node. This is also the reason why the after-next pointer must be correct. In figure 3.8 the nodes, which pointers, and how they are changed, are drafted. If the after-next and after-next node's previous pointer are faultless, they are used to correct the next pointer. The fact that not all pointers are demanded to be corrected apart from the next pointer also applies here. When two errors occur and neither the previous nor the after-next pointer is the second fault it is not necessary to distinguish which of the pointers were changed in addition to the next pointer.

Correcting the after-next pointer In the preceding paragraph it was ensured that the next pointer is faultless or corrected, so it is not necessary to bother about this pointer any more. Furthermore, there are two cases again but this time only depending on the correctness of the next pointer of the next node. This is outlined in figures 3.6 and 3.7.

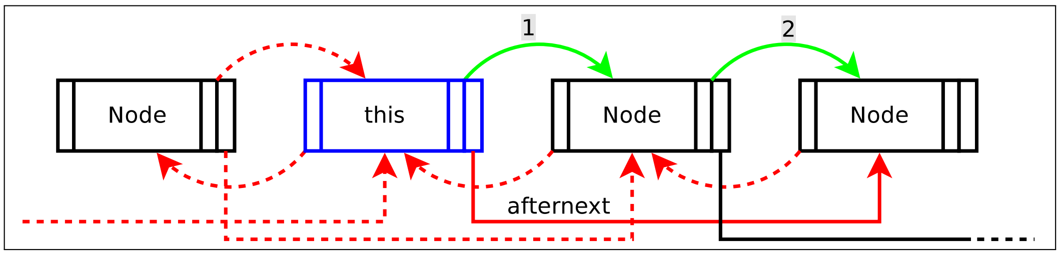

Figure 3.6: Way to correct the afternext pointer (red solid arrow) with the next and the next pointer of the next node are faultless.

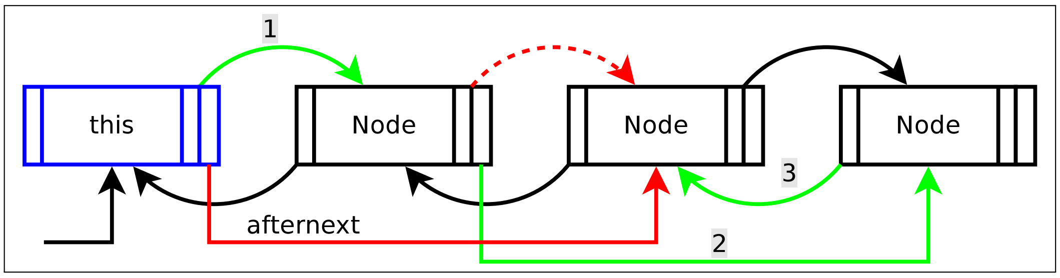

Figure 3.7: Way to correct the afternext pointer (red solid arrow) if the next pointer of the next node is faulty.

Just in case the next node's next pointer is broken the list has to be corrected like shown in figure 3.7. In all remaining cases the correction is done with the first option in figure 3.6. As a special case the previous pointer is faulty in addition to the after-next pointer. This alternative must be detected so as to have the possibility to correct the previous pointer after rectifying the after-next pointer without another test. With this knowledge the number of error constellations which need to be distinguished decreases. Because of this it is the duty to find appropriate comparison functions which was also shown in the paragraph about correcting the next point above.

Correcting the previous pointer On the one hand, the next or after-next pointer can be corrupted in addition to the previous pointer. This was already handled in the two paragraphs above. After correcting the concerned pointer there is only the previous pointer left. With this pointer being the only way to go backwards through the list, it is needful to walk forwards until reaching the correct reference of the previous pointer to set it right. Going through the complete list is inevitable because the previous node is unknown when the pointer is broken. As more than one change can occur to the data structure, we are forced to make sure with every step, that the instances and pointers passing through are not faulty. Of course this increases the costs and time running through the data structure by the overhead of the checking and as the case may be correcting procedures. But as an advantage the list is checked completely and possible errors were reduced in the entire data structure. As an optimization the list can be passed through by the after-next pointer. Now it is not necessary to pass n but $\frac{n}{2}$ nodes. On the other hand, other error constellations than previous plus next or after-next pointer are not of any interest. The only thing that is important is to detect the broken previous pointer. The reason is that the other pointers are not needed for the correction. The after-next pointer was already checked and with every step through the list the following after-next pointer is checked again.

Results

Not all error constellations are important for the correcting process. So the number of different options which have to be distinguished decreases. The goal is to find perfect fitting comparison functions. This is the opportunity for optimization to get detection done in as little as possible steps and consequently as fast as possible. After construction of the n functions there is a bit mask with n bits. To call the correction function in $O(1)$ an array of references of the fitting correction functions were structured. The correction function is located in the array field with the number of the resulting bit mask.

Listing 3.5: Error detection using comparison functions

```
void DanceList::ft(Node *node) {
  int errcode = 0;

  node->afternext == node->next->next ? errcode = errcode | 1 : errcode;
  node->next == node->afternext->prev ? errcode = errcode | 1 << 1 :
      errcode;
  node == node->next->prev ? errcode = errcode | 1 << 2 : errcode;
  node == node->prev->next ? errcode = errcode | 1 << 3 : errcode;
  node == node->prev->afternext->prev ? errcode = errcode | 1 << 4 :
      errcode;
  node == node->afternext->prev->prev ? errcode = errcode | 1 << 5 :
      errcode;
  node->afternext == node->prev->afternext->next ? errcode = errcode;

  if(errcode == MAXERROR) return; // Everything's fine

  // call correction function
  function[errcode](this, node);
}
```

In listing 3.5 the functions used in this research are drafted. But they are not perfect: It is not possible to decide which correction mode (figure 3.6 or 3.7) to use when the after-next pointer is broken. The problem is that all pointers which are involved are needed to check against the others, meaning the next node's after-next pointer must be used to check the next node's next pointer and vice versa. So it is only possible to detect that one of the three pointers (the next node's next, after-next, and the next node's after-next node's previous pointer) is broken. That is the reason why this case is handled as one and has no extra comparison function. To solve this issue the next node is checked first. Changing the point of view the broken pointer can be detected and corrected as described above, because now it is the node's own next or after-next pointer or the after-next node's previous pointer. And this error is detectable and correctable. After fixing this problem the actually checked node respectively its after-next pointer can be repaired. Of course this method has more overhead with a second node to be checked. This is something to be optimized in future works.

Summarized using triple linked list and comparison functions gives following advantages:

- 2-correctable data structure which implies more robustness against transient errors.
- Detection of a second error, if any, while detecting the first one in a fast way.
- Correction of a forward pointer is done in $O(const.)$ and not $O(n)$ like in double-linked lists.

But there is also more overhead because of the additional pointer, which increases the number of changes when the list is manipulated. Inserting a node results in eight changes: the three pointers of the inserting node and its identifier field, the next and after next pointer of the node before the inserted one, the previous pointer of the node after the inserter one, and the after-next pointer of the node before the previous one. This is drafted in figure 3.8.

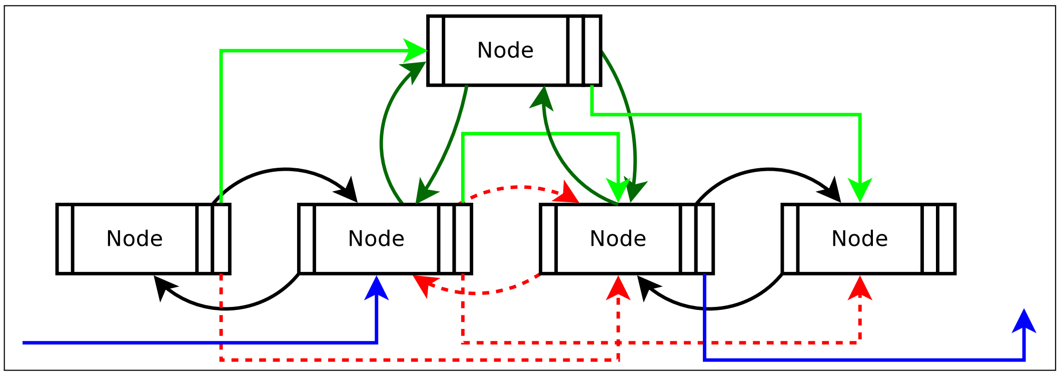

Figure 3.8: Modifications on DanceList when inserting a node in different tones of green for new pointer settings and red dotted pointers for removed references. The lighter green arrows shows the after-next pointers, the dark green ones the next and previous pointers. Missing in this draft: the identifier field of the inserting node.

3.2.5 Micro-benchmark

With the will to get more robust data structures higher costs come for every manipulation made to the list. Different measurements were made to check the speed of DanceList compared to a double- and single-linked list when no errors occur. Furthermore, the times of DanceList detecting and correcting single errors against the ones of a double-linked list is of interest in this work. Following this, the costs of correcting two changes will be determined.

Mostly no error occurs. That's why it is important to know how much overhead a fault tolerant data structure causes in case of no error. When no fault detection functions are enabled measuring with the time stamp counter delivers no result. So the number of instructions were analysed to get a representative outcome. While the default list needs nine instructions for a node's insertion, the double-linked list already takes fourteen steps. At last the DanceList needs more than twice as much instructions than the simple list. Enabling fault detection function means an additional function call and additional instructions checking the data structure. In terms of figures, this means that the DanceList takes about 50% more cycles to finish an insertion of a node than both the single-linked list and a not fault-tolerant double-linked list, like it is included in eCos. If this makes the DanceList just partially the speed of the double-linked list or less is clarified in the evaluation in chapter 4, as this is highly depending on the case of application.

It was clear from the outset that more robust data structures are slower than simple ones. So the next step is to examine the behaviour when errors occur. As double-linked lists are 1-correctable single errors were measured first. To do this error injection was also done static in the test cases by setting the specific pointer to another node. After this a certain number of nodes were inserted and the average number of cycles the insertions last was taken. To compare the DanceList against a double-linked list a simple detection and correction function for the double-linked list was implemented. This is shown in listing A.1.

First of all the next pointer was manipulated. To correct this error in double-linked list there is the need to go through the entire list. This makes the results depending on the number of nodes in the list. With an extra pointer to the after-next node the DanceList however is free of these effects when correcting the next pointer. So the DanceList has stable cycle values independent from the number of nodes, while the double-linked list's results increase with the quantity of nodes. But even on low count of nodes the DanceList is faster. Inserting ten nodes the double-linked list is about 70% slower on average, if this error occurs.

But considering the previous pointer of the DanceList it comes out that it is the same problem here. As there is no additional previous pointer to the second last node it is also necessary to go through the whole list. Therefore, the time needed for correction is also depending on the number of nodes. In figure 3.9 the overhead in relation to the next pointer correction is drafted.

There is high potential for optimizations to get better results. One idea is to add an extra pointer to the penultimate node. But this is not in scope of this work and may be of interest for future research on this topic.

Using the comparison function visualized in listing 3.5 the correction of the after-next pointer is the same problem as the previous pointer's one. This is because in the current implementation the after-next pointer correction depends on the strategy of previous pointer correction. The reason is that the detection of the next node (as a reminder: the after-next node's correction is done by checking the next node as well) results mistakenly in an error of its previous pointer. Here is also a starting point for optimization: generating better comparison functions may solve the problem.

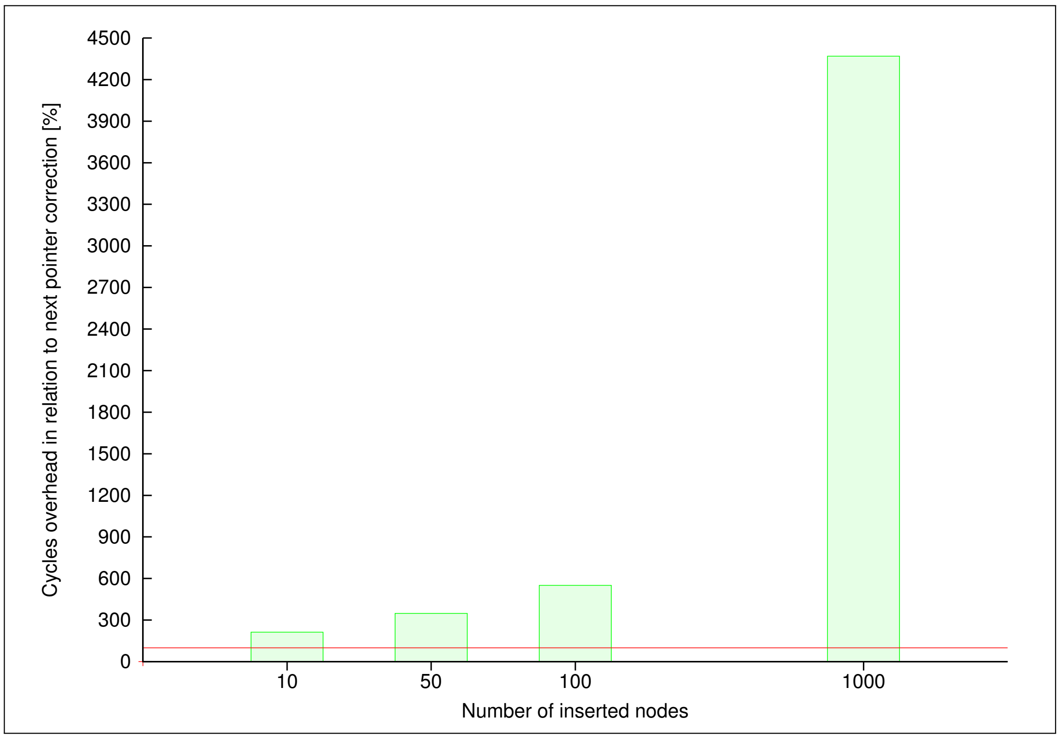

Figure 3.9: Overhead of correcting the previous pointer depends on the number of nodes in the list. This plot is showing difference between the correcting the next pointer and previous pointer in percent. The red line denotes 100%.

With the next, previous, and after-next pointer the complete organization structure of node in the DanceList is checked and, if necessary, corrected, if just one error occurs. But the DanceList is 2-correctable and, therefore, the question is what overhead is to be expected if two errors arise. Like having just one error, it depends on the type of error and on the comparison functions. A broken previous pointer results in the same problems as described above no matter what other pointer was changed. Things are different with a defect after-next pointer. If two errors are detected the after-next pointer can be corrected very fast due to the fact that it is known which correction mode (figure 3.6 or 3.7) must be used. If the previous pointer is broken additionally to the after-next pointer for example, this pointer can be corrected by taking on the reference of the next node's next pointer, as the next pointer is not damaged. After the correction of the after-next pointer, the previous pointer can be handled as the only error with the known issues. So the correction of two errors, which are discovered in one detection round, is a one by one correction of the two single errors. That is why the overhead depends, in case of one broken pointer being the after-next pointer, on the other one. If this for instance is the next pointer, there is little overhead, because first of all the next pointer is corrected by the previous node's after-next pointer and subsequently the damaged after-next pointer is corrected by next node's next pointer, as the after-next pointer is the only broken one after the next pointer was fixed. This example is drafted in figure 3.10. Finally, if two errors occur, the problem with the correction of the previous pointer remains, but the after-next pointer can be corrected fast, since the other broken pointer is known and consequently the way to correct this after-next pointer.

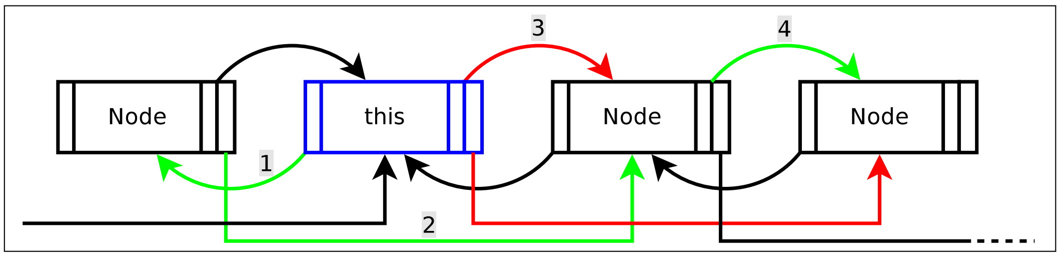

Figure 3.10: The correction starts with step 1 as it it is known that this previous pointer is not faulty. After step 2 the next pointer is fixed. Consequently it can be used for the correction of the after-next pointer as step 3 and finally complete the correction using step 4.

In the current status the inner structure of the DanceList is guarded. The thing to get access to the nodes, the head pointer, however, has never been mentioned yet. If there is no way to get to the nodes in the list any more, the entire data structure is useless. As shown in the data integrity part redundancy can protect such things as the head pointer. In the DanceList this is done automatically as there is an additional tail pointer to the end of the list. Using this pointer the head pointer can be re-constructed as its reference is the tail's next node.

In conclusion, the DanceList can be a great method to realize fault tolerance to data structures. Optimizations are necessary to get powerful data structures with the ability to be robust against more than one error. In the current version there are lacks in the detection and correction process. So the goal in future works must be to find better fitting comparison functions. If so, data structures are prepared for future tasks to be more fault tolerant.

CHAPTER 4

Evaluation

Two different methods to prevent operating systems faults caused by transient hardware faults were shown in the last chapter. Protecting data integrity was already evaluated in operating system context as described in subsection 3.1.4. The result was, that with higher demands more complex protection mechanisms are necessary, but this is associated with more overhead regarding the runtime. Adapted and fast implementations need hardware support embedded processors mostly do not supply. So the idea was to create own hardware to assist the processor, which might be a topic of future research. If the goal is to get a very fast implementation and this is not a false belief as soft errors are rare, a simple checksum with a few disadvantages can be taken instead of developing tricky mechanisms.

Data structure correctness was treated in a different way. The DanceList, a more fault-tolerant data structure, was developed and evaluated outside the operating system context. Resulting from the DanceList taking more than one and half times the cycles to insert a node when no error occurs the question was if the runtime is also one and a half times the amount of the not fault-tolerant double-linked list. To give an answer to this the DanceList was included into eCos so much that the scheduling queue was replaced by the fault-tolerant data structure, which got wrapper functions to fit into the eCos context. With a few threads suspending and resuming the entire test the scheduling procedure was called all along including the measuring of the time needed for adding the current thread to the scheduling queue, selecting a new thread from the queue, and switching to the next thread. The result is the average value of one thousand measurements done again with Intel's RDTSC. It became clear that the DanceList is not in the slightest half the speed of the double-linked list in this context. On the contrary the resulting overhead of runtime is just about 5% when no errors occur, which makes the fault-tolerant data structure profitable and to an acceptable choice to the standard scheduling queue implementation. One reason for this low overhead is that the access on the data structure during the scheduling process is very short. So the additional instructions for checking the correctness of the data structure are practically negligible. But this does not depreciate the DanceList in any way. Accesses on lists are a common procedure in operating systems during other processes like scheduling or organizing locks (threads may be suspended and, therefore, stored to a list, which arranges these threads). Lists respectively queues are supporting structures for the operating system and as a consequence the entire procedure must be considered and not only the overhead of fault tolerant mechanisms merely during the data structure's proceedings.

In case of an error the overhead depends on the kind of the error occurred, as mentioned in 3.2.5. If there are broken pointers, which can be fast repaired, the overhead increases barely perceptible. In the current configuration the next pointer is an example in return. Following the comparison functions the type of error is clarified and it is possible to jump to the correction function directly, which rectifies the data structure in a few steps. If it is possible to manage to achieve better results in the correction process for all errors, those can be corrected very fast and with extraneous overhead. Currently a broken previous pointer for example gives higher overhead depending on the number of threads in the list. So a goal in future studies can be to eliminate this disadvantage. But it must be kept in mind that the double-linked list has the same problems and furthermore the correction of the next pointer is also the same issue.

CHAPTER 5

Conclusion

In this thesis data in main memory were made fault tolerant. At first a very quick way to ensure data integrity was examined. The result is that very simple implementations are acceptable as the overhead is very slight and it is still 1-correctable. But if there is the need for more correctability, because soft error rate increases too much, better mechanism must be included, which results in more overhead. Therefore, hardware support for specific algorithms like the population count in a hamming code could reduce the runtime and make this error correcting code a reasonable possibility, and is a topic for future work.

Having more robust data structure was the second goal of this work. The DanceList is a 2-correctable data structure with several advantages. Adding an extra pointer was the basic idea which resulted in a better fault tolerance and a faster strategy for error detection was created. Because of this it is possible to say which pointer is broken and correct it in a quick way. Nevertheless, there is need for improvement, less the idea by itself, but the special comparison functions it is using, which is also giving space for enhancement. So with better and maybe less comparisons it is possible to speed up the detection and above all the correction of broken pointers. There is the idea to make an additional previous pointer to the node before the previous one to solve the problem with correcting the previous pointer. But then there is the question of the disadvantages like additional overhead. With all these thoughts it must kept in mind that the type of errors we try to detect and correct are very infrequent. Even so they must not be underestimated, as "the protection from radiation induced transient faults has become as important as other product characteristics such as performance or power consumption" [MZ08].

In this work simple and static test cases were used. Fault injection was done by manipulating values directly in the implementation to simulate errors. In future work fault injection must be done on real hardware and the tests have to be repeated. Therefore, real conditions can be emulated. Another point for future work is the handling of traps which has not been treated in this thesis. If a pointer of a data structure references not valid memory the trap, which is thrown, must be handled by correcting the broken pointer in the handling function. Additionally the two parts of this work (data integrity and correct data structure) must be combined in an efficient way with the result that not only the data structure but also its content is protected against transient hardware faults.

Finally, this field gives space for enhancement in future works which may result in more robust and efficient fault tolerance for operating system data structures.

Bibliography

[Che] Chen, Wickham(1); Liu, Tiankuan(2); Gui, Ping(1); Xiang, Annie C.(2); Cheng-AnYang(2); Zhang, Junheng(1); Zhu, Peiqing(1); Ye, Jingbo(2) und Stroynowski, Ryszard(2): Single Event Effects in a 0.25 m Silicon-On-Sapphire CMOS Technology, Techn. Ber., 1: Department of Electrical Engineering, Southern Methodist University, Dallas, TX, 75275; 2: Department of Physics, Southern Methodist University, Dallas, TX, 75275 (Cited on page 11)

[Coo97] Coorporation, I.: Using the RDTSC Instruction for Performance Monitoring, Techn. Ber., tech. rep., Intel Coorporation (1997) (Cited on page 22)

[Die] Dietz, Henry Gordon: The Aggregate Magic Algorithms, Techn. Ber., University of Kentucky, URL `http://aggregate.org/MAGIC/` (Cited on page 21)

[Her06] Herold, H.; Lurz, B. und Wohlrab, J.: *Grundlagen der Informatik*, Pearson Studium (2006) (Cited on page 18)

[Jog05] Joglekar, A.; Kounavis, M.E. und Berry, F.L.: A scalable and high performance software iSCSI implementation, in: *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies-Volume 4*, USENIX Association, S. 20–20 (Cited on page 19)

[Koh00] Kohler, E.; Morris, R.; Chen, B.; Jannotti, J. und Kaashoek, M.F.: The Click modular router. *ACM Transactions on Computer Systems (TOCS)* (2000), Bd. 18(3):S. 263–297 (Cited on page 13)

[Kou] Kounavis, M.E. und Berry, F.L.: A systematic approach to building high performance software-based CRC generators, in: *Computers and Communications, 2005. ISCC 2005. Proceedings. 10th IEEE Symposium on*, IEEE, S. 855–862 (Cited on pages 18 and 19)

[Mas03] Massa, A.J.: *Embedded software development with eCos*, Pearson PTR (2003) (Cited on pages 13, 16, and 43)

[MZ08] Miskov-Zivanov, N. und Marculescu, D.: Modeling and optimization for soft-error reliability of sequential circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* (2008), Bd. 27(5):S. 803–816 (Cited on page 39)

[Sar88] Sarwate, D. V.: Computation of cyclic redundancy checks via table look-up. *Commun. ACM* (1988), Bd. 31:S. 1008–1013, URL `http://doi.acm.org/10.1145/63030.63037` (Cited on pages 18 and 19)

[Shi02] Shivakumar, P.; Kistler, M.; Keckler, S.W.; Burger, D. und Alvisi, L.: Modeling the effect of technology trends on the soft error rate of combinational logic, in: *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, IEEE, S. 389–398 (Cited on pages 11, 12, 28, and 43)

[Sin] Singhal, R.: Inside Intel® Next Generation Nehalem Microarchitecture, in: *Hot Chips*, Bd. 20 (Cited on page 19)

[Tay80a] Taylor, D.J.; Morgan, D.E. und Black, J.P.: Redundancy in data structures: Improving software fault tolerance. *Software Engineering, IEEE Transactions on* (1980), (6):S. 585–594 (Cited on pages 27 and 28)

[Tay80b] Taylor, D.J.; Morgan, D.E. und Black, J.P.: Redundancy in data structures: Some theoretical results. *Software Engineering, IEEE Transactions on* (1980), (6):S. 595–602 (Cited on page 27)

List of Figures

List of Tables

APPENDIX A

Appendix

Detecting and correcting changes in a double-linked list

Listing A.1: Not optimized error detection and correction in a double-linked-list

```
void DList::ft(Node *node) {
  if(node->next->prev == node) return;

  // Next ptr broken
  if(node->next->id != listid) {

    // pointing to invalid memory
    Node *tmp = node;
    Node *schlepp;

    // prev until we point on node again. schlepp is next now
    while(node != (tmp = tmp->prev)) schlepp = tmp;

    node->next = schlepp;
    return;
  }

  // next->prev ptr broken
  if(node->next->prev->id != listid) {
    node->next->prev = node;
    return;
  }

  // next OR next->prev ptr broken; one is pointing to a wrong node in list
  // => count number of nodes while going forward and backward through the
     list
  // the one with less nodes lost some and is the broken ptr
  int i = 0, j = 0;
  Node *tmp = node;

  while(node != (tmp = tmp->next)) i++;

  tmp = node->next;

  while(node->next != (tmp = tmp->prev)) j++;
```

```
    if(i < j) { // next ptr broken
      Node *tmp = node;
      Node *schlepp;

      // prev until we point on node again. schlepp is next now
      while(node != (tmp = tmp->prev)) schlepp = tmp;

      node->next = schlepp;
      return;
    } else if (i > j) { // next->prev ptr broken
      node->next->prev = node;
      return;
    } else { // something went wrong, but should not happen!
      printf("ERROR while correcting DList\n");
    }
  }
```