Static Analysis of Lockless Microcontroller C Programs

Eva Beckschulze Sebastian Biallas Stefan Kowalewski

Embedded Software Laboratory
RWTH Aachen University, Germany
{lastname}@embedded.rwth-aachen.de

Concurrently accessing shared data without locking is usually a subject to race conditions resulting in inconsistent or corrupted data. However, there are programs operating correctly without locking by exploiting the atomicity of certain operations on a specific hardware. In this paper, we describe how to precisely analyze lockless microcontroller C programs with interrupts by taking the hardware architecture into account. We evaluate this technique in an octagon-based value range analysis using access-based localization to increase efficiency.

1 Introduction

Static analysis based on abstract interpretation [7] is a formal method that found its way into practice by several commercial code analysis tools. Proving the absence of run-time errors in microcontroller programs is of particular importance as microcontrollers are often deployed in safety-critical systems. However, C code analyzers usually do not cope with C extensions and hardware-specific control prevalent in microcontroller programs. This control is not only necessary for data input/output but also needed to implement interrupt service routines (ISRs), which allows some form of concurrency and can be used for asynchronous hardware communication and periodic tasks. Since the control functions of the hardware are often exposed through normal memory accesses, a sound analysis of microcontroller programs has to reflect these registers in its memory model. On the Atmega16 [2], for example, it is possible to enable/disable interrupts by a write to the `SREG` register which is located at the memory address `0x5F`.

Beside these peculiarities in programming microcontrollers, software engineers often rely on additional assumptions outside the scope of standard C semantics on how the compiler will translate a program and on how the microcontroller behaves w. r. t. the atomicity of some operations. For example, they might omit the locking of shared data because an 8 bit read/write is always executed atomically (such algorithms are typically called *lockless* [10] or lockfree). This saves program space on the controller as well as execution time but makes a precise analysis on C code level particularly challenging. In this paper, we deal with atomicity assumptions when analyzing interrupts. We exploit the characteristics of interrupts w. r. t. concurrency to design an efficient fixed-point computation.

1.1 Concurrency Induced by Interrupts

Compared to concurrency implemented by threads concurrency induced by interrupts exhibits some essential differences [17]. While threads can preempt each other, interrupts can preempt the main program but the main program cannot interrupt an ISR. An interrupt can only trigger if both the specific interrupt is enabled and interrupts are globally enabled. Locks to guarantee atomic sections that are not interrupted are, therefore, implemented by disabling interrupts globally. By default interrupts are disabled in ISRs such that an ISR runs to completion. Explicitly enabling interrupts in ISRs is allowed but due to the

F. Cassez, R. Huuck, G. Klein and B. Schlich (eds.):
Systems Software Verification Conference 2012 (SSV 2012)
EPTCS 102, 2012, pp. 103–114, doi:10.4204/EPTCS.102.10

limited stack size an error-prone approach. In this paper, we concentrate on programs without such nested interrupts. Considering all these specifics of interrupts we can design a more precise analysis of microcontroller C code.

1.2 Analysis Framework

We consider interrupts in the context of a data flow analysis evaluating pointers and value ranges based on the octagon abstract domain [13], in which the relations between variables (memory locations) x, y are expressed as inequalities $\pm \mathtt{x} \pm \mathtt{y} \leq c$, where c is a constant. To consider hardware dependencies, our memory model is augmented with hardware-specific knowledge [4] so as to capture, e. g., the setting or resetting of interrupt enable bits. In this paper, we show how to extend this analysis to interrupts by including hardware specifics and taking the C semantics into account.

1.3 Contribution and Outline

The contribution of this paper is twofold: (a) We develop a set of rules which lockless C programs must follow to behave predictable under different compilers. (b) We present a combined analysis of value ranges, pointers and interrupts for lockless microcontroller C programs. This analysis combines ramifications of the C memory model with understanding of the underlying hardware to allow a sound representation of lockless code.

Our paper is laid out as follows. First, Sect. 2 introduces our technique exemplified on a lockless UART driver. Then, Sect. 3 details the notion of atomicity we implemented to analyze such programs on a C and hardware-specific level. Our analysis is described in Sect. 4 and is evaluated in Sect. 5. The papers ends with a survey of related work in Sect. 6 and a concluding discussion in Sect. 7.

2 Motivating Example

In this section, we introduce a UART driver that operates without locking shared data by exploiting the atomicity of certain operations on the specific hardware architecture. We discuss different approaches to analyze such a program.

2.1 Lockless UART driver

Consider the source code excerpt in Fig. 1 implementing the receiver of a UART (Universal Asynchronous Receiver Transmitter) driver that is supposed to run on an AVR microcontroller[1]. The driver uses a FIFO rx_buff to buffer incoming data software-based in addition to the hardware-implemented buffer. An integer variable rx_in (rx_out) is used as an index to denote the position where the next byte is to be stored (where the next byte is to be read). The function getNextPos is called to increment the index by one or to reset it to 0 when the index is out of bounds. Reading a byte out of the hardware register called UDR and storing it in the FIFO buffer is performed by ISR, an interrupt service routine that is triggered by hardware. The function getByte returns the data located at position rx_out. We assume that the global interrupt enable bit is set initially and remains set all the time, while the specific interrupt used by the UART is disabled when the buffer is full (line 36) and enabled when there is at least one free position (line 27). Hence, the functions getByte and isEmpty might be interrupted anywhere in between two

[1] This is a slightly modified excerpt of the code found here http://www.mikrocontroller.net/topic/101472#882716

```
#define vu8(x)  (*(volatile uint8*)&(x))

uint8 rx_buff[RX0_SIZE];
uint8 rx_in;
uint8 rx_out;


uint8 getNextPos(uint8 pos, size){
  pos++;
  if (pos >= size){
    return 0;
  }
  return pos;
}


uint8 isEmpty(){
  return rx_out == vu8(rx_in);
}


uint8 getByte(){
  uint8 data;
  while( isEmpty() );
  data = rx_buff[rx_out];   // get byte
  rx_out = getNextPos(rx_out, RX0_SIZE);
  URX0_IEN = 1;       // enable RX interrupt
  return data;
}


ISR( USART0_RX_vect ){
  uint8 i = rx_in;
  i = getNextPos(i,RX0_SIZE);
  if( i == rx_out ){      // buffer overflow
    URX0_IEN = 0;     // disable RX interrupt
    return;
  }
  rx_buff[rx_in] = UDR;
  rx_in = i;
}
```

Figure 1: UART driver

operations. By way of contrast, the ISR always runs to completion due to the automatic global interrupt disable implemented by hardware. As both the main program and the ISR access rx_out and rx_in and one of the accesses is a write access, these variables could be subject to a data race. However, as reading and writing an 8-bit variable on an 8-bit processor architecture is atomic, locking is unnecessary in this case.

2.2 Analyzing Interrupts

A typical question verified by static analyses is that of all array accesses being within the bounds of the array. This requires a value range analysis of variables to determine possible values for variables used as indices to access an array. Interrupts have to be considered during the analysis in two ways: First, as calls to ISRs are not visible in the code, ISR code has to be added to the control flow and taken into account appropriately. For example, this can be done by nondeterministic calls to the ISR between two control flow nodes. Second, we need to deal with shared variable accesses, i. e., an access to a variable in the main program that might be performed incompletely before an interrupt is triggered. As this may result in corrupted data, care has to be taken for such race conditions to design a sound value range analysis. Next, we discuss the assumptions made during analysis, first in case the analysis is designed for an arbitrary hardware platform and second in case of hardware specifics that can be used to refine the analysis.

2.2.1 Hardware Agnostic Approach

Static analyzers for C code usually do not consider any hardware specifics such as interrupts. To analyze microcontroller C programs using generic static analyzers, the user is advised to annotate the program to provide the analyzer with further constraints. Without an appropriate annotation, for example, the analyzer will take the ISR function as dead code since it cannot see the implicit calls by the hardware. Further annotations are required to deal correctly with atomic sections that can be defined by toggling the interrupt bit in some microcontroller dependent absolute address. The analysis of ISRs is integrated into another analysis by interleaving all expressions outside of atomic sections with non-deterministic calls to ISRs. If the analysis encounters an expression in the main program that accesses a variable that is also accessed by an ISR and one of the accesses is a write access then there is a race condition. Therefore, to be sound, a value range analysis unaware of the hardware architecture has to assume that this variable may take any value within its type bounds. We find such race conditions in the UART example in Fig. 1:

- Read access to `rx_in` in line 18, concurrent write access to `rx_in` in line 40
- Write access to `rx_out` in line 26, concurrent read access to `rx_out` in line 35

Unfortunately, assuming type bounds for these variables that are used as an index to the buffer `rx_buffer` results in a presumed array out of bounds access which is spurious. In the next section we discuss how considering a hardware-specific memory model can refine the analysis and avoid this false alarm.

2.2.2 Considering Hardware Specifics

Programming microcontrollers is intrinsically tied to dealing with the specific hardware. Taking hardware specifics into account as well when designing static analyses avoids tedious user annotations while increasing precision of analyses. In a hardware-specific memory model, an access to an absolute address (register) is linked to the semantics of this register [4]. This way, interrupts can be identified and added automatically to the control flow wherever they may occur. We delay the discussion of reducing the number of ISR analyses further to Sect. 4. With respect to the shared read/write access mentioned above, hardware considerations enhance the precision of the analysis significantly. Knowing that the target platform is an 8-bit architecture, we conclude that reading or writing to 8-bit variables will always be performed atomically. Thus, `rx_in` and `rx_out` always have consistent values. To make sure that all possible values are considered when reading `rx_in`, it is sufficient to compute a fixed-point over the ISR in advance and propagate it non-deterministically. Similarly, before writing to `rx_out`, the analysis of the ISR takes account of the old abstract value of `rx_out` while the analysis of the ISR after the write access will consider its new abstract value. Note that it is not the C expression that we assume to be atomic, but the eventual load or store instruction executed by the processor. In the following section, we will detail this notion of atomicity.

3 Requirements for Lockless Code

Precisely analyzing shared data in concurrent programs on a high abstraction level such as C code is usually not possible as we are unaware how a compiler translates a C code expression into processor instructions. On the other hand, stable concurrent programs should not depend on a certain compiler version or compilation options. In this section, we try to infer basic rules under which lockless C programs can operate in a well-defined manner. We then enrich these rules by certain hardware specifics: In the last section, e. g., we argued that writing or reading to an 8-bit variable cannot be interrupted on an 8-bit hardware architecture, which gives rise to a basic form of atomic access.

With the rules derived in this section, we achieve two goals: First, we can detect program errors (which might manifest themselves depending on compiler specifics) by checking whether a program follows these rules. Second, we can formulate the foundations of a sound and precise analysis for each program which follows these rules.

3.1 Atomicity at the Level of C

In the absence of locks, we assume that all data shared between the main function and the ISRs is performed using volatile accesses only (i. e., either by casting the access to a volatile data type or declaring the variable volatile in the first place). Accessing non-volatile shared data is prone to failure and thus reported, since an optimizing compiler might eliminate seemingly unnecessary reloads and dead stores.

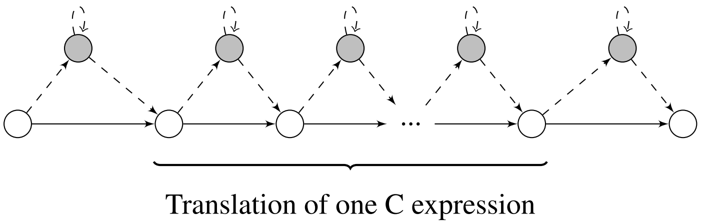

Figure 2: Possible Control Flow on Instruction Level

In the following, we call a C expression *competing* if it contains an access to volatile data. The term competing stresses that competing expressions should be evaluated in a well-defined order.

C compilers are allowed to schedule loads, stores and calculations of expressions in arbitrary order as long as this does not alter the *visible* behavior of the program (as-if rule). In the presence of volatile accesses, however, at least all volatile objects must be stable at sequence points [11, Sect. 5.1.2.3]. To illustrate this, consider the shared variables a, b and the statement a = ++b; which writes twice to shared data between sequence points. Let us further assume that we have the precondition a $\leq$ b. Although we know that a and b must be stable after this statement, the assignment operator forms no sequence point, and a compiler might thus translate this expression into one of the following two pseudo-assembly snippets:

```
1 LOAD  temp, b                                       1 LOAD  temp, b
2 INC   temp                                          2 INC   temp
3 STORE b, temp                                       3 STORE a, temp
4 STORE a, temp                                       4 STORE b, temp
```

In the right snippet, a is stored before b. An ISR invoked between line 3 and line 4 might thus observe that a $>$ b, while a $\leq$ b is an invariant in the left snippet. Hence, we want to detect and warn about such statements.

Now, consider an arbitrary C expression between two sequence points. Fig. 2 shows the control flow graph for such an expression assuming that the compiler translated it into a certain sequence of instructions (white nodes), where each instruction might be followed by one or more calls to an ISR (gray nodes). Between the two sequence points, we do not know how the compiler translated the expression. This gives rise to two requirements for writing stable lockless code:

1. The effect of the execution of an ISR between two sequence points must be covered by executing the ISR at the sequence point before or after the expression.
2. The effect of the execution of an ISR must not depend on the exact instructions generated by the compiler.

The first requirement stems from the fact that the analysis – as well as the programmer – can predict the program state only at sequence points, and thus, the ISR behavior should only depend on this predictable state. The second requirement can be regarded as a corollary of the first: If a compiler creates a different set of instructions (because of, e. g., different options), we still want requirement 1 to hold. We call expressions that fulfill these requirements *well-formed*.

Observe that the well-formedness of expressions depends on two distinct properties: How much freedom the compiler has to translate an expression (especially scheduling loads and stores) and which atomic primitives are provided by the underlying hardware. We will defer the discussion of the latter to Sect. 3.2 and for now assume that all loads and stores of shared variables cannot be interrupted by the ISRs. We will first formally define well-formed expressions inductively:

1. A constant expression $expr ::= const$ and a variable expression $expr ::= v$ is well-formed.
2. A unary expression $expr ::= \ominus expr_1$ with $\ominus \in \{-, !, \tilde{}\}$ is well-formed iff $expr_1$ is well-formed.
3. Let $expr ::= expr_0 \odot expr_1$ be a binary expression, $\odot \in \{+, -, /, *, \%, \ll, \gg, |, \&, \hat{}, <, <=, >, >=, ==, !=\}$. Then, $expr$ is well-formed iff (a) $expr_0$ and $expr_1$ are well-formed and (b) at most one of $expr_0$ and $expr_1$ is competing.
4. Let $expr ::= expr_0 \bowtie expr_1$ be a comma or logic expression, $\bowtie \in \{||, \&\&, \text{“,”}\}$. Since $\bowtie$ forms a sequence point, $expr$ is well-formed iff $expr_0$ and $expr_1$ are well-formed.
5. Let $expr ::= \texttt{fun}(expr_0, \ldots, expr_n)$ be a function call. Function calls are sequence points, yet the order in which the arguments are evaluated is not defined. Thus, $expr$ is well-formed if $expr_0, \ldots, expr_n$ are well-formed and at most one $expr_i$ is competing. Additionally, $expr$ is competing iff the body of `fun` accesses shared data.
6. An assignment $expr ::= lvalue = expr_1$ is well-formed iff (a) $lvalue$ is not competing and $expr_1$ is well-formed or (b) $expr_1$ is well-formed and does not write shared data (this does not span into functions which are being called).

Intuitively, a well-formed expression is an expression in which the order of all accesses of shared data is determined by the C standard, i. e., the evaluation of competing expressions is ordered by sequence points. Additionally, we require that we have at most one write to shared data in a well-formed expression (follows from 6.). Furthermore, note that the second part of bullet point 5. forbids delicate cases such as `a = f()+g()` where `f` and `g` access shared data (the order in which `f` and `g` are invoked is not defined). Yet, expressions such as `a = f()+1` are well-formed even if `a` is shared and `f` is competing.

The latter point deserves a more detailed study: The assignment operator = does not form a sequence point. Hence, expressions such as `a = b = 0` are not well-formed, since the store to `a` and `b` can be performed in arbitrary order. However, the compiler must only generate one store to the left-hand-side of an assignment, which depends on the evaluation of the right-hand-side. Thus, if the right-hand-side contains no writes to shared data, such as in `a = b`, an assignment is well-formed. We allow this construct because it is typically used in lockless code. Alternatively, this construct could be avoided by introducing temporary (unshared) variables.

The design rules that we derive to achieve robustness of lockless programs against compiler reordering are simple. Function call expressions should be either full expressions or should adhere to bullet point 5. Further, the programmer is advised to avoid subexpression with side effects (accessing a volatile object is a side effect, too). By splitting up complex expression in several simple ones without side effects in subexpression or by at least encapsulating them in simple assignments the desired order of evaluation is made explicit. These are well-known design rules that usually aim at well-defined (compiler-independent) behavior of single-threaded code. However, it also contributes to well-formed expressions which matter in lockless concurrent programs only.

Finally, we require the compiler to behave *sensible*. That is, in essence, that loads and stores to volatile data are translated to elementary load and store operations (one instruction where possible). Note that this requirement is, to the best of our knowledge, fulfilled by all compilers used in industry and is exactly what a programmer expects – and exploits – when writing lockless code. In the next section, we will connect the general requirements of C to the specific offerings of the hardware to create a concise memory model.

3.2 Atomicity on the Hardware Level

Crucial for our approach is a hardware model that reflects atomicity at the assembly level. In the last section, we assumed that each elementary load and store of shared data can be performed atomically,

i. e., it cannot be interrupted. Depending on the type and size of the data object, this is in general not the case. However, accessing data of the size of general purpose registers can usually be performed by one instruction, which then is not interruptible. Loading and storing data of different sizes, on the other hand, is usually performed by a sequence of instructions. Thus, an ISR might interrupt this sequence and read or modify corrupted data. Our analysis allows to configure the size of atomic data types beforehand and aims to detect cases where atomic accesses are required but not possible.

Additionally, some processors provide atomic primitives such as compare-and-swap as one instruction. These instructions can only be accessed using compiler specific functions, which are usually built-in or provided as inline assembly. Our analysis can simulate these instructions atomically if the functions provided by the compiler are annotated appropriately. With this structure in place, we can now describe our analysis in the context of interrupts and lockless code.

4 Designing Analyses Considering Interrupts

Analyzing ISRs and the sound handling of shared data requires adaptations of the existing data flow analysis. In the following, we first describe the original analysis and subsequently discuss how to integrate ISR analysis and shared data handling.

4.1 Original Analysis

The original analysis is a combined analysis of pointers and value range analysis based on octagons [13]. It is a fixed-point computation iterating over the nodes in the control flow graph (CFG) and propagating the computed results along control flow edges until old and new results coincide. The analysis is flow and context sensitive, i. e., it determines an abstract value for each node in the CFG and distinguishes different calling contexts for function calls. The octagon analysis evaluates nodes concerning value ranges while an address taking node is handled by the pointer analysis.

In order to increase efficiency of the octagon based analysis, we apply access-based localization, a technique that reduces the abstract state that is propagated when analyzing a function to the subset of the abstract state that is actually accessed in this function or in subroutines [3, 15]. The set of accessed variables in each function is over-approximated by a pre-analysis based on a flow-insensitive pointer analysis.

4.2 Invoking Interrupts in Fixed-Point Computation

For efficiency reasons, ISRs shall only be analyzed if necessary. Therefore, we make use of our memory model [4] in order to find accesses to absolute addresses that correspond to setting or clearing the global or individual interrupt bits. The current status of the interrupt bits is added to the abstract state propagated along the CFG. The naïve approach would be to trigger ISR analysis in between two arbitrary nodes where the interrupt is enabled. However, the execution of an ISR does not necessarily affect all subsequent operations of the main program. Precisely stated, an ISR affects an operation in the main program if and only if the operation accesses data that might be written by the ISR. In order to incorporate such dependencies ISRs have to be analyzed before analyzing a shared data access. In our analysis, we trigger ISR analysis after each node that enables interrupts (globally or individually). This way we also ensure that ISRs are analyzed at least once between two atomic sections.

With respect to the control flow, the ISR analysis corresponds to an extra node calling the ISRs added between an interrupt-enabling node and its successor node while the direct connection between these

```
main(){
  ...
  sei();        //interrupt enable
  ...
  a = b;        //shared data access
  ...
  cli();        //interrupt disable
  ...
}

ISR(){
  ...
  b = c;        //shared data access
  ...
}
```

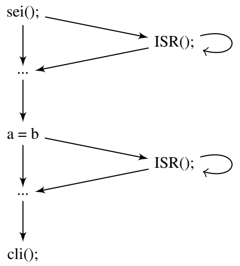

Figure 3: Code example (left) and its control flow graph (right) enhanced with necessary non-deterministic calls to the ISR that need to be considered in the analysis

nodes is kept to consider the case that no interrupt is triggered (cf. Fig 3). The analysis applies the join operation to the abstract states of both branches. We compute a fixed-point over all ISRs that might be triggered considering every possible order and frequency of interrupts in order to over-approximate the number of possibly occurring interrupts. Indeed, on an AVR microcontroller each ISR execution is followed by at least one main program instruction. However, the exact number of possible ISR executions cannot be determined on C level, as we are unaware how the compiler translates a given code sequence into instructions.

Still, analyzing interrupts at one location between two atomic sections is not sufficient if shared data is written in the main program and read by the ISR. Therefore, we trigger another analysis of ISRs after analyzing a shared access node (cf. Fig 3). As previously noted, interrupts in the analysis are considered non deterministic but possibly occurring infinitely often. Note that analyzing ISRs both before and after a shared access node is sufficient if the requirements in Sect. 3.1 are fulfilled.

Finally, to speed up the analysis, we apply access-based localization as explained above also to ISRs. For this purpose, we arrange the pre-analysis collecting accessed variables to analyze ISRs as well. Additionally, we subdivide accessed variables into read and written variables in order to be able to distinguish the different cases of shared access.

4.3 Sound Analysis of Shared Data

Shared data handling depends on whether a full expression (cf. [11, Sect. 6.8]) is well-formed. As our control flow nodes often represent subexpressions of a full expression, we add to each node whether the corresponding full expression is well-formed. Table 1 reviews the cases of shared accesses and indicates the behavior of the analysis. We omitted the case where both the main program and an ISR only read the same variable as it does not raise any problem.

The first row of the table shows the case that the shared accesses are atomic, the main program writes(reads) and an ISR read (writes) the shared variables and the corresponding full expression is well-formed. In this case the octagon analysis is performed as usual. All issues of concurrency are in this case considered by triggering the ISR analysis somewhere before and immediately after this node. In the second row we consider a well-formed full expression that includes an atomic write of a variable that is also written in an ISR. Though it does not cause corrupted data, one write may immediately overwrite the other one. As this might be unintended by the programmer, we issue a warning that data loss might occur.

The third row considers the case that the analysis encounters a full expression where all shared accesses are atomic but the expression is not well-formed. In this case the analysis sets all shared variables to type

atomic access	access type	well-formed expression	analysis behavior
yes	r/w	yes	no special behavior
yes	w/w	yes	issue a warning
yes	*	no	set shared variables to type bounds, issue a warning
no	*	*	set shared variables to type bounds, issue a warning

Table 1: Cases of shared access

Program	**Loc**	**# global vars**	**Time**	**# of ISR analyses**	**# Warnings**	**# Warnings**
					(legitimate)	(spurious)
UART buffer	175	32	7.3 s	136	0	1
RGB-LED	95	22	0.8 s	27	1	0
Traffic Light	68	5	0.1 s	30	0	0

Table 2: Results of the case study

bounds thereby overapproximating any possible order of execution. Note that these overapproximations are also propagated to the subsequent ISR analysis. We issue a warning that this expression is unspecified behavior. The case that the access is non-atomic (row 4) is handled by the analysis the same way. Here, we issue the warning that a non-atomic access might result in inconsistent or corrupted data.

5 Case Studies

Our implementation is written in JAVA and builds on the Eclipse Framework. We used it to analyze several microcontroller programs written for the AVR ATmega 16 microcontroller unit. For this processor our analysis assumes that only 8 bit memory accesses are performed atomically.

The results of the case study are presented in Table 2. For each checked program, we provide the lines of code, number of global variables, time for analysis, number of times we have to check the ISRs and the number of (spurious) warnings. The analyzer was able to prove the absence of array out of bounds accesses in the UART buffer. Yet, we still had a spurious warning, since this program writes to the data buffer without locking from both the ISR and the main function. The correctness of such an operation has to be checked manually. In the RGB-LED program, we found an unlocked shared access to a 16-bit variable, which was a legitimate warning. Finally, the Traffic Light program, controlling an intersection with two traffic lights, could be checked without triggering a warning.

6 Related Work

Traditionally, model checking has been used to verify concurrent programs such as in [9] where partial order reduction is used to increase efficiency. Schlich et al. [18] implement this technique for model checking embedded software on the assembly level. Atig et al. [1] describe how to model check in the presents of a weak memory model, which corresponds to the lockless programs described in this paper.

Regehr and Cooprider [17] describe how to map microcontroller programs (interrupt-driven code) to thread-driven code. In particular, they point out the differences between threads and interrupts and show

how to exploit existing techniques for verification tools for multithreaded code to verify interrupt-driven embedded software. Cooprider [6] describes how to increase efficiency by only analyzing ISRs at certain locations. His approach, however, is restricted to properly locked programs.

Pratikakis et al. describe in [16] the LOCKSMITH tool which can check automatically for proper locking of shared data in an abstract interpretation framework. We focus on the verification of microcontroller programs even in the absence of locks.

Recently, Miné presented an extensive work [14] on analyzing embedded real-time parallel C programs based on abstract interpretation. He defines the semantics of expressions based on interference, i. e., whenever a thread reads a variable `x`, all abstract values another thread might set `x` to are also considered. These interference sets are non-relational and flow-insensitive information while we interchange relational and flow-sensitive information between the main program and ISRs. Due to the flow-insensitivity dealing with the order of execution of C expressions is superfluous in Miné's approach, while it is essential in our approach. Additionally, his approach differs from ours in considering all primitive statements to be atomic independent of types and the underlying hardware. This way shared data access is not handled correctly in case of incomplete assignments.

In [8] Fehnker at al. extend the generic (unsound) static analyzer Goanna to detect microcontroller specific bugs. In their approach, the CFG is labeled with occurrences of syntactic constructs of interest, while the desired behavior is put into a CTL formula that can be checked by model checking techniques. Their work focuses on integrating hardware information to specify and check simple rules that should be followed when programming the specific microcontroller. Instead, this paper advocates the use of a hardware-specific memory model to enhance precision of data flow analyses and to avoid false alarms.

Finally, there is an ongoing effort to formalize memory models for existing languages, which recently cumulated in a memory model for the new C++ standard [5]. Yet, programmers of microcontroller software still rely on non-strictly defined semantics and "sensible" compiler behavior mixed with knowledge about the underlying hardware. Our approach aims to implement these assumptions, which are sometimes quite subtle, in a verification framework.

7 Concluding Discussion

This paper advocates a static analysis for lockless microcontoller C programs combining different techniques to make the approach precise as well as tractable. To achieve precision for such programs, it is necessary to deal both with C semantics and hardware specifics. Our memory model reflects what the user can (and will) expect from the compiler on the one side and what atomic primitives the hardware can provide on the other side. Using the derived rules for well-formed expressions, we can detect latent bugs that would manifest themselves only in certain circumstances (such as different compiler options) and we can detect bugs that result from improper communication between the main program and the ISR using our fine-grained value analysis.

Still – as Meyers and Alexandrescu [12] point out – thread-unaware languages such as C pose inherent difficulties to write thread-aware code. It makes proper (manual) synchronization exceptionally hard since all corner cases of the languages have to be considered. Additionally, as we have shown in this paper, it poses obstacles for the program analysis, since the order of certain operation is often unclear. In such cases, either imprecise non-relational analyses have to be deployed or a careful analysis of all C expressions is necessary.

Acknowledgements

This work was supported, in part, by the DFG research training group 1298 *Algorithmic Synthesis of Reactive and Discrete-Continuous Systems* and by the DFG *Cluster of Excellence on Ultra-high Speed Information and Communication*, German Research Foundation grant DFG EXC 89. Further, the work of Sebastian Biallas was supported by the DFG. We thank Volker Kamin for sharing his thoughts on the ideas described in this paper, and the anonymous referees for their helpful comments.

References

[1] M. F. Atig, A. Bouajjani, S. Burckhardt & M. Musuvathi (2010): *On the verification problem for weak memory models*. In: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '10, ACM, New York, NY, USA, pp. 7–18, doi:10.1145/1706299.1706303.

[2] Atmel Corporation (2009): *The Atmel 8-bit AVR Microcontroller with 16K Bytes of In-system Programmable Flash*. www.atmel.com/atmel/acrobat/doc2466.pdf.

[3] E. Beckschulze, J. Brauer & S. Kowalewski (2012): *Access-Based Localization for Octagons*. Electronic Notes in Theoretical Computer Science 287(0), pp. 29 – 40, doi:10.1016/j.entcs.2012.09.004. Proceedings of the Fourth International Workshop on Numerical and Symbolic Abstract Domains, (NSAD 2012).

[4] E. Beckschulze, J. Brauer, A. Stollenwerk & S. Kowalewski (2011): *Analyzing Embedded Systems Code for Mixed-Critical Systems Using Hybrid Memory Representations*. In: 14th IEEE Int. Symp. on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, IEEE, pp. 33–40, doi:10.1109/ISORCW.2011.40.

[5] H.-J. Boehm & S. V. Adve (2008): *Foundations of the C++ concurrency memory model*. In: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, PLDI '08, ACM, New York, NY, USA, pp. 68–78, doi:10.1145/1375581.1375591.

[6] N. Cooprider (2008): *Data-flow analysis for interrupt-driven microcontroller software*. Ph.D. thesis, The University of Utah, Salt Lake City, UT, USA.

[7] P. Cousot & R. Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In: POPL, ACM Press, pp. 238–252, doi:10.1145/512950.512973.

[8] A. Fehnker, R. Huuck, B. Schlich & M. Tapp (2009): *Automatic Bug Detection in Microcontroller Software by Static Program Analysis*. In: SOFSEM 2009: Theory and Practise of Computer Science,Spindleruv Mlýn, Czech Republic, Lecture Notes in Computer Science 5404, Springer, pp. 267–278, doi:10.1007/978-3-540-95891-8_26.

[9] P. Godefroid (1996): *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. LNCS 1032, Springer, doi:10.1007/3-540-60761-7.

[10] T. E. Hart, P. E. McKenney & A. D. Brown (2006): *Making Lockless Synchronization Fast: Performance Implications of Memory Reclamation*. In: 20th IEEE International Parallel and Distributed Processing Symposium, Rhodes, Greece, doi:10.1109/IPDPS.2006.1639261.

[11] ISO (1999): *ANSI/ISO/IEC 9899-1999: Programming Languages — C*. International Organization for Standardization, Geneva, Switzerland.

[12] S. Meyers & A. Alexandrescu (2004): *C++ and the perils of double-checked locking*. http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf.

[13] A. Miné (2006): *The Octagon Abstract Domain*. Higher-Order and Symbolic Computation 19(1), pp. 31–100, doi:10.1007/s10990-006-8609-1.

[14] A. Miné (2012): *Static Analysis of Run-Time Errors in Embedded Real-Time Parallel C Programs*. Logical Methods in Computer Science 8(1–26), pp. 1–63.

[15] H. Oh, L. Brutschy & K. Yi (2011): *Access analysis-based tight localization of abstract memories*. In: *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, Springer-Verlag, Berlin, Heidelberg, pp. 356–370, doi:10.1007/978-3-642-18275-4_25.

[16] P. Pratikakis, J. S. Foster & M. Hicks (2011): *LOCKSMITH: Practical static race detection for C*. *ACM Trans. Program. Lang. Syst.* 33(1), pp. 3:1–3:55, doi:10.1145/1889997.1890000.

[17] J. Regehr & N. Cooprider (2007): *Interrupt Verification via Thread Verification*. *Electron. Notes Theor. Comput. Sci.* 174(9), pp. 139–150, doi:10.1016/j.entcs.2007.04.002.

[18] B. Schlich, T. Noll, J. Brauer & L. Brutschy (2009): *Reduction of Interrupt Handler Executions for Model Checking Embedded Software*. In: *Haifa Verification Conference (HVC 2009), Haifa, Israel*, *LNCS* 6405, Springer, pp. 5–20, doi:10.1007/978-3-642-19237-1_5.