

# Interval Analysis of Microcontroller Code using Abstract Interpretation of Hardware and Software

Jörg Brauer  
Embedded Software Laboratory  
RWTH Aachen University  
Ahornstr. 55, D-52074 Aachen  
brauer@embedded.rwth-aachen.de

Thomas Noll  
Software Modelling and Verification Group  
RWTH Aachen University  
Ahornstr. 55, D-52074 Aachen  
noll@cs.rwth-aachen.de

Bastian Schlich  
Embedded Software Laboratory  
RWTH Aachen University  
Ahornstr. 55, D-52074 Aachen  
schlich@embedded.rwth-aachen.de

## ABSTRACT

Static analysis is often performed on source code where intervals – possibly the most widely used numeric abstract domain – have successfully been used as a program abstraction for decades. Binary code on microcontroller platforms, however, is different from high-level code in that data is frequently altered using bitwise operations and the results of operations often depend on the hardware configuration. We describe a method that combines word- and bit-level interval analysis and integrates a hardware model by means of abstract interpretation in order to handle these peculiarities. Moreover, we show that this method proves powerful enough to derive invariants that could so far only be verified using computationally more expensive techniques such as model checking.

## Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—*formal methods*; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning About Programs—*mechanical verification*; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—*program analysis*

## General Terms

Algorithms, Theory, Verification

## Keywords

Static analysis, abstract interpretation, interval analysis, embedded systems, binary code

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SCOPES '10, June 28-29, 2010, St. Goar, Germany  
Copyright 2010 ACM 978-1-4503-0084-1/10/06 ...\$10.00.

## 1. INTRODUCTION

Abstract interpretation can be applied to the systematic construction of methods and algorithms for approximating undecidable or complex problems in program analysis [8]. Particularly, abstract interpretation involves methods for replacing an analysis on the concrete program semantics by an analysis on a simpler, more abstract, domain. Intervals are probably the most widely used numeric domain for approximating the values of program variables, and have long been considered a suitable domain for static analysis of programs written in high-level languages [7], mainly for two reasons. Firstly, intervals often provide enough precision to prove interesting program properties, for instance, that all array accesses are confined to the bounds of the corresponding array. Secondly, the interval abstract domain is computationally attractive.

Contrary to existing work, our approach is tailored to disassembled binary programs for the ATMEL ATmega16 microcontroller, which differ strongly from high-level programs [2, 3, 26]. A particularly intricate feature of this microcontroller is that registers and the main memory are mapped to the same address space. An interesting problem for any verification tool for this platform is therefore to show that indirect store operations do not accidentally overwrite the contents of registers, which is analogous to proving that array accesses in high-level languages such as C respect the array bounds. In binary code, however, registers are often transformed using bitwise instructions – such as OR (bitwise or), EOR (bitwise exclusive-or), AND (bitwise and), or NEG (bitwise negation) – for which word-level intervals are not well-suited: They lead to coarse over-approximations of the actual value sets. Moreover, control logic is implemented using branching instructions that depend on certain bits of the status register such as the carry flag or the zero flag. This necessitates reasoning about values at the granularity of bits.

In this paper, we describe how to combine word- and bit-level analyses in order to obtain tight over-approximations of value sets for microcontroller binary code. The analysis has been integrated into [MC]SQUARE<sup>1</sup>, which is a verification platform for microcontroller code that supports model

<sup>1</sup><http://mcsquare.embedded.rwth-aachen.de>

checking and static analysis. In [MC]SQUARE, static analysis is used for finding bugs and providing results used in static state-space reduction methods such as partial order reduction [27], dead variable reduction, and path reduction [30]. While the target platform considered in this paper is the ATMEL ATmega16 8-bit microcontroller [1], the approach is easily transferable to other platforms.

## 1.1 Motivation

In the following, we detail four peculiarities of microcontroller binary code that motivate the construction of our analysis:

### *Word- and bit-level intervals.*

Intervals for representing value ranges of variables are naturally well-suited for analyzing high-level programs that contain arithmetic operations. While parts of binary code have a similar structure, for instance, loops where a loop counter is incremented and compared to a bound, many instructions alter only single bits of registers. Suppose, for instance, that only bit 6 of register `r0` is accessed in an instruction, and this bit can hold either 0 or 1, while all other bits are cleared. In an interval representation for register `r0`, this would yield the interval  $[0, 2^6]$ , which is a coarse over-approximation of the actual value set  $\{0, 2^6\}$ .

On the other hand, if each bit is represented as an interval with possible values in  $[0, 1]$ , the concrete values  $\{127, 128\}$  of a register would lead to the bitwise representation  $[0, 1]^8$ , which corresponds to the value set  $\{0, \dots, 255\}$  upon concretization.

### *Loop conditions.*

A major source for precision in (non-relational) interval analysis is the presence of loop conditions, which allow to bound ranges of values accessed in loops, etc. In binary code, however, control logic is formulated in terms of status flags such as the carry or negative flag, and the path taken depends on the outcome of some preceding instructions. As an example, consider the instruction `BRNE k`, which increases the program counter by `k+1` iff either the negative flag or the overflow flag in the status register is set. If this is not the case, the program counter is incremented by 1. Therefore, conditionals cannot be handled by computing the intersection between the intervals of the variable accessed in the loop and the naive branching condition as obtained from the program code.

### *Indirect stores.*

An indirect store is an operation in which the contents of one register are stored at a target address that is held in another register. On microcontrollers such as the ATMEL ATmega16, registers are reserved locations in the same address space as the SRAM. Thus, it is possible to mutate a register, such as the stack pointer or the status register, if the target coincides with the address of the register. Existing analyses often assume that indirect writes never alter registers [24]. Though appealing in its simplicity, this assumption is dubious for handcrafted code, and it is not unknown for compilation itself to introduce errors, particularly in the field of embedded systems. The problem of reasoning about targets of indirect stores is compounded by the fact that such operations often arise in loops.

### *Hardware dependencies.*

Furthermore, an analysis of microcontroller binary code needs to take care of hardware dependencies. On the ATMEL ATmega16, for instance, four I/O ports can be used for communicating with peripherals such as sensors. Each port (denoted `x`) has three associated registers: a data register `PORTx`, a data direction register `DDRx`, and a port input register `PINx`. If the  $n$ -th bit of `DDRx` is set, the  $n$ -th bit of `PINx` is configured as an output pin. This means that it stores the value that was written to `PORTx`. If it is cleared, the  $n$ -th bit of `PINx` is configured as an input pin, and has a nondeterministic value when it is read. Modeling the behavior of the microcontroller hardware for the analysis instead of resorting to nondeterminism whenever an I/O register is accessed is a key for obtaining precise analysis results and detecting false usage of hardware features such as writing to reserved registers.

## 1.2 Contributions

All four challenges are handled by our approach. In summary, we make the following contributions:

- We describe an interval analysis for binary code supporting both arithmetic and logical instructions by combining word- and bit-level intervals (cf. Sect. 3 and Sect. 4). In our approach, we combine word- and bit-level interval domains based on the *reduced product* [8] in order to precisely handle both kinds of operations, relying on the precision of one domain in case that precision is lost in the other.
- In microcontroller binary code, addresses of the available memory are fully determined (the ATmega16 has 1120 bytes of addressable memory), and hence, when verifying such code, it is not necessary to use symbolic memory representations: an address will suffice. We explain how to model indirect reads and stores using interval abstractions.
- We detail how to integrate conditional branching using a safe heuristics based on pattern matching and a path-based fallback solution (cf. Sect. 4.3).
- We explain how behavior of the hardware depending on its current memory configuration is integrated into the analysis using the notion of *environment transformers*, which model the values of I/O registers that depend on the values of other registers (cf. Sect. 5).
- While a widening approach is not necessary to ensure termination due to finiteness of the analysis domains, we describe a simple widening operator for intervals based on *monotone cycles* in the program dependence graph [16] in order to speed up fixed point computation (cf. Sect. 6).
- We evaluate the performance and precision of our approach in a case study. In particular, we compare our results to those obtained for an automotive application using model checking. Furthermore, we describe the integration with static state-space reduction methods such as dead variable reduction and path reduction (cf. Sect. 7).

The only prerequisite for our approach is that the binary program has successfully been disassembled, including a reconstruction of the complete control flow graph, which we consider reasonable, acknowledging the recent advances in the field of intermediate-representation recovery [20]. Reconstructing the control flow is a challenging problem in itself, but orthogonal to our work.

The remainder of this paper is structured as follows. First, a short description of preliminaries with respect to abstract interpretation is given in Sect. 2. Then, Sect. 3 explains the construction of the analysis domain for word- and bit-level interval reasoning, followed by a description of the analysis for the instruction set of the ATmega16 in Sect. 4. This section also contains an overview of the loop heuristics that is used in order to infer branching conditions and details our tactic for handling pseudo-concurrency introduced by interrupt handlers. In Sect. 5, the concept of environment transformers to represent hardware-dependent behavior in the analysis is introduced. A case study that highlights the effectiveness of our approach is presented in Sect. 7. Finally, related work is described in Sect. 8.

## 2. PRELIMINARIES

Our approach for interval analysis is based on the theory of abstract interpretation [8], which is a methodology that allows to replace concrete computations with more abstract ones whilst preserving correctness by construction. The key idea in abstract interpretation is to simulate the execution of each concrete operation  $g : L \rightarrow L$  with an abstract analogue  $f : M \rightarrow M$ , where  $L$  and  $M$  are domains of concrete values and descriptions. In order to keep the paper self-contained, this section repeats some fundamental results.

Given a set  $L$ , a *partial order* on  $L$  is a transitive, reflexive, and anti-symmetric binary relation  $\sqsubseteq$  on  $L$ . Then,  $(L, \sqsubseteq)$  is called a *partially ordered set*. For  $L' \subseteq L$ ,  $l \in L$  is an upper (lower) bound of  $L'$  if  $l' \sqsubseteq l$  ( $l \sqsubseteq l'$ ) for all  $l' \in L'$ . A *least upper bound* (*greatest lower bound*)  $l'$  of  $L'$  is an upper (lower) bound of  $L'$  that satisfies  $l' \sqsubseteq l_0$  ( $l_0 \sqsubseteq l'$ ) whenever  $l_0$  is another upper (lower) bound of  $L'$ . A complete lattice is a partially ordered set  $(L, \sqsubseteq)$  such that all subsets of  $L$  have least upper bounds and greatest lower bounds. For  $L' \subseteq L$ , the greatest lower bound (least upper bound) of  $L'$  is denoted by  $\sqcap L'$  ( $\sqcup L'$ ).

In the following, let  $(L, \sqsubseteq_L)$  and  $(M, \sqsubseteq_M)$  be complete lattices. Then  $(L, \alpha, \gamma, M)$  with  $\alpha : L \rightarrow M$  and  $\gamma : M \rightarrow L$  is called a *Galois connection* if  $\alpha$  and  $\gamma$  are total functions and  $\alpha(l) \sqsubseteq_M m$  iff  $l \sqsubseteq_L \gamma(m)$  for all  $l \in L$  and  $m \in M$ . The function  $\alpha$  is called an *abstraction* and  $\gamma$  is called a *concretization*.

Galois connections can be combined in a component-wise manner using the *reduced product* operation [8]. Given two Galois connections  $(L, \alpha, \gamma, M_1)$  and  $(L, \alpha, \gamma, M_2)$ , the reduced product is the Galois connection  $(L, \alpha, \gamma, \psi(M_1 \times M_2))$  where  $\alpha(l) = (\alpha_1(l), \alpha_2(l))$ ,  $\gamma(m_1, m_2) = \gamma_1(m_1) \sqcap \gamma_2(m_2)$ , and  $\psi(m_1, m_2) = \sqcap \{(m'_1, m'_2) \mid \gamma_1(m_1) \sqcap \gamma_2(m_2) = \gamma_1(m'_1) \sqcap \gamma_2(m'_2)\}$ . That is, abstraction is performed independently, but concretization yields only those elements that are concretized from both  $M_1$  and  $M_2$ .

The force of this approach is that information lost in one domain can be recovered from the other upon concretization. The reduced product will be used in the following to design an interval analysis that provides enough expressiveness to handle arithmetic as well as logical operations.

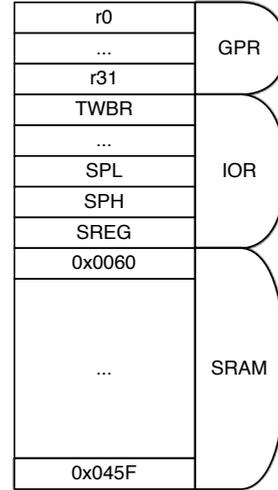


Figure 1: Memory layout of the ATmega16 microcontroller

## 3. ANALYSIS DOMAINS

The ATMEL ATmega16 is an 8-bit microcontroller platform that features 32 general-purpose registers, 64 I/O registers, and 1024 bytes of SRAM, leading to 1120 addressable memory locations, which are all mapped to the same address space (cf. Fig. 1). Furthermore, all registers can be accessed using both arithmetic and logical instructions, but the SRAM can only be accessed using word-level copy-instructions. Consequently, each memory location is represented by two abstract elements: one interval covering values between 0 and 255 and one 8-tuple of 0/1 intervals. The construction of these two domains is covered in the remainder of this section.

### 3.1 Word-Level Intervals

The domain of word-level intervals consists of elements  $[l, u]$  with  $l, u \in \mathbb{N}_{<256}$  and  $l \leq u$ , and additionally the empty interval  $\emptyset$ . We denote this domain, which represents all possible values of an unsigned 8-bit register, by  $\mathcal{I}_{256}$ . The join of two intervals  $[l_1, u_1], [l_2, u_2] \in \mathcal{I}_{256}$  is defined in the natural way, that is,  $[l_1, u_1] \sqcup [l_2, u_2] = [\min\{l_1, l_2\}, \max\{u_1, u_2\}]$ , and  $(\mathcal{I}_{256}, \sqcup)$  with least element  $\perp = \emptyset$  and greatest element  $\top = [0, 255]$  forms a complete lattice. A Galois connection  $(2^{\mathbb{N}_{<256}}, \alpha_{2s}, \gamma_{2s}, \mathcal{I}_{256})$  between  $2^{\mathbb{N}_{<256}}$  and  $\mathcal{I}_{256}$  with  $\alpha_{2s} : 2^{\mathbb{N}_{<256}} \rightarrow \mathcal{I}_{256}$  and  $\gamma_{2s} : \mathcal{I}_{256} \rightarrow 2^{\mathbb{N}_{<256}}$  can be defined as follows for  $N \subseteq \mathbb{N}_{<256}$  and  $[l, u] \in \mathcal{I}_{256}$ :

$$\begin{aligned} \alpha_{2s}(N) &= [\min(N), \max(N)] \\ \gamma_{2s}([l, u]) &= \{n \in \mathbb{N}_{<256} \mid l \leq n \leq u\} \end{aligned}$$

### 3.2 Bit-Level Intervals

For each byte of the microcontroller memory, we additionally introduce a sequence of 8 elements of 0/1 intervals in order to keep track of bitwise operations. In the following, denote the domain of single bitwise intervals by  $\mathcal{I}_2 = \{\emptyset, [0, 0], [0, 1], [1, 1]\}$ . Each memory location consisting of 8 bits is then represented by the cartesian product of 8 elements of  $\mathcal{I}_2$ , numbered from 0 through to 7. This domain is denoted by  $\mathcal{I}_{8 \times 2}$ . Naturally,  $(\mathcal{I}_{8 \times 2}, \sqcup)$  with the component-wise join of intervals is a complete lattice.

For a formal definition of a Galois connection for relating concrete and abstract representations, let  $\pi_i : \mathbb{N}_{<256} \rightarrow \mathbb{N}_{<2}$  be a projection of an integer onto the  $i$ -th bit in a bit-wise representation. Moreover, let  $\nu_i : 2^{\mathbb{N}_{<256}} \rightarrow \mathcal{I}_2$  be an auxiliary function that computes minimal and maximal values for the  $i$ -th bit of a set of integers. A Galois connection  $(\mathbb{N}_{<256}, \alpha_{8 \times 2}, \gamma_{8 \times 2}, \mathcal{I}_{8 \times 2})$  between  $\mathbb{N}_{<256}$  and  $\mathcal{I}_{8 \times 2}$  with  $N \subseteq \mathbb{N}_{<256}$  and  $[l_i, u_i]_{0 \leq i \leq 7} \in \mathcal{I}_{8 \times 2}$  can be defined as follows:

$$\begin{aligned} \nu_i(N) &= [\min\{\pi_i(n) \mid n \in N\}, \\ &\quad \max\{\pi_i(n) \mid n \in N\}] \\ \alpha_{8 \times 2}(N) &= \nu_0(N) \times \dots \times \nu_7(N) \\ \gamma_{8 \times 2}([l_i, u_i]_{0 \leq i \leq 7}) &= \{\sum_{i=0}^7 2^i \cdot x_i \mid x_i \in [l_i, u_i]\} \end{aligned}$$

### 3.3 Combining Word- and Bit-Level Intervals

These representations give rise to the reduced product domain for combining word-level intervals  $\mathcal{I}_{256}$  and bit-level intervals  $\mathcal{I}_{8 \times 2}$  with  $\mathbb{N}_{<256}$ . That is,  $(\mathbb{N}_{<256}, \alpha, \gamma, \psi(\mathcal{I}_{256} \times \mathcal{I}_{8 \times 2}))$  according to the definition given in Sect. 2 forms a Galois connection. In this combined domain, concrete values are obtained from the abstract representations if and only if they are contained in both abstract domains. This property is ensured by construction and is a key for the gain in precision.

In the following, let  $\mathcal{P}$  denote the set of all program locations in the program under scrutiny, that is,  $\mathcal{P}$  corresponds to the set of instructions. Each program location has incoming intervals, which are transformed using transfer functions to yield output intervals (often called *entry-* and *exit-*sets). Therefore, the concrete state space of the program with respect to its collecting semantics is defined as  $\mathcal{S} = \mathcal{P} \times (2^{\mathbb{N}_{<256}})^{1120} \times \text{Dir}$ , where  $\text{Dir} = \{\text{in}, \text{out}\}$ . Accordingly, the abstract state space of the program is defined as  $\mathcal{S}^\# = \mathcal{P} \times (\mathcal{I}_{256} \times \mathcal{I}_{8 \times 2})^{1120} \times \text{Dir}$ .

## 4. INSTRUCTION SET

This section details the abstract interpretation of the program under scrutiny (hardware dependencies are ignored for the time being). First, it describes the semantics for some instructions of the ATmega16. Afterwards, it explains how loop conditions are inferred in order to narrow down the obtained intervals. Finally, this section details the integration of interrupt handlers for the analysis of interrupt-driven programs.

Abstract interpretation of the instruction set forms the basis for Sect. 5, which explains how a model of the hardware is integrated into this framework. It turns out that hardware models integrate smoothly with this approach.

### 4.1 Overview

Each instruction (out of 131) has a well-defined semantics that is given in the instruction-set specification [1]. All instructions operating on general-purpose registers read at most two memory locations and write at most one general-purpose register. Moreover, the status register – which contains the carry, half-carry, interrupt, negative, overflow, sign, transfer, and zero flag – is altered depending on the result by some of these instructions. For instance, an arithmetic instruction such as `ADD r0 r1` writes the sum of `r0` and `r1` into `r0` and sets the negative flag (among others) if the most significant bit of the result is set.

SRAM locations of the microcontroller can only be ac-

cessed by instructions that copy words from SRAM into general-purpose registers or vice versa. The ATmega16 also supports indirect read and indirect store operations using one of three pointer registers `X` (`r26` and `r27`), `Y` (`r28` and `r29`) or `Z` (`r30` and `r31`), which are combined for 16-bit addressing. This means, for example, that the address indicated by `X` is computed as `r26 + 256 · r27`. The runtime stack is utilized using the instructions `PUSH` and `POP`, which perform indirect reads/stores using the stack pointer (`SPL` and `SPH`), which is part of the I/O registers.

The control flow of programs is changed by certain conditional branching instructions `BRxx k`. These instructions increment the program counter by `k+1` or 1 if the corresponding branching condition evaluates to true or false, respectively. Loops in programs are implemented using conditional branching instructions or unconditional jumps with a target address smaller than the current program counter.

### 4.2 Semantics

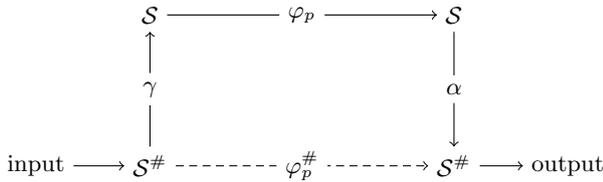
Consider an instruction `ADD r0 r1` at a program location  $p \in \mathcal{P}$ . The instruction transforms a concrete input state  $s = (p, d, \text{in}) \in \mathcal{S}$  into an output state  $s' = (p, d', \text{out}) \in \mathcal{S}$ , where only the values of register `r0` and certain bits of the status register `SREG` in  $d$  are mutated to yield  $d'$ . Overall, the instruction has the following concrete semantics, where the output variables are primed,  $m_k$  denotes the  $k$ -th bit of a memory cell  $m$ , and symbolic names for the status bits are used:

$$\begin{aligned} r0' &\leftarrow r0 + r1 \\ SREG'_{\text{zero}} &\leftarrow \bigwedge_{i=0}^7 \neg r0'_i \\ SREG'_{\text{carry}} &\leftarrow r0_7 \wedge r1_7 \vee r1_7 \wedge \neg r0_7 \vee \neg r0_7 \wedge r0_7 \\ SREG'_{\text{negative}} &\leftarrow r0_7 \\ SREG'_{\text{overflow}} &\leftarrow r0_7 \wedge r1_7 \wedge \neg r0_7 \vee \neg r0_7 \wedge r1_7 \wedge r0_7 \\ SREG'_{\text{sign}} &\leftarrow SREG'_{\text{neg}} \oplus SREG'_{\text{overflow}} \\ SREG'_{\text{half-carry}} &\leftarrow r0_3 \wedge r1_3 \vee r1_3 \wedge \neg r0_3 \vee \neg r0_3 \wedge r0_3 \\ SREG'_{\text{transfer}} &\leftarrow SREG_{\text{transfer}} \\ SREG'_{\text{interrupt}} &\leftarrow SREG_{\text{interrupt}} \end{aligned}$$

All other values remain unchanged by this instruction. Note how this approach takes care of overflows by performing the addition in the concrete semantics. For instance, adding the intervals  $[0, 1]$  and  $[254, 255]$  yields  $\top$ . For indirect store instructions, however, the target of the instruction is determined through the contents of the corresponding pointer register. In the concrete case, such instructions perform strong updates. Due to the over-approximation, indirect stores *may* overwrite a memory location but are not guaranteed to. Consequently, the original results of the register are joined with the new contents but not replaced. As an example, consider an instruction such as `STPI X r0`, which stores the value of `r0` in the memory cell indicated by register `X` and then post-increments `X`. The address indicated by `X` is denoted by  $\llbracket X \rrbracket$ , leading to the following concrete transfer function:

$$\begin{aligned} \llbracket X \rrbracket' &\leftarrow \llbracket X \rrbracket \cup r0 \\ X' &\leftarrow X + 1 \end{aligned}$$

The semantics for indirect reads is defined accordingly. Note that this modelling is also applied to instructions affecting the runtime stack such as `PUSH`, `POP`, `CALL`, and `RET`. This way, concrete transfer functions for the complete instruction set can be specified, which are denoted by  $\varphi_p : \mathcal{S} \rightarrow \mathcal{S}$  for each  $p \in \mathcal{P}$ . Naturally, program analysis by means of



**Figure 2: Abstract interpretation for instruction-set semantics**

abstract interpretation is then performed using the sequential composition of concretization and the concrete transfer function followed by abstraction. This yields an abstract transfer function  $\varphi_p^\# : \mathcal{S}^\# \rightarrow \mathcal{S}^\#$  with:

$$\begin{aligned} \text{output}(p, d^\#, \text{dir}) &= (p, d^\#, \text{out}) \\ \varphi_p^\#(p, d^\#, \text{in}) &= \text{output}((\alpha \circ \varphi_p \circ \gamma)(p, d^\#, \text{in})) \end{aligned}$$

This situation is depicted in Fig. 2. The sole purpose of function `output` is the renaming of states, from entry-sets to exit-sets. Given the control flow graph  $(\mathcal{P}, E)$  of a program, an abstract transition relation that joins the output states of all  $p \in \mathcal{P}$  in order to yield the input state of  $p' \in \mathcal{P}$  iff  $(p, p') \in E$  can easily be generated. The solution to this system is then computed using fixed point iteration.

### 4.3 Loop Conditions

In high-level programs, conditionals are easily obtained from the program’s abstract syntax, and then used to constrain values of variables on certain execution paths. In binary code, extracting the conditionals is more difficult because these are implemented using a sequence of instructions and a branching instruction that depends on certain status flags. The meaning of the status flags and their combinations depends on instructions executed before: The flags alone do not resemble any semantic information. Additionally, interrupt handlers may interfere with this sequence.

The instruction-set specification of the ATmega16 suggests to implement conditional branching using so-called compare instructions: `CPI` (compare to immediate), `CP` (compare to register), and `CPC` (compare to register with carry). These are followed by a branching instruction. The first operand of a compare instruction is a register  $r$ , and the second operand is either a register  $p$  or a constant value  $k$ . The instructions then compute  $r-p$ ,  $r-k$ , or  $r-p\text{-carry}$ , respectively, but the result is not stored. Only status flags are mutated according to the result. The branching instruction then evaluates the status flags. A statement such as `if (x == 0)`, where  $x$  is a 16-bit integer (stored in registers  $r24$  and  $r25$ ), for example, is translated into a sequence such as `LDI r0 0; CPI r24 0; CPC r25 r0; BREQ 3`.

In order to take value range restrictions stemming from such conditionals into account, pattern matching is performed to recognize the most frequently found branching sequences. Then, constraints are added, depending on the pattern found. For the above equality condition, for instance, value range restrictions  $r24 \sqcap [0, 0]$  and  $r25 \sqcap [0, 0]$  are added to the outgoing *then*-edge. This approach requires that interrupts are turned off in the sequence. Otherwise, it is possible that the first compare instruction is executed, the value of the corresponding register is then mutated by an interrupt handler, which invalidates the constraints.

Hand-written assembly code and certain compiler-generated code, however, often do not implement conditional branching using these patterns. Consequently, a computationally more expensive approach is required to handle such cases and recover semantic information. Our method involves statically identifying all possible paths that influence the status flags and simulating each of these paths in the concrete semantics where it is required in order to determine which branch is taken. This approach is explained in the following using an example.

Consider a C code fragment such as  $x = \sim(1 \ll c)$  where  $c$  and  $x$  are 8-bit variables. While this appears to be a sequence of two primitive expressions in C, it is translated into a loop in binary code because the AVR instruction set only supports shifting by a constant value (see Fig. 3). This code essentially initializes  $x$  (`r24`) and doubles its value  $c$  times in a loop. Here,  $c$  is stored in register `r18`. The loop terminates once the highest bit of  $c$  is set in instruction `0x99`, that is, decrementing `r18` yields a negative number in two’s complement (the negative flag is cleared). Afterwards, the `COM` instruction is executed to compute the bitwise complement.

In the example in Fig. 3, there exists a single relevant path for the branching instruction, consisting only of the instructions at `0x98` and `0x99`. Suppose that the interval analyzer has inferred that  $r18 \in [1, 1]$  before `DEC r18`. The abstract interpreter then reaches `BRPL -3` for the first time: It evaluates the path with the concrete input value 1 for `r18` and adds 0 to the output set for the case that the negative flag is cleared. No value causes the negative flag to be set. Then, when the abstract interpreter visits the instruction at `0x99` again, the interval analyzer will have computed  $r18 \in [0, 1]$  before `DEC r18`. Now, the concrete assignment  $r18 = 0$  before `DEC r18` causes the negative flag to be set in `BRPL -3`, and the resulting value  $r18 = -1$  is added to the *else*-successor of the `BRPL -3` instruction.

In general, paths of interest are determined as follows: Starting from the branching instruction, the definition-use chain for the negative flag is evaluated to yield those instructions that write its value. If the outcome of the previous instructions depends on the value of a status flag itself, the definition-use chain is evaluated in order to find fresh relevant predecessors. The evaluation of definition-use chains is repeated until one of the following termination criteria holds: (1) The value computed by the respective instruction can be determined from a set of register values because the interval analyzer has already produced an approximation of these values, or (2) a cycle in the relevant path is detected. Condition (2) guarantees termination.

This way, a set of paths that need to be evaluated is discovered. Each such path  $\pi = p_0 \dots p_n$ , where  $p_n$  represents a branching instruction, is analyzed in the concrete program semantics with the possible input assignments. Concretization of values is performed on-the-fly, triggered by the instruction semantics. After  $p_n$  has been analyzed, it is evaluated whether the branching condition is satisfied or not. Depending on the outcome, the computed concrete values are either added to the *then*- or to the *else*-path and abstracted afterwards.

Similar ideas, namely performing a concrete execution of certain paths to recover precision lost in the symbolic/abstract domain can also be found in the field of concolic testing [3]. A technically more involved approach to handling this problem would be to encode identified paths in a

```

0x94: LDI r24 1    ; x <- 1
0x96: RJMP 1      ; jump to 0x99
0x97: ADD r24 r24 ; x <- x*2
0x98: DEC r18     ; c <- c-1
0x99: BRPL -3    ; branch if positive
0x9a: COM r24     ; x <- ~x

```

**Figure 3: C statement  $x = \sim(1 \ll c)$  compiled into AVR assembly**

bit-vector theory and either use CLP- or SAT-based solvers for computing range information as performed by Bardin et al. [4].

## 4.4 Interrupts

Computing the least fixed point for single threaded programs is straightforward using standard techniques based on the control flow graph. However, microcontroller programs are typically interrupt-driven. This means that whenever interrupts are enabled, an interrupt may fire and the corresponding interrupt handler is executed, which most likely has an effect on the execution of the main process. It is important to note that the firing of a nondeterministic interrupt is optional because it depends on the environment. In the following, we briefly explain our strategy for computing least fixed points in the presence of (pseudo-) concurrency introduced by interrupt handlers:

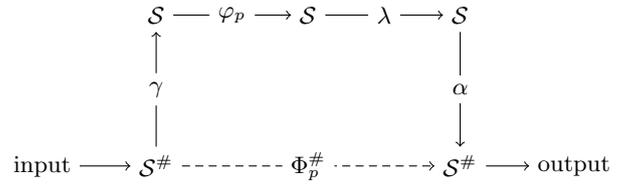
1. In a first step, the least fixed point of the main process is computed. This gives a first approximation of the global interrupt flag, which is stored in the highest bit of the status register.
2. For each instruction  $p$  of the main process where the global interrupt flag may be set, all interrupt handlers are executed using the context of  $p$ , and their results are joined with analysis results in  $p$  until a fixed point is reached.

Note that the execution of an interrupt handler may influence the main process. Consequently, the first approximation of the global interrupt flag is not necessarily an over-approximation of the actual values, and the execution of interrupt handlers may lead to larger analysis results for the main process. During startup of the microcontroller, however, interrupts are disabled and have to be enabled by the main process before an interrupt can fire. Thus, the second step is repeated using a worklist algorithm until a global fixed point of the main process is reached.

Moreover, performing a context-sensitive analysis of interrupt handlers is particularly important in order to suppress the propagation of information stemming from one instruction into another instruction through an interrupt handler. In our approach, each interrupt handler is analyzed from the context of the corresponding instruction only. This implies that detecting a fixed point of the main program suffices for termination detection.

## 5. HARDWARE MODEL

The technique described so far ignores hardware dependencies such as those of the I/O ports described in Sect. 1. In order to take care of such dependencies, we introduce an



**Figure 4: Abstract interpretation including environment transformer**

additional *environment transformer*  $\lambda : \mathcal{S} \rightarrow \mathcal{S}$ . Hardware-dependent behavior is only possible for certain I/O registers. For example, the pin register PINA depends on the values of the corresponding data-direction register DDRA and the port register PORTA. The general-purpose registers as well as the SRAM behave deterministically.

An environment transformer extracts the state of influencing registers, which are I/O registers as well, and produces a corresponding output state. Let  $\varphi_p : \mathcal{S} \rightarrow \mathcal{S}$  denote the concrete transfer function of some program location  $p \in \mathcal{P}$ , converting an input into some output (cf. Sect. 4). Then,  $\lambda \circ \varphi_p : \mathcal{S} \rightarrow \mathcal{S}$  defines a concrete transfer function including hardware behavior, which yields an abstract transfer function  $\Phi_p^\# : \mathcal{S}^\# \rightarrow \mathcal{S}^\#$  (cf. Fig. 4) as follows:

$$\Phi_p^\#(p, d^\#, \text{in}) = \text{output}((\alpha \circ \lambda \circ \varphi_p \circ \gamma)(p, d^\#, \text{in}))$$

The advantage of this method is that it provides a separation of concerns: The hardware model is separated from the semantics of instructions. As an example, consider the I/O port A and its corresponding I/O registers DDRA, PINA, and PORTA, which behave as explained in the introduction. The environment transformer models the behavior of the  $i$ -th bit of the PINA register as follows:

$$\text{PINA}'_i \leftarrow \begin{cases} \text{PORTA}'_i & : \text{DDRA}'_i = 1 \\ \{0, 1\} & : \text{otherwise} \end{cases}$$

Note that this approach allows modeling the hardware at an arbitrary level of detail, where mapping the input to  $\top$  generates a safe over-approximation. In our model, for instance, we have abstracted from timers as the analysis itself is not cycle-accurate. On the other hand, we have modeled I/O ports in all their details, as well as strictly deterministic I/O registers such as the stack pointer or the status register.

## 6. WIDENING

As a motivation for applying a widening operator during static analysis, consider the program given in Fig. 5. The program loads bytes from program memory into  $\text{r0}$  using the  $\text{Z}$  pointer register and then copies  $\text{r0}$  into the address indicated by the  $\text{X}$  register. Effectively, the program copies two bytes starting from program memory address 100 into the SRAM addresses 96 and 97. The CFG of the program fragment is given in Fig. 6.

The interval analyzer infers that  $\text{X} \in [96, 97]$  and  $\text{Z} = \top$  hold before instruction  $0x49$  is executed, which is a well-known drawback of non-relational analyses. While stabilization is ensured due to the finite ascending chain condition that holds for our interval domains, many iterations are required until the result  $\text{Z} = \top$  is eventually reached. While this typically is no problem on 8-bit microcontrollers due to

```

0x39: EOR r0 r0      0x45: CPC r27 r0
0x40: LDI r26 96    0x46: BRNE 1
0x41: EOR r27 r27   0x47: RET
0x42: LDI r30 100   0x48: LPMPI Z r1
0x43: EOR r31 r31   0x49: STPI X r1
0x44: CPI r26 98    0x4a: RJMP -7

```

Figure 5: Copy two bytes from program memory into SRAM

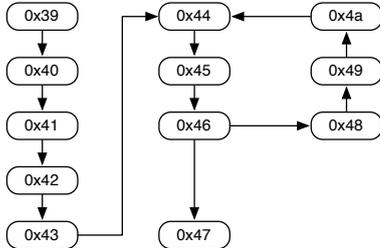


Figure 6: CFG of the program in Fig. 5

small domains, it can easily lead to a performance hit on 16- or 32-bit platforms. In this section, we propose a widening method that performs a-priori widening of  $Z$  in order to accelerate stabilization.

A *program dependence graph* (PDG) is a directed graph with vertices corresponding to instructions and control predicates (conditional branches), and edges corresponding to data and control dependences. Often, PDGs are used in static slicing [16], where all vertices reachable from some predefined *slicing criterion* (the vertex under consideration) are considered the slice. Therefore, PDG representations of programs are available in most static analyzers already.

We propose to perform widening based on an *annotated data dependence graph* (ADDG). An ADDG is essentially a PDG consisting only of data dependencies, and additionally vertices are annotated with monotonicity information about the corresponding instructions. The key idea of this representation is to determine registers that will result in the complete interval  $\top$  a-priori. Monotonicity information for each written memory location is represented by an annotation  $m \in \mathcal{M} = \{+\mathcal{M}, -\mathcal{M}, \star\mathcal{M}\}$ , where  $+\mathcal{M}$  represents monotone increasing instructions and  $-\mathcal{M}$  represents decreasing operations. Unknown or non-monotonic modifications are denoted by  $\star\mathcal{M}$ . Here, it is important to note that the analysis domain is unsigned. Each vertex – that is, each instruction – is annotated with monotonicity information about the altered memory locations and the kind of monotonicity, leading to annotations that are pairs  $(v, m)$  for variables  $v$  and  $m \in \mathcal{M}$ . For instance, vertices representing `INC r0` or `STPI X r0` are annotated with  $(r0, +\mathcal{M})$ , while a vertex representing `EOR r0 r1` is annotated with  $(r0, \star\mathcal{M})$ . Moreover, restrictions on intervals emerging from branching conditions (cf. Sect. 4.3) induce a non-monotonic data dependence. Consequently, the respective vertex is annotated with  $\star\mathcal{M}$ .

Our widening techniques identifies cycles in the data dependence edges of the ADDG, where all modifications of  $v$  are annotated with the same monotonicity information  $m$ . These cycles correspond to an unbounded monotone modifi-

Table 1: Performance of interval analysis

Program	LoC	# IHs	Runtime	Dead code
light_switch	162	0	0.52s	×
plant	243	2	1.24s	✓
traffic_light	218	0	2.41s	×
window_lift	225	3	1.49s	✓

cation of  $v$  in the non-relational domain, which yields  $v = \top$  eventually. In case that such a cycle exists, the initial configuration of  $v$  is set to  $\top$  before the interval analyzer is executed. Of course, this is a sufficient, but not a necessary condition. The detection of monotone cycles can be implemented as a straightforward modification of Tarjan’s algorithm for computing strongly connected components [28].

Note, however, that the ADDG is an under-approximation of the possible dependencies due to indirect stores and reads. Since indirect stores/reads are modeled as *weak* updates, such instructions cannot break a monotone cycle.

The ADDG for the program in Fig. 5 is depicted in Fig. 7 with the relevant edges and vertices highlighted. Here, the status flags are omitted for clarity. The monotone self-loop for  $Z$  in instruction `0x48` in the ADDG is highlighted, and consequently,  $Z$  is initially widened to  $\top$ . In contrast,  $X$  is not widened because the branching condition induces a data dependency with unknown monotonicity, therefore breaking the monotone cycle, which is emphasized using dashed lines.

## 7. CASE STUDY

This section evaluates the presented analysis with respect to performance, precision, and the effects on state-space reduction methods in model checking. The case studies were conducted on an IBM ThinkPad T60p, equipped with a 2.33 GHz dual-core processor and 4 GB of RAM.

### 7.1 Performance

The runtime for all our standard benchmarks (see Tab. 1) was less than 2.5 seconds. Moreover, it can be observed that the presence of interrupt handlers does not noticeably influence the performance of the analysis. It turns out that most of the runtime is spent copying and comparing states for fixed point detection because complete states, consisting of 1120 abstract elements, have to be compared in many cases.

Our original JAVA implementation of the bitwise analysis required a long runtime. Two `char` variables were used to store lower and upper bounds for a single bit. This runtime could be reduced by a factor of 5 by storing lower and upper bounds for an element of  $\mathcal{I}_{8 \times 2}$  in two `char` variables, allowing join, meet, and inclusion to be computed using computationally cheap bitwise operations.

### 7.2 Precision

Previously, Schlich [26] has detailed a case study of the program `window_lift`. The program implements a state machine that controls the operation mode of a window lift, which is stored in an unsigned 8-bit variable `mode`. Overall, the state machine has seven different modes and by its specification, the value of `mode` must always be less than or equal to 6. The value of `mode` is altered by the main process and in each of three interrupt handlers of the pro-

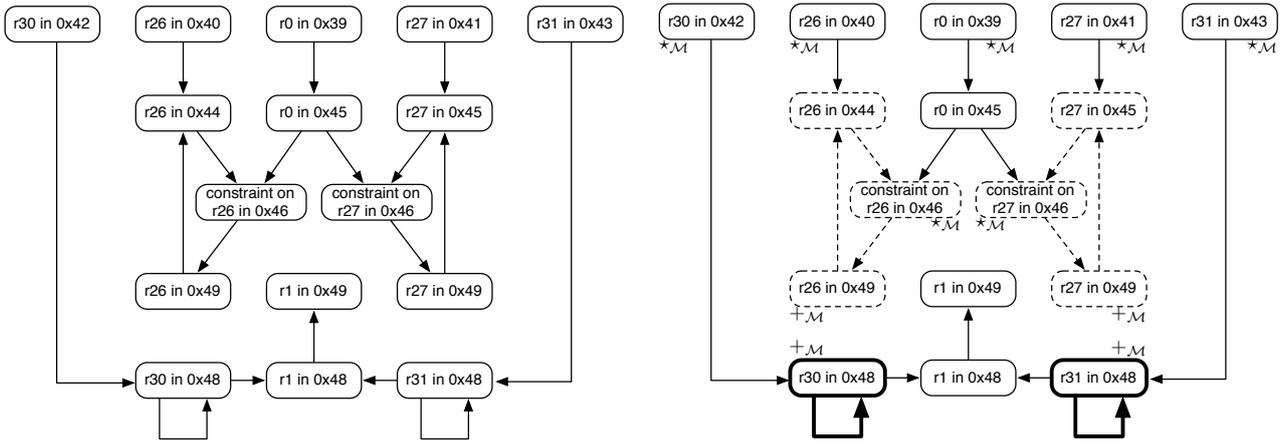


Figure 7: Data dependences in PDG (left) and ADDG (right) of the program given in Fig. 5

gram. Verifying this program without any abstraction took 62 seconds. While the runtime could be reduced to approximately 6 seconds using different static and dynamic abstraction techniques, the development and implementation of these techniques took several years. Despite these efforts, model checking using [MC]SQUARE is still very susceptible to state explosion, which is not the case for our abstract interpreter.

This invariant is formally specified in CTL as  $AG (0 \leq \text{mode} \wedge \text{mode} \leq 6)$ . In the compiled binary, `mode` is placed at location `0x0061` in the SRAM of the microcontroller. The combined interval analysis infers that  $\text{mode} \in [0, 6]$  in any program location, and hence, the same invariant is also verified through static analysis. This is not possible if only either word- or bit-level intervals are used. Word-level intervals infer that  $\text{mode} \in [0, 255]$  because bitwise loop conditions are not handled properly. On the other hand, using bitwise intervals only infers that  $\text{mode} \in [0, 7]$ . Other temporal properties verified using model checking in [26] require combinations of temporal and path operators, and thus, cannot be proven using our method. As byproducts, the interval analyzer infers that only global variables are accessed using indirect writes and that the maximum stack size is 7. These results are equal to those that can be observed using model checking.

The program `light_switch` contains two global variables that store the mode of the light switch and the current brightness. The interval analyzer was able to infer that both variables remain within range and that indirect stores/reads only access these global variables. Finally, we have validated the safety of our implementation by comparing the results produced by the interval analyzer to state spaces built using [MC]SQUARE.

### 7.3 Optimization

During our case studies, we observed that the compiler inserts code for fallback branches into code generated from `switch-case` statements. The AVR-GCC compiler attempts to implement `switch-case` statements by means of nested branches instead of indirect jumps using a jump table location in the program memory. In many cases, our approach was able to infer that these branches are dead code because

the branching condition for the fallback-branch is never satisfied. Moreover, during startup, the compiler implements a fixed pattern comparable to the loop given in Fig. 5 for initializing global variables using values from the program memory. This code is also contained in the compiled binary if the program does not contain any global variables and the loop is never entered. The programs for which dead code could be detected using interval analysis are highlighted in Tab. 1.

These results suggest that our approach could also be used for compiler optimization, which is particularly important in embedded systems software, where software is executed under tight performance, memory, and energy constraints. Such an application of assembly code analysis to code optimization has recently been described by Yang et al. [29].

## 7.4 Application to State-Space Reductions

We have integrated interval analysis into [MC]SQUARE, which is an explicit-state model checker for several microcontroller platforms. In [MC]SQUARE, several abstraction techniques are used to reduce state spaces during model checking. This includes adaptations of well-known techniques such as dead variable reduction (DVR) and path reduction (PR) [30]. In the following, we briefly describe how the applicability of DVR and PR for binary code verification is affected by our approach.

### 7.4.1 Dead Variable Reduction

The idea of DVR is to reset variables during state-space generation that are not *live*, that is, variables whose value is not going to be read in the future before it is overwritten. On binary-code level, this technique suffers from indirect reads in case that target addresses of such reads are unknown: The analysis has to assume that any memory location could be accessed, and therefore, all memory become live. This reduces the effectiveness of DVR.

In Sect. 7.2, we have described and evaluated the precision of our analysis with respect to the pointer registers. By integrating these results into live variables analysis, a reduced number of live memory locations is obtained and DVR yields better results. The improvement can be in orders of magnitude, depending on the program. For a program called `vector`, consisting of 930 instructions, combin-

ing the techniques reduced the state space from more than 4,000,000,000 states to 917,458 states. Many memory locations could be reset early, which caused numerous different states to be merged, eliminating a source for the exponential growth of state spaces.

### 7.4.2 Path Reduction

The application of PR causes state-space reductions by collapsing single-successor chains in the state space into single nodes. Boundary nodes of such single-successor chains are called *breaking points* and determined using a static analysis. Here, we name only two conditions for breaking points that are affected by interval analysis: Reading nondeterministic input and writing atomic propositions leads to breaking points. However, as with DVR, this technique also suffers from indirect writes and the lack of a precise hardware model.

Without knowledge about pointer registers, it has to be assumed that any program location could be accessed by an indirect store operation, and therefore, every indirect write is marked as a breaking point in order to generate an over-approximation. To name only two situations, without interval analysis every call instruction is a breaking point because a return value is indirectly stored on the stack, and there are several breaking points in each interrupt handler that saves the context on the stack.

Furthermore, whenever an I/O register is read, the value read is assumed to be nondeterministic, which also generates an over-approximation: Recall our explanation on I/O ports and the dependency between data direction registers and port registers. The integration of interval analysis results and the hardware model with PR leads to fewer breaking points, and thus, to smaller state spaces.

## 8. RELATED WORK

Interval analysis has long been used in program analysis and abstract interpretation [8], particularly in the context of high-level programming languages. To name only one example, the approach of Gawlitza et al. [13] is used in GOANNA to prune out false paths during static analysis [12].

Probably the most prominent platform assembly and binary code analysis is CODESURFER/x86 as described by Balakrishnan et al. [2, 25]. In their approach, intervals are used in combination with other domains such as congruences for the analysis of x86 executables. However, their approach is based on a symbolic memory representation and neglects soundness, which is reasonable for bug-finding frameworks but infeasible in the context of verification of safety-critical systems. Another low-level pointer analysis for x86/assembly code was described by Debray et al. [11], where addresses are represented as congruence values, but they only keep track of low-order bits of registers and all information is lost whenever addresses are written to the main memory. Guo et al. [14] perform a flow-sensitive pointer analysis only for registers, but memory locations are treated in a flow-insensitive manner.

Numerous special-purpose analyses for microcontroller assembly code have been developed. For example, stack size analysis for AVR microcontrollers was described by Regehr et al. [24]. The interval analysis described in this paper produces exactly the same results as a byproduct. Their analysis relies on the assumption that indirect stores access only the SRAM and do not alter I/O registers such as the

stack pointer registers SPL and SPH or the status register. This assumption can be proven to be valid by our approach. To the best of our knowledge, our approach is the first that proves the *safety* of indirect stores on microcontroller platforms using static analysis since it guarantees that no I/O registers are accessed. Several other techniques, such as the slicing approach for x86 binary code described by Cifuentes and Fraboulet [6], ignore the effects of indirect writes, rendering them unsound, where the integration of our method could be used to provide information about pointer values.

The widening operator [9] described in Sect. 4 can be seen as an adaptation of the widening approach used by Gawlitza et al. [13]. In their method, interval analysis is performed using an extension of the Bellman-Ford algorithm, where cycles of negative weight are identified in order to perform a widening of the interval boundaries. In contrast, our widening approach explicitly identifies monotonely ascending cycles based on an annotated version of the PDG.

Not strictly related to our approach, where static cycle detection is performed in order to speed-up convergence of the analysis, (dynamic) cycle detection is also employed in other fields such as flow-insensitive pointer analysis in order to guarantee convergence [23]. While flow-insensitive pointer analyses have proven to be useful for high-level programs [15], such techniques do not provide meaningful results for AVR microcontrollers because all indirect stores and reads are performed using one out of three pointer registers: The result will most likely be a single equivalence class of pointer values.

## 9. CONCLUSION

### Summary.

In this paper, we have described how word- and bit-level interval analysis can be combined in order to obtain precise results for binary code that uses both arithmetic and logical operations. Furthermore, we have detailed an analysis of branching conditions and how hardware dependencies can be integrated into the abstract interpretation framework. In order to accelerate convergence, we have sketched an offline widening approach based on PDGs.

The effectiveness and performance of the approach was evaluated in a case study. Moreover, we have described the integration of interval analysis with state-space reduction methods.

### Future Work.

One aspect of future work is the support for other microcontrollers of the AVR-family, which can be achieved by adjusting the hardware model and the state space. Supporting the ATMEL ATmega128, for instance, would allow us to analyze programs written for TINYOS. From our experience with different microcontroller platforms, we do not expect any major difficulties porting the analysis to other target platforms.

Since the described analysis is non-relational, it fails to discover relationships between registers such as *r0 is incremented iff r1 is decremented*, which sometimes leads to unbounded values of variables altered monotonely in loops without being restricted by loop conditions. Such over-approximations are then propagated through the analysis. In the future, we want to integrate different relational anal-

yses in order to narrow down the results. This approach is also followed by several static analyzers for high-level languages such as F-SOFT [18] or ASTREE [10].

On the classical side, well-studied analyses for affine relationships [19] could serve as a basis for this work. As an example, consider that our interval analyzer infers that  $\mathbf{r0} \in [96, 98]$  and  $\mathbf{r1} \in [0, 255]$ , while an affine relationship  $\mathbf{r1} = 50 + \mathbf{r0}$  is discovered. Combining these results allows to derive that  $\mathbf{r1} \in [146, 148]$ . Another interesting domain are congruences, for which there has been a resurgence of interest as they capture the effects of overflows that are natural to machine arithmetic [21, 22]. Evaluating the suitability of other (weakly) relational domains such as interval polyhedra [5] and logahedra [17] for binary code analysis appears to be interesting as well.

## 10. ACKNOWLEDGEMENTS

This work has been supported partly by the UMIC Research Centre, RWTH Aachen University. We have been greatly helped by discussions with Andy King on abstract interpretation. Further, we thank our student Andreas Weigelt for his support on the implementation of the algorithms described in this paper. We also thank the anonymous referees for their helpful suggestions.

## 11. REFERENCES

- [1] Atmel Corp. *8-bit AVR Instruction Set*, July 2008.
- [2] G. Balakrishnan, T. W. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What you see is not what you execute. In *VSTTE 05*, volume 4171 of *LNCS*, pages 202–213. Springer, 2005.
- [3] S. Bardin and P. Herrmann. Structural testing of executables. In *ICST 08*, pages 240–249. IEEE, 2008.
- [4] S. Bardin, P. Herrmann, and F. Perroud. An alternative to SAT-based approaches for bit-vectors. In *TACAS 2010*, volume 6015 of *LNCS*. Springer, 2010.
- [5] L. Chen, A. Mine, J. Wang, and P. Cousot. Interval polyhedra: An abstract domain to infer interval linear relationships. In *SAS 2009*, volume 5673 of *LNCS*, pages 309–325. Springer, 2009.
- [6] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *ICSM 97*, pages 188–195. IEEE, 1997.
- [7] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. of the 2nd International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.
- [8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL 77*, pages 238–252. ACM, 1977.
- [9] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP 92*, volume 631 of *LNCS*, pages 269–295. Springer, 1992.
- [10] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. The Astrée analyser. In *ESOP 05*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.
- [11] S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *POPL 98*, pages 12–24. ACM, 1998.
- [12] A. Fehnker, R. Huuck, and S. Seefried. Incremental false path elimination for static software analysis. In *ATVA 09*, volume 5799 of *LNCS*, pages 255–270. Springer, 2009.
- [13] T. Gawlitzka, J. Leroux, J. Reineke, H. Seidl, G. Sutre, and R. Wilhelm. Polynomial precise interval analysis revisited. In *Efficient Algorithms*, volume 5760 of *LNCS*, pages 422–437. Springer, 2009.
- [14] B. Guo, M. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D. August. Practical and accurate low-level pointer analysis. In *CGO 05*, pages 291–302. IEEE, 2005.
- [15] M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *SAS 98*, *LNCS*, pages 57–81. Springer, 1998.
- [16] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, January 1990.
- [17] J. Howe and A. King. Logahedra: A new weakly relational domain. In *ATVA 09*, *LNCS*. Springer, 2009.
- [18] F. Ivancic, Z. Yang, M. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-Soft: Software verification platform. In *CAV 05*, volume 3576 of *LNCS*, pages 301–306. Springer, 2005.
- [19] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.
- [20] J. Kinder, H. Veith, and F. Zuleger. An abstract interpretation-based framework for control flow reconstruction from binaries. In *VMCAI 09*, volume 5403 of *LNCS*, pages 214–228. Springer, 2009.
- [21] A. King and H. Søndergaard. Automatic abstraction for congruences. In *VMCAI 10*, volume 5944 of *LNCS*, pages 197–213. Springer, 2010.
- [22] M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. *ACM Trans. Program. Lang. Syst.*, 29(5), August 2007.
- [23] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis of C. *ACM Trans. Program. Lang. Syst.*, 30(1), 2007.
- [24] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. In *EMSOFT 03*, pages 306–322, 2003.
- [25] T. W. Reps and G. Balakrishnan. Improved memory-access analysis for x86 executables. In *CC 08*, volume 4959 of *LNCS*, pages 16–35. Springer, 2008.
- [26] B. Schlich. *Model Checking of Software for Microcontrollers*. Dissertation, RWTH Aachen University, Aachen, Germany, June 2008.
- [27] B. Schlich, T. Noll, J. Brauer, and L. Brutschy. Reduction of interrupt handler executions for model checking embedded software. In *HVC 2009*, *LNCS*. Springer, 2009. To appear.
- [28] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
- [29] X. Yang, N. Coopriider, and J. Regehr. Eliminating the call stack to save ram. In *LCTES 09*. ACM Press, 2009. To appear.
- [30] K. Yorav and O. Grumberg. Static analysis for state-space reductions preserving temporal logics. *Formal Methods in System Design*, 25(1):67–96, 2004.