2012 IEEE 18th International Conference on Parallel and Distributed Systems

Self-Stabilizing Micro Controller for Large-Scale Sensor Networks in Spite of Program Counter Corruptions due to Soft Errors

Christopher Boelmann*, Torben Weis*, Michael Engel† and Arno Wacker‡

*Distributed Systems Group, University of Duisburg-Essen, Duisburg, Germany
Email: christopher.boelmann@uni-due.de, torben.weis@uni-due.de
†Design Automation for Embedded Systems Group, TU Dortmund University, Dortmund, Germany
Email: michael.engel@tu-dortmund.de
‡Software-engineering for Ubiquitous-Computing Applications, University of Kassel, Kassel, Germany
Email: arno.wacker@uni-kassel.de

***Abstract*—For large installations of networked embedded systems it is important that each entity is self-stabilizing, because usually there is nobody to restart nodes that have hung up. Self-stabilization means to recover from temporary failures (soft errors) and adapt to a change of network topology caused by permanent failures. On the software side self-stabilizing algorithms must assume that the hardware is executing the software correctly. In this paper we discuss cases in which soft errors invalidate this assumption, especially in cases where CPU registers or the watchdog timer are affected by the fault. Based on the observation that a guaranteed self-stabilization is only possible as long as the watchdog-timer is working properly after temporary failures, we propose and compare three different approaches that meet the requirements of sensor networks, to solve this problem with a combination of hardware- and software-modifications:**

1) **A run-time verification of every watchdog access**
2) **A completely hardware-based approach, without any software modifications**
3) **A 2^X byte code alignment, to realign a corrupted program counter**

Furthermore we determine the average code-size increase and evaluate necessary hardware-modifications that come along with each approach.

***Keywords*-Self-Stabilization; Soft Errors; Embedded Systems; Static Code Analysis.**

I. Introduction

Large-scale networked embedded systems consist of hundreds of sensor nodes distributed over a large area or building that shall never stop operating. The larger the network, the more likely are temporary soft errors or even permanent node failures. Therefore, the system should be able to recover from these failures without manual intervention.

Since soft-errors primarily occur within flip-flops, mainly hardening flip-flops against soft errors is considered by microprocessor designers. However, advances in technology result in decreased device feature sizes and higher operating frequencies of microprocessors leading to an increasing soft-error rate (SER) in combinational-logic. Thus, the SER of combinational-logic may soon dominate over the SER of flip-flops and become a major vulnerability in terms of soft errors as Mahatme et al. have shown [10].

However in large-scale networked embedded systems hardware redundancy and expensive casing of nodes to harden micro controllers against soft errors are prohibitive to reduce hardware costs and energy consumption of sensor nodes. Thus, it is more likely that environmental influences (heat, electric fields, mechanical stress) cause random bit errors and hence soft errors which can affect RAM and CPU registers alike. As a consequence CPU instructions might be carried out wrongly due to a temporary external influence. This means the system must be able to recover from any temporary failure which may have left the system in an arbitrary state.

Fault tolerant systems are designed to show no failure despite faults for a limited set of faults. In contrast, self-stabilization [2], [8], [9], [11] does not prevent the failure, but it ensures that the system converges to a correct state in a fixed amount of time after the externally induced fault is gone, i.e. RAM, registers, ALU etc. operate normally again. An important advantage of self-stabilization is that it can recover from a wide set of faults. In our case we can assume that all memory bits in the system can be altered and it will still recover in fixed time. The drawback of self-stabilization is that there are no guarantees during stabilization time. This is the trade-off between fault tolerance and self-stabilization.

Many self-stabilizing algorithms have been devised [2], [11] since Dijkstra's paper on self-stabilization [1] which can recover from erroneous data structures or modified messages caused by RAM or communication failures. However, these algorithms assume that the CPU always executes the self-stabilizing algorithm and no arbitrary code [3]. Unfortunately, this assumption is not necessarily true even if the program itself is burned in the ROM. As we show, a bit error in the PC register can cause the system to execute bogus code resulting in an endless loop. An often used standard measure against such endless loops is a watchdog timer (WDT) resetting the CPU if the WDT is not reset regularly.

1521-9097/12 $26.00 © 2012 IEEE
DOI 10.1109/ICPADS.2012.75

IEEE computer society

However, in the worst case the bogus code forces the CPU in an endless loop where it continuously resets the WDT. This would prevent the system from stabilizing without a manual reset. Thus, to achieve a fully self-stabilizing system we must consider hardware and software together.

In this paper we propose methods to ensure that succeeding a soft error a CPU will execute the correct instructions again after a fixed amount of time while meeting the special requirements of sensor networks. This ensures that the self-stabilizing software running on a sensor node can recover from the failure as well in fixed time and thus increases the overall reliability in an ongoing system without manual administration while keeping energy consumption and additional hardware-costs low. Thus, a memory failure can arbitrarily modify the RAM and CPU registers and the system will nevertheless recover in fixed time.

The paper is organized as follows: First we present related work regarding self-stabilization and a self-stabilizing microprocessor in Section II and point out which unsolved problems are sorted out by our approaches. In Section III and IV we discuss our system model and analyse which CPU register faults could keep the CPU from executing the intended machine code permanently. Consequently, we propose three solutions to prevent this permanent failure. In Section V we estimate and compare the average code-size increase and the implications of hardware- and software-modifications for each approach. Finally, we discuss how our approaches can be applied to other hardware architectures as well.

II. Related Work

The concept of a self-stabilizing microprocessor was first introduced by Dolev et al. [3] who proposed to use a watchdog circuit to recover a microprocessor from an infinite loop within its micro-code[1] due to soft errors. This watchdog circuit resets the microprocessor if it did not perform a fetch-decode-execute sequence for a determined period of time. Furthermore, corrupted memory values are detected by determining *safe states* within the program that are reached infinitely often and validating predefined invariants within these states. Moreover, self-stabilization preventing problems caused by arbitrary register values or corrupted PCs are considered by validating the current program state whenever a program statement has been executed [5]. However, in contrast to our approaches these methods does not consider all possible PC corruptions since they assume that only real program code will be executed. Thus, they do not prevent a CPU from executing infinite loops on the data fields of machine code instructions instead of executing the opcodes of the instruction that might bypass the watchdog circuit since misinterpreted code is fetched, decoded and executed and therefore may not guarantee self-stabilization in every case of PC corruption.

[1]Machine-code instructions are sequences of micro-code instructions

In early 2012 Dolev et al. presented a method to enable self-stabilization for commercial off-the-shelf microprocessors [4]. They propose to force a periodic reboot of the commercial microprocessor initiated by an additional microprocessor running as a watchdog. The program contains code to store and restore the current system state that is called before the processor is reset by the watchdog. Thus, the values can be validated on each reset and every register that has not been stored will be reinitialized with default values. Since the address of the next instruction (PC) to execute after a reboot is stored and restored, too, a corrupted PC will also be restored. To handle this problem Dolev et al. propose to include sanity checks for (calculated) jump destinations preceding every jump instruction to secure the control flow of the program. However it is not considered that a corrupted PC might point at data-fields of instructions and thus can get in an unintended infinite loop, unable to perform the proposed sanity checks for (back-)jumps. In this worst case scenario even the watchdog reset is unable to repair the corrupted PC which is fatal for an ongoing system.

Dolev et al. state that *"the criteria for a processor to be self-stabilizing is a repeated execution of a fetch-decode-execute sequence"* [3]. In this paper we address self-stabilization preventing problems that can be caused by a corrupted PC pointing at data fields, executing arbitrary misinterpreted instructions. Thus we expand the criteria for a self-stabilizing processor to require the repeated execution of a fetch-decode-execute sequence to be performed only *on intended instructions*.

Since the approach by Dolev et al. requires an additional watchdog processor to ensure self-stabilization for a microprocessor the extra costs to protect a large-scale sensor network are not negligible. Furthermore the energy consumption of sensor nodes increases with a second microprocessor attached while the periodic reset slows down its execution speed. Therefore their approach is not optimal suited for highly resource constrained and cheap devices used in large-scale sensor networks that we consider. In this paper we propose three approaches to ensure the self-stabilization property of a microprocessor to solve the problems resulting from a PC corruption while meeting the requirements for sensor networks where self-stabilization is crucial.

Yee at al. proposed a 32 byte code alignment to control and realign the program flow of untrusted x86 native code [12]. Their intention is to provide a sandbox with high computational performance for browser-based applications without compromising security. Besides other measures they restrict jumps to only point at addresses with valid instructions to maintain the control flow integrity. The validation of jump destinations is realized in software by using a new jump instruction that computes the target and checks if it is 0 mod 32.

Even though our alignment approach (see Section IV-C) uses a similar alignment approach to restrict the possible program flow it does not have the purpose to consider security issues in terms of attackers but to guarantee the eventual realignment of a corrupted PC. Furthermore our approach realizes the validation of jump destinations in hardware thus preventing further problems in cases where a corrupted PC might bypass a validation that is realized in software.

III. System Model

Our investigation focuses on the MSP 430 micro controller that is often used in embedded systems [6] and requires a small amount of hardware modifications, depending on the chosen approach. Without these hardware modifications we cannot guarantee the self-stabilization property, as shown in Section IV.

The embedded systems we consider are equipped with sensors, radio and sometimes actuators. A basic application consists of a main loop which waits for the next event, processes it and subsequently starts over again:

```
void main(){
  while(true){
    ev = wait_for_event();
    process_event(ev);
    send_output();
    reset_watchdog();
}}
```

The Texas Instruments MSP 430 micro controller has a 16 bit reduced instruction set with 27 instructions and 24 emulated instructions [7]. The program code is stored in ROM. The MSP 430 provides a WDT, that is realized by an integer register which is incremented automatically by a hardware timer. If the WDT is not reset in time and the integer value exceeds a threshold value the CPU will be reset.

In line with the general assumptions for self-stabilization, we can assume, that there will be no persistent errors. Therefore the timer will eventually return to incrementing the register normally again, even if an error occurred. Alterations of the integer value within the register will stabilize automatically, since the value will either be below the threshold and eventually exceed it or it will be above the threshold and lead to an overflow exception resetting the CPU when exceeding the maximum register size.

IV. Watchdog-based Self-stabilization Approaches

The standard measure to recover from event processing problems is to start a WDT. If the event processing does not complete fast enough, the WDT will interrupt and reset the system. This solves the problem that a CPU can get stuck in an erroneous endless loop[2]. Since the WDT reset is called in the tail of the main loop, it can only be triggered under normal running conditions and not out of an unwanted endless loop. Therefore, as long as only program code will be executed we can guarantee, that a CPU will eventually be reset after it got stuck, and thus operate normally again.

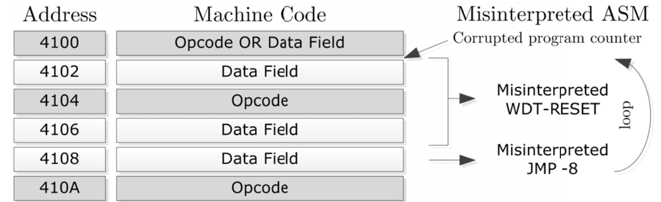

Figure 1: Loop containing a misinterpreted WDT reset within data fields

However, if the PC register gets corrupted and points at a data field of a CPU instruction, another problem can occur since the CPU cannot distinguish opcodes from data fields[3]. In the best case the data field represents an illegal opcode that cannot be executed by the CPU, which triggers an interrupt and reset. If it can be executed, the PC will move forward and might eventually hit an illegal opcode or a real opcode field and execute normal program code again.

In the worst case the CPU might get stuck in an endless loop on data fields that are interpreted as opcodes. The WDT will take care of this problem as well, as long as there is no misinterpreted machine code in the data fields that resets the WDT. Even though this scenario is rare its occurrence has a fatal influence on the reliability of an ongoing embedded system, since it would cause the CPU to be trapped in an endless loop with no chance for self-stabilization. An example[4] for this undesirable case is shown in Figure 1.

Two conditions must hold to create an inescapable loop: The CPU must *execute a loop* and the *loop must reset the WDT* before it resets the machine. To prevent the PC from getting stuck in such an inescapable loop it is sufficient to meet one of the following two conditions:

- Prevent unintended loops and ensure that the PC will eventually point to correct CPU instructions again
- Prevent that a corrupted PC-register can lead to a watchdog reset

Each proposed approach focuses on meeting either one of these conditions. Since in every case a running WDT is necessary to escape endless loops on real instructions

[2]The only desired endless loop is the main loop

[3]Flagging opcodes and data fields by setting the first bit to 0 or 1 is prohibited, because this would enforce a completely new instruction set since there is one bit less to encode operations and also 2 bits less to encode data.

[4]Indirect WDT resets are also possible when e.g. a valid reset setting is stored in a register and afterwards written to the WDT register at the address 120h

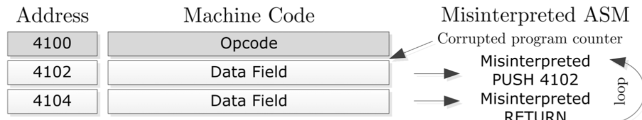

Figure 2: Loop caused by an absolute jump (`RETURN` emulated by `MOV @SP+,PC`)

we need a customized micro controller with the following hardware modifications for all approaches.

First we require that the memory is divided in two segments, RAM and ROM. The ROM contains only the executable program code and RAM contains static and dynamic data segments[5]. Execution of program code in RAM is not possible. This can be realized by forcing the upper PC bits to a static value. If the PC was able to point to RAM, we could not state anything about the arbitrary RAM content, rendering static program analysis useless.

Our second required hardware modification is that (1) the WDT cannot be turned off, (2) that the WDT does not allow non-maskable interrupts to be triggered instead of resets and (3) that its usage as timer is prevented (i.e. it is always in reset-mode). This is required to deal with bit alterations within WDT registers, because otherwise we could not be sure that the WDT will never be turned off by an altered instruction. Thus, we can only *reset the WDT* and *change the reset interval*.

We assume that the main loop is the only endless loop and that the CPU will only execute instructions in ROM. Thus, the only possible sources of endless loops outside the main loop are data fields interpreted as instructions. An intuitive approach is to analyse the program code for possible WDT resetting loops and break them. However, even if we could propose an algorithm to break every endless loop occurring because of relative jumps within data fields there would still be the possibility of loops based on `RETURN` instructions. Since the destination of `RETURN` instructions is determined by the stack content that can contain unknown arbitrary values at runtime, we are not able to find and prevent inescapable endless loops as shown in Figure 2, rendering this approach non-viable. Therefore, other approaches have to be considered. Three possible approaches are described in the next sections.

A. Approach 1: Register/ROM Compare

The first approach prevents any unintended WDT access caused by a corrupted PC pointing on data fields and thus meets the second condition from Section IV.

In an MSP 430 micro controller the WDT register is located at address `120h`. With static code analysis (SCA) we are able to find any potential direct WDT access (e.g. `MOV @PC+,120h`), since it is always structured in the same 48 bit pattern and thus can be located within the machine code. However, because of indirect addressing, SCA cannot determine all writes to `120h` statically.

Since we cannot find and prevent every unintended WDT access we restrict the access to the WDT register. We observed, that a typical program can be restricted to accessing the WDT in the main loop only. Thus, there is exactly one instruction at a certain address in the program code that is allowed to reset the WDT. To achieve this, for every WDT access the PC is compared with the stored valid address. If the address is valid, the WDT access is granted, otherwise the CPU is reset.

The only remaining problem is an unfortunate setting of the WDT interval. If the interval is too short to execute the complete main loop, because the setting was read wrong (e.g. in case of a bit-flip), the WDT will trigger and the main loop is entered again, causing an endless loop. This can be fixed by moving the WDT reset (which is also setting the interval) to the beginning of the main-loop. A corrupted WDT interval setting would thus be repaired after the CPU is reset and thus heal itself.

The valid address is stored at a predefined position in ROM and thus cannot be corrupted. This can be realized in the code-emit step of compiling by writing the WDT access address e.g. in the last bytes of the ROM. Validation of WDT access is done with an additional comparison register that stores the valid address. To ensure that the comparison register content is correct it is read from ROM preceding every comparison. Afterwards the access address and the comparison register content are compared with an `AND` operation in hardware. Since this comparison is only done when the WDT is accessed the execution slowdown is irrelevant, because either the WDT access was intended and the main-loop is currently restarting or the WDT access was unintended and the CPU is reset.

An advantage is that the WDT will take care of all loops and PC corruptions since only intended WDT accesses are possible. Either the WDT is reset in time or the CPU will be reset after the WDT expires, regardless of PC corruptions and register faults. Furthermore only minor compiler-modifications are necessary to store the valid WDT address in ROM. There is also no code-size increase since there is no need for code modifications.

However, since there can only be one valid WDT access in a program it may be unsuitable in some cases. A workaround in form of a WDT reset function to make the WDT accessible from more than one address is prohibited, because the function may be called from an unintended loop and thus create an inescapable WDT resetting loop with no chance for self-stabilization.

[5] The compiler can convert static data stored in program code (e.g. static tables of constants) into a set of instructions writing the static data into RAM for further access

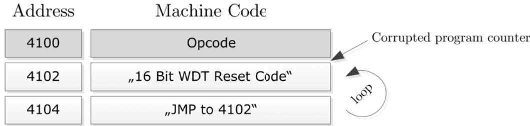

Figure 3: An inescapable WDT resetting endless loop within data fields of one instruction

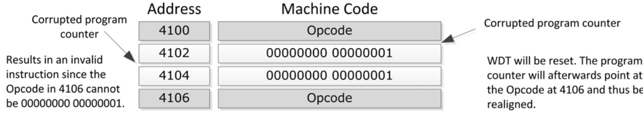

Figure 4: Possible outcomes of a corrupted PC hitting an unintended `WDTRST` in data fields of one instruction

B. Approach 2: Remove WDT register from address space

As stated in Section IV-A preventing a corrupted PC from accessing the WDT register with SCA is almost impossible because of indirect addressing. Our second approach moves the physical location of the WDT register out of the accessible address space to restrict the WDT access and can be realized completely in hardware. Thus, the original WDT access instructions are no longer able to access the WDT. Furthermore we extend the instruction set with a new WDT reset instruction `WDTRST` to access the moved WDT register. `WDTRST` uses a predefined WDT setting specified in hardware as well. Therefore the WDT cannot be set to unintended settings or intervals anymore. Adding `WDTRST` can easily be done since we do not need any addressing information and thus may use any instruction code that can never be used by any other instruction.

In case of the MSP 430 we chose `00000000 00000001` for `WDTRST`, since it enables us to easily prove the self-stabilization property[6]. Furthermore `WDTRST` must have a data field, in our case also containing `00000000 00000001`, to stretch the instruction length to 32 bit. It has to be at least 32 bit long since with a 16 bit instruction we are not able to ensure that a corrupted PC will either reset the CPU or be realigned (see Fig. 3).

Even though a `WDTRST` could be completely located within two sequential data fields we can state that the PC will either be realigned to real instructions if it accidentally executed a `WDTRST` within data fields or the CPU will be reset because of an invalid instruction (see Fig. 4). Thus a corrupted PC could reset the WDT only once and will be realigned to valid instructions afterwards preserving the self-stabilization property.

To restrict the WDT register access it has to be moved out of the accessible address space. Furthermore, the `WDTRST` instruction with the default WDT settings has to be realized

[6]We did not choose `00000000 00000000` because in this case the empty memory space at the end of the program would be interpreted as `WDTRST` as well

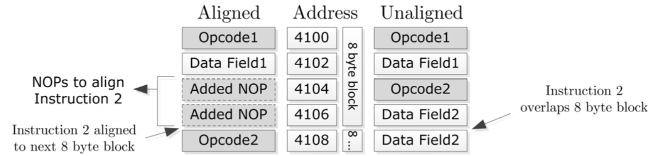

Figure 5: Example for 0 mod 8 aligned (left) and unaligned (right) program code

in hardware to access the relocated WDT register. The settings should contain the maximum possible WDT interval to ensure that the CPU is eventually reset without influencing normal program execution.

In summary, our second approach ensures that no unintended loops (inside instruction data fields) can continuously reset the WDT, because of the `WDTRST` opcode structure. It follows that such unintended loops will eventually be left because of a WDT reset. An advantage over the initial approach is that WDT resets are possible in multiple locations in the program. However, the programmer must meet our requirement that his code (when executed normally) contains no endless loops with WDT resets. Our approach only guarantees that bit-errors in memory and registers do not cause infinite loops. Hence, our first approach is more strict and less dependent on the specifics of the MSP 430 instruction set.

C. Approach 3: 2^X Byte Alignment

In contrast to the previously described approaches this approach prevents unintended loops and thus fulfils the first condition mentioned in Section IV by restricting the possible program flow and ensuring that the PC will eventually point at valid instructions again.

This is done by aligning the machine code so that no instruction overlaps a 2^X byte boundary. This 2^X byte alignment ensures that at an address dividable by 2^X (0 mod 2^X) there is either a valid instruction or the value null (see Fig. 5). A similar 32 byte alignment was used in Google Chrome's Native Client [12]. The modified micro controller must ensure that all possible jumps are only performed if their destination is a 0 mod 2^X address, i.e. ensure that the last X bits of the destination address are zero. In other cases the CPU will be reset. Thus any jump will only be performed if it is aiming for a 0 mod 2^X address and thus a valid instruction or the value null (empty code space).

For the specified system we can make the following statements to point out that the CPU will eventually execute valid instructions again and thus prove the self-stabilization property.

1) If everything works correctly the main-loop will periodically reset the WDT.
2) If the PC gets stuck in an endless loop on valid instructions (e.g. a faulty register value corrupts a loop

break condition) the WDT will not be reset in time and in turn resets the CPU.

3) If a corrupted PC points at a data field and there is no jump instruction within the following misinterpreted machine code, the PC will eventually reach the end of the program code and reset the CPU if it does not encounter an invalid instruction first.
4) If the corrupted PC hits a misinterpreted jump instruction there are two possible outcomes:
   a) The jump points at a valid 0 mod 2^X address and the PC returns to executing valid instructions again leading to 1) or 2).
   b) The jump does not point at a 0 mod 2^X address and the CPU is reset.

To demonstrate the realization of this technique we use an 8 byte alignment[7]. The customized micro controller has to verify the PC register value upon each programmatic modification. Instructions that might modify the PC register in case of the MSP 430 are jump instructions, `CALL`, `RETURN` and instructions that immediately access the PC register (e.g. `MOV ...,PC`). Thus every writing PC register access has to be checked in hardware to verify that the written address is 0 mod 8. The compiler ensures that no instruction overlaps an 8 byte boundary by putting 16 bit no-operation instructions (`NOPs`) in front of an overlapping instruction (see Fig. 5). Furthermore we increase the range of jump instructions by using their relative 10 bit address as the amount of blocks we want to jump. Thus it is possible to jump 512 blocks forward or 511 blocks backward, increasing the range by a factor of 4. This is necessary because the alignment with injected `NOP` instructions might cause relative jumps to exceed their range. The range extension can be realized by appending `000` to each jump destination address. However, since the address of the jump instruction itself is not necessarily 0 mod 8 aligned, the relative address might as well be unaligned (i.e. pointing into a block instead of its beginning) and cause a CPU reset. Therefore the actual PC is 0 mod 8 aligned by setting the last 3 bits to `000` to ensure that the PC points at an address dividable by 8 before calculating the jump destination address. Additionally `CALLs` have to be aligned to the last bytes of a block so that the `RETURN` address will be a 0 mod 8 address.

This approach is independent from micro controller opcodes and does only need the described hardware modifications. It is not needed to consider unintended WDT resets since we prevent unintended loops and thus unintended WDT resets can only happen a finite amount of times. The downside of this approach is the possible code-size increase caused by the alignment (see Section V).

[7]In Section V we provide a formula to ease selecting an appropriate alignment depending on the actual program

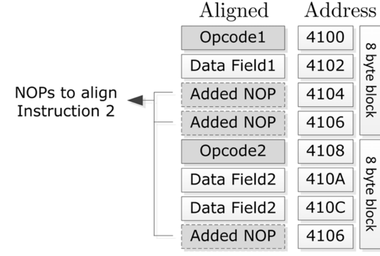

Figure 6: Max. overhead for 8 byte alignment (MSP 430): 6 `NOP` bytes for 10 aligned instruction bytes $\Rightarrow W = 60\%$

V. Analysis

The probability for a randomly chosen program to contain a WDT reset loop on data fields of machine instructions is very low. However, if a program has such a WDT reset loop, then there is a significant probability that the system enters a state from which it cannot self-stabilize. This is because our applications consist of a few thousand instructions only. The probability that a random PC error hits an unintended WDT reset loop is therefore in the order of $\frac{\#vulnerable\text{-}addresses}{program\text{-}size}$. Thus, the overall probability for this error occurrence is low, while the dependent probability in programs that are vulnerable for this error is not negligible.

In this section we compare and evaluate the different approaches in terms of average code-size increase and implications on hardware- and software-modifications.

The first approach does not increase the code-size. The only change in code-size results from code adaptations prior to compiling, e.g. storing the WDT access address and code refactoring to reduce WDT reset occurrences to the main loop.

The second approach does not increase code-size either, because the normal 48 bit WDT reset occurrences can be directly replaced in the machine code with the new 32 bit `WDTRST` followed by a 16 bit `NOP` without shifting or influencing other instructions.

The third approach increases the code-size, because it has to insert `NOPs` to realize the alignment. To estimate the amount of additional `NOPs` instructions we have to consider the following three cases:

1) Instruction alignment at block boundaries
2) Jump-target alignment
3) `CALL` instruction alignment.

The average amount of added `NOP` instructions $NOP_{\#}$ is composed of the average amount of `NOPs` added for instruction alignment $NOP_{\#ins}$, jump-target alignment $NOP_{\#jmp}$ and `CALL` instruction alignment $NOP_{\#call}$.

$$NOP_{\#} = NOP_{\#ins} + NOP_{\#jmp} + NOP_{\#call} \qquad (1)$$

To align an instruction that crosses a block boundary, `NOP` instructions with 2 bytes each have to be added preceding the

aligned instruction (see Fig. 5). The worst case for $NOP_{\#ins}$ in bytes consists of the program size in bytes PS divided by the chosen alignment A, times the maximum waste of bytes of an alignment block in percent by added `NOPs` W. The overhead W depends on A and the maximum and minimum instruction sizes INS_{max} and INS_{min}.

$$W = \frac{INS_{max} - INS_{min}}{A - (INS_{max} - INS_{min})} \quad for \quad A \geq 2 * INS_{max} \quad (2)$$

In case of the MSP 430, W is 60% for 8 byte alignment (see Fig. 6), 33% for 16 byte and 14% for 32 byte. To get the average amount of additional `NOP` instructions the calculated worst case in bytes has to be divided by four[8]. Thus $NOP_{\#ins}$ can be estimated as:

$$NOP_{\#ins} = \frac{PS}{A} * W * \frac{1}{4} \quad (3)$$

Jump targets must be shifted to the beginning of an alignment block. One jump target can be aligned with at most $A - INS_{min}$ additional bytes where INS_{min} is the minimal instruction size in bytes. The average amount of additional `NOP` instructions $NOP_{\#jmp}$ is determined by the maximum amount of bytes needed to align one target and the number of jump targets in the program J, again divided by 4 to get the average amount.

$$NOP_{\#jmp} = (A - INS_{min}) * J * \frac{1}{4} \quad (4)$$

`CALL` instructions have to be the last instruction in a block to align their `RETURN` address. The amount of additional `NOPs` depends on how many instructions are already within the block. At most we have to add $A - sizeof(\texttt{CALL})$ bytes to align the call if the `CALL` is the first instruction in a block. Thus, the average depending on the number of `CALLs` C in the program can be estimated with:

$$NOP_{\#call} = (A - sizeof(\texttt{CALL})) * C * \frac{1}{4} \quad (5)$$

The complete amount of `NOPs` to add $NOP_{\#}$ can thus be estimated with the sum of the formulas:

$$NOP_{\#} = \frac{(\frac{PS*W}{A} + (A - INS_{min}) * J + (A - size(CALL)) * C)}{4}$$

Since the chosen alignment A is a factor as well as a divisor in the formula, the `NOPs` count depends on the other constants: The program size PS, the overhead constant W, the amount of jump targets J and `CALLs` C. For short programs with many jumps a small alignment is better while for long programs with fewer jumps the alignment should be higher. Thus the best suited alignment can only be determined for known program code.

[8]Divide by 2 since one `NOP` is 2 bytes and we look for the amount of `NOPs` added. Furthermore we divide by 2 since we assume that in average we neither have the worst case amount, nor the best case amount (0) of additional bytes, and thus take the arithmetic mean.

VI. Generalization

So far, we described our approaches only for the MSP 430 micro controller. In this section we identify the general requirements for applying them to other hardware architectures as well.

The first approach is independent from the hardware used. It only requires an additional check of the WDT access origin. This can be realized in micro controllers with an additional register and the `AND` instruction. This validation has to be performed in hardware whenever the WDT register is accessed. Furthermore the valid WDT access address must be stored in ROM by the compiler.

The hardware modifications for the second approach are limited to moving the WDT register out of the addressable space and make it accessible with a new `WDTRST` instruction. The structure of `WDTRST` depends on the instruction set of the micro controller. The `WDTRST` instruction must be unique within the machine code and thus has to be chosen in a way that it can never collide with other valid instructions. Since all instruction sets use prefixes followed by the opcode to determine the structure of an instruction it is always possible to create a `WDTRST` with a code that is lower than the lowest opcode prefix (e.g. in case of the MSP 430 the lowest opcode prefix is `000100` [7]). Furthermore the chosen `WDTRST` instruction opcode has to be at least as long as the maximum size of data fields to guarantee the realignment of a corrupted PC as stated in Section IV-B.

To transfer the third approach to other micro controllers it has to be checked if there exist more instructions that are able to modify the PC register because additional PC modifications must be verified for the chosen alignment in hardware, too. The flow control can be realized in other micro controllers as well by checking in hardware if the last X bits of the value written to the PC register are zero in case of an 0 mod 2^X alignment. Furthermore the code alignment is realized by the compiler and does not depend on any special property of the MSP 430. Any program code can be aligned to 0 mod 2^X addresses as long as all possible instructions for the micro controller are shorter than 2^X bytes and there is enough free memory for the aligned machine code. It is also important that there exists an instruction that does not need a data field and also does not influence the program execution for injecting and aligning the code (in our case the `NOP`). However, if such an instruction does not already exist within the instruction set it shall be possible to create it (e.g. with `MOV R3,R3`).

VII. Conclusion

In this paper, we presented three approaches that ensure that a micro controller will eventually self-stabilize and return to executing valid instructions without manual intervention despite soft errors. Our approaches solve the problem that a corrupted PC might point at data fields of machine instructions and might furthermore reset the WDT

infinitely often causing the CPU to be stuck in an endless loop. The approaches differ in their trade-offs in terms of code-size and hardware-modifications.

The first approach requires a hardware check of each WDT register access and thus needs a non-trivial hardware change to the WDT access logic. However there is no change to the opcodes and only little compiler changes are required. The second approach introduces a new opcode with special constraints on its byte layout and prohibits direct addressing of the WDT register. Depending on the target hardware architecture this can be more or less difficult to achieve. In addition, the compiler back-end must be adapted to use the modified instruction set, i.e. the `WDTRST` instruction. The third approach requires only a small hardware change (checking the lower bits of the PC), but the compiler needs substantial work to implement the required code alignment. However the native client compiler has already proven that this is possible [12].

We have shown, that all proposed approaches are generally applicable for other micro controllers as well, as long as the requirements mentioned in Section VI are met by the controller and its instruction set.

Furthermore all of our approaches are suitable for large scale sensor networks since they can guarantee that permanent errors due to soft errors will not occur while they keep the energy-consumption and additional hardware-costs low. Thus, micro controllers implementing our approaches are a desirable foundation for self-stabilizing algorithms in sensor networks.

REFERENCES

[1] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17:643–644, November 1974.

[2] S. Dolev. Self-Stabilization. *MIT Press*, Cambridge, MA, USA, 2000.

[3] S. Dolev and Y. Haviv. Self-Stabilizing Microprocessor: Analyzing and Overcoming Soft Errors. *IEEE Transactions on Computers*, 55, pages 385–399, IEEE Computer Society, Los Alamitos, CA, USA, February 2006.

[4] S. Dolev and Y. Haviv. Stabilization Enabling Technology. *IEEE Trans. Dependable Secur. Comput.*, 9, 2, pages 275–288, IEEE Computer Society Press, Los Alamitos, CA, USA, March 2012.

[5] S. Dolev, Y. Haviv and M. Sagiv. Self-stabilization preserving compiler. *ACM Trans. Program. Lang. Syst.*, 31, pages 22:1–22:42, August 2009.

[6] A. Dunkels. Operating Systems for Wireless Embedded Devices. In *Wiley Encyclopedia of Computer Science and Engineering*, John Wiley & Sons, Inc., 2007

[7] Texas Instruments. MSP430x1xx family user's guide (rev. f). http://www-s.ti.com/sc/psheets/slau049f/slau049f.pdf.

[8] M. A. Jaeger, G. Mühl, M. Werner and H. Parzyjegla. Reconfiguring Self-Stabilizing Publish/Subscribe Systems. In R. State, S. van der Meer, D. O'Sullivan and T. Pfeifer, editors, *Proceedings of the 17th IFIP/IEEE Intl. Workshop on Distributed Systems: Operations Management (DSOM 2006)*, volume 4269 of *Lecture Notes in Computer Science*, pages 233–238, Berlin/Heidelberg, Germany, Oct. 2006. Springer.

[9] M. A. Jaeger, G. Mühl, M. Werner, H. Parzyjegla and H.-U. Heiss. Algorithms for Reconfiguring Self-Stabilizing Publish/Subscribe Systems. Mahr and Sheng, *Autonomous Systems - Self-Organisation, Management and Control.* Springer, 2008.

[10] N. N. Mahatme, I. Chatterjee, B. L. Bhuva, J. Ahlbin, L. W. Massengill and R. Shuler. Analysis of Soft Error Rates in Combinational Logic and Sequential Logic and Implications of Hardening for Advanced Technologies. *Reliability Physics Symposium*, pages 1031–1035 , IEEE International, Amaheim, CA, USA, June 2010.

[11] T. Weis, H. Parzyjegla, M. A. Jaeger and G. Mühl. Self-organizing and Self-stabilizing Role Assignment in Sensor/Actuator Networks. In *OTM Conferences (2)*, pages 1807–1824, 2006.

[12] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. *Commun. ACM*, 53, pages 91–99, New York, NY, USA, January 2010.