Static Analysis of Run-Time Errors in Interrupt-Driven Programs via Sequentialization

XUEGUANG WU, National University of Defense Technology
LIQIAN CHEN, National University of Defense Technology
ANTOINE MINÉ, Université Pierre et Marie Curie
WEI DONG, National University of Defense Technology
JI WANG, National University of Defense Technology

Embedded software often involves intensive numerical computations and suffers from a number of run-time errors. The technique of numerical static analysis is of practical importance for checking the correctness of embedded software. However, most of the existing approaches of numerical static analysis consider sequential programs, while interrupts are a commonly used facility that introduces concurrency in embedded systems. Therefore, a numerical static analysis approach is highly desired for embedded software with interrupts. In this paper, we propose a static analysis approach specifically for interrupt-driven programs based on sequentialization techniques. We present a method to sequentialize interrupt-driven programs into non-deterministic sequential programs according to the semantics of interrupts. The key benefit of using sequentialization is the ability to leverage the power of the state-of-the-art analysis and verification techniques for sequential programs to analyze interrupt-driven programs, for example, the power of numerical abstract interpretation to analyze numerical properties of the sequentialized programs. Furthermore, to improve the analysis precision and scalability, we design specific abstract domains to analyze sequentialized interrupt-driven programs by considering their specific features. Finally, we present encouraging experimental results obtained by our prototype implementation.

CCS Concepts:•**Software and its engineering** → **Software defect analysis;**

General Terms: Analysis, Algorithms

Additional Key Words and Phrases: Embedded Software, Interrupt-Driven Programs, Static Analysis, Run-Time Errors, Abstract Interpretation, Sequentialization

ACM Reference Format:
Xueguang Wu, Liqian Chen, Antoine Miné, Wei Dong and Ji Wang, 2016. Static Analysis of Run-Time Errors in Interrupt-Driven Programs via Sequentialization. *ACM Trans. Embedd. Comput. Syst.* 0, 0, Article 0 (2016), 27 pages.
DOI: 0000001.0000001

1. INTRODUCTION

An interrupt is a signal to the processor indicating an event that needs immediate attention and requiring the interruption of the current code the processor is executing. Interrupts are commonly used in embedded systems to introduce concurrency, which is required for real-time applications. For example, embedded control software often uses interrupts to obtain sensor data from the physical environment. In a program, during the running of the normal tasks, an interrupt service routine (ISR) is invoked

This work is supported by the 973 Program under Grant No. 2014CB340703, the NSFC under Grant Nos. 61120106006, 61532007, 91318301.

Author's addresses: X. Wu, L. Chen, W. Dong and J. Wang, State Key Laboratory of High Performance Computing, College of Computer Science, National University of Defense Technology, Changsha 410073, China; A. Miné, LIP6 Laboratory, Université Pierre et Marie Curie, Paris 75252, France. L. Chen and J. Wang are corresponding authors; email: {lqchen,wj}@nudt.edu.cn.

Permission to make digital or hard copies of part or all 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. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2016 Copyright held by the owner/author(s). 1539-9087/2016/-ART0 $15.00

DOI: 0000001.0000001

once an interrupt alerts the processor to a higher-priority condition. Such a program is said to be interrupt-driven. In interrupt-driven programs (IDP), interrupts may cause unexpected interleaving and even unexpected erroneous behaviors [Yang et al. 2015]. Therefore, there is a great need in practice to ensure that IDPs work correctly in the presence of interrupts, since IDPs are often used in safety critical fields such as avionics, spaceflight and automotive. However, analyzing and verifying IDPs are challenging. The main reason is that an ISR may be triggered at any time and the number of possible execution interleavings caused by concurrency between tasks and ISRs is quite huge.

IDPs often appear in embedded systems, while embedded software usually involves intensive numerical computations which have the potential to cause numerical runtime errors (such as division by zero, arithmetic overflow and array out-of-bound) [Blanchet et al. 2003]. Hence, analyzing numerical properties of IDPs is of significant importance to check for the correctness of embedded software. Numerical static analysis is a commonly used technique to discover numerical properties of programs. However, most of the existing numerical static analysis approaches consider only sequential programs. For IDPs, if we perform numerical static analysis over each task and each ISR separately without considering the interleaving between them, the analysis results may be not sound.

```
1: int x, y, z;
2: void task(){
3:   if(x < y){
4:     z = 1/(x − y);
5:   }
6:   return;
7: }
```

```
1: void ISR1(){
2:   x++;
3:   y−−;
4:   return;
5: }
```

Fig. 1. A motivating example

Fig. 1 shows a motivating example, where the functions $task()$ and $ISR1()$ represent the entry functions of a task and an interrupt service routine respectively. $task()$ performs the division operation only when x is strictly less than y. $ISR1()$ increases x by 1 and decreases y by 1. Performing numerical static analysis over $task()$ without considering interrupts would answer that the program is safe. However, when taking interrupts into consideration, the $task()$ function is not safe. For example, if $x = 1, y = 3$ and the interrupt is triggered between line 3 and 4 of $task()$, there will be a division-by-zero error in this program. Thereby, a sound numerical static analysis method is desired for IDPs.

Recently, a few numerical static analysis approaches have been proposed for general concurrent programs such as multi-threaded programs [Miné 2011; 2014], but very few approaches have considered the specific features of IDPs [Cooprider and Regehr 2006; Monniaux 2007]. Compared with multi-threaded programs, IDPs have their own specific features. For example, higher-priority interrupts will never be interrupted by lower ones. In other words, tasks and lower priority interrupts will never be aware of the intermediate states of higher priority interrupts during their running. Moreover, IDPs in embedded systems usually make use of hardware features such as interrupt mask registers (IMR) to control the interference between tasks and interrupts.

In this paper, we propose a sound numerical static analysis approach specifically for IDPs, which enables the use of existing analysis and verification techniques for sequential programs. The basic idea is to sequentialize IDPs by a semantic sound program transformation and then to analyze the resulted sequential programs by abstract interpretation enriched with newly designed abstract domains specific to the features of IDPs. The main contributions are as follows.

(1) We present a method to sequentialize IDPs into non-deterministic sequential programs according to the semantics of interrupts and the interaction between tasks and interrupts. The data flow dependency and IMR information are used to optimize the transformation and reduce the size of the resulted sequential programs.
(2) We design specific abstract domains to improve the precision of numerical static analysis with reasonable scalability. These abstract domains incorporate the specific features which often appear in the sequentialized interrupt-driven programs.
(3) We conduct a set of experiments to evaluate the approach on both benchmarks and real-world programs from open source communities as well as industrial communities. The preliminary results show that our approach is promising.

The rest of this paper is organized as follows. Section 2 presents the program syntax of IDPs. Section 3 presents methods for sequentializing IDPs. In Section 4, we show how to use abstract interpretation to analyze the sequentialized IDPs. Section 5 presents our implementation together with preliminary experimental results. Section 6 discusses some related work. Finally, conclusions as well as suggestions for future work are given in Section 7.

This paper is an extended version of our EMSOFT 2015 paper [Wu et al. 2015]. On top of [Wu et al. 2015], we have added more algorithms and examples to describe our approach in detail (Section 3). We have designed a new specific abstract domain namely syntactic equality abstract domain to improve the precision of the analysis of sequentialized IDPs (Section 4). We have collected more benchmarks and real-world IDPs from open source communities as well as aerospace industrial communities, and have conducted more experiments (Section 5).

2. INTERRUPT-DRIVEN PROGRAMS

An IDP consists of a fixed set of a finite number of tasks and interrupts, each of which has an entry function. In this paper, recursive functions are not allowed in IDPs, and all functions have been inlined (except the entry functions of tasks and interrupts). Tasks are scheduled in a cooperative, round-robin manner and interrupts are assigned priorities. Moreover, the tasks execute with interleaved semantics (on a uniprocessor) and each task can finish its job within the given time slice. Each interrupt has a fixed priority level attribute p which is a positive integer and a larger priority level means higher priority. It means that higher-priority interrupts can preempt lower-priority interrupts and tasks, but the opposite preemption can not happen. Without loss of generality, we use the following two assumptions throughout this paper:

1) We assume that an IDP only consists of a single task. Since tasks are scheduled in a cooperative, round-robin manner, for an IDP including multiple tasks, we can design a wrapper function which consists of calling each of the tasks one by one in sequence to simulate the round-robin scheduling of multi-tasks.
2) We assume that each priority level contains only one interrupt. For an IDP which contains multiple interrupts with the same priority, we can design a new wrapper ISR of that priority to over-approximate the program behaviors. The new wrapper ISR consists of a loop in which each iteration non-deterministically calls one of the original interrupts of that priority.

We now present a simple language to model IDPs. The syntax of our language is depicted in Fig. 2. An IDP consists of one task and N interrupts. $ISR := \langle entry, p \rangle$ represents the ISR entry function of an interrupt and its priority. Without loss of generality, we use ISR_i to represent the interrupt with priority $p = i$ where $i \in [1, N]$. $enableISR(i)$

$$
\begin{array}{lll}
\mathit{AExpr} & := & l \mid C \mid E_1 \diamond E_2 \text{ (where } l \in NV,\ C \text{ is a constant, } E_1, E_2 \in \mathit{AExpr} \\
 & & \text{and } \diamond \in \{+, -, \times, \div\}) \\
\mathit{BExpr} & := & \mathit{true} \mid \mathit{false} \mid AE_1 \diamond_1 AE_2 \mid BE_1 \diamond_2 BE_2 \text{ (} AE_1, AE_2 \in \mathit{AExpr}, \\
 & & BE_1, BE_2 \in \mathit{BExpr},\ \diamond_1 \in \{>, \geq, <, \leq\} \text{ and } \diamond_2 \in \{\&\&, ||\}) \\
\mathit{Stmt} & := & l = g \mid g = l \mid l = e_a \mid \textbf{skip} \mid \textbf{return} \mid \mathit{enableISR}(i) \mid \mathit{disableISR}(i) \mid S_1; S_2 \\
 & & \mid \textbf{if } e_b \textbf{ then } S_1 \textbf{ else } S_2 \mid \textbf{while } e_b \textbf{ do } S \text{ (where } l \in NV, g \in SV, \\
 & & e_a \in \mathit{AExpr}, e_b \in \mathit{BExpr}, i \in [1, N], S_1, S_2, S \in \mathit{Stmt}\ ) \\
\mathit{Task} & := & \mathit{entry} \text{ (where } \mathit{entry} \in \mathit{Stmt}) \\
\mathit{ISR} & := & \langle \mathit{entry}, p \rangle \text{ (where } \mathit{entry} \in \mathit{Stmt},\ p \in [1, N]) \\
\mathit{Prog} & := & \mathit{Task} \parallel \mathit{ISR}_1 \parallel \ldots \parallel \mathit{ISR}_N
\end{array}
$$

Fig. 2. Syntax of interrupt-driven programs

and $\mathit{disableISR}(i)$ represent the instructions that enable and disable the i-th interrupt respectively by writing to the interrupt mask register (IMR). We assume that at the exit-point of an ISR, the IMR is reset to the initial value of IMR at the entry-point of this ISR. We use SV and NV respectively represent the set of shared and non-shared variables. We restrict that the task has and only has one return statement.

For the sake of simplicity but without loss of generality, we restrict that shared variables can only appear in the following two kinds of statements:

- $l = g$ which represents reading the value from a shared variable g to a non-shared variable l,
- $g = l$ which represents writing the value of a non-shared variable l to a shared variable g.

In fact, any program statement involving shared variables can be transformed into these forms by introducing auxiliary variables. E.g., a test $\{if(g > e) \ldots\}$ can be transformed into the sequence $\{l = g; if(l > e) \ldots\}$. Moreover, we assume that the statements $g = l$ and $l = g$ are *atomic*.

In this paper, we consider analyzing IDPs at the level of source code rather than that of machine code. However, one program statement in the source code can be translated into several machine instructions. Hence, an interrupt may be triggered during the running of a program statement if the statement is not atomic. For example, consider an IDP that contains one task $\{z = x + y;\}$ and one interrupt whose ISR is $\{x = 1; y = 1;\}$. Suppose that both shared variables x, y are initialized to 0. If we consider only the case that the interrupt is triggered before or after the assignment statement $\{z = x + y;\}$ in the task, the value of variable z can be 0 or 2. However, the interrupt may be triggered during the running of this statement at machine instruction level. For example, the interrupt may be triggered after reading x and before reading y, and then the value of variable z can be 1 in this case. In order to avoid this ambiguity, we only allow two kinds of statements (i.e., $l = g$ and $g = l$) that are atomic to access a shared variable variable g in the syntax shown in Fig. 2.

3. SEQUENTIALIZING INTERRUPT-DRIVEN PROGRAMS

In this section, we will describe how to sequentialize IDPs into non-deterministic sequential programs in a sound way. In other words, we will guarantee that the program behaviors of the sequentialized program are an over-approximation of the behaviors of the original IDP. In the following, we will first show in Sect. 3.1 how to sequentialize IDPs into sequential programs by considering IDPs as priority preemptive scheduling systems. Then we will make use of data flow dependency information between tasks and interrupts to remove unnecessary scheduling in Sect. 3.2. After that, we will consider further the IMR information at program points to remove unnecessary scheduling in Sect. 3.3. In addition, we will consider the interrupts which have no data flow dependency with tasks in Sect. 3.4. On this basis, we will give the overall algorithm of our sequentialization approach in Sect. 3.5. Finally, we will discuss the soundness of our sequentialization approach in Sect. 3.6.

3.1. Sequentializing IDPs by simulating priority preemptive scheduling

First, let us review the running process of an IDP. During the running of a task, the i-th interrupt may be triggered before any program statement of the task. If the i-th interrupt is triggered, the task is preempted and resumes only when ISR_i has finished. Hence, this situation can be simulated by calling the ISR_i function in the stack once the i-th interrupt is triggered. Similarly, before each program statement of the i-th interrupt, if the j-th interrupt with higher priority (i.e., satisfying $i < j$) is triggered, ISR_i is preempted and resumes only when ISR_j has finished. This situation also can be simulated by calling the ISR_j function in the stack when the j-th interrupt is triggered. In general, because the ISR of a preempted lower-priority interrupt (or task) will not resume until the ISR of a higher-priority interrupt has finished, the task and ISRs in an IDP can share the same stack. It means that, in IDPs, interrupt preemption can be modeled as merely a function call.

Based on this insight, inspired from [Kidd et al. 2010] where a $Schedule()$ function is used by Kidd et al. to simulate the priority preemptive scheduler for sequentializing multi-tasking programs, we add an explicit call to the $Schedule()$ function before each program statement of the task and ISRs in an IDP. That is, if a task or ISR consists of program statements $St_1, \ldots, St_n$, then we will get $St'_1, \ldots, St'_n$, where each St' is defined as: $St' \overset{\text{def}}{=} Schedule(); St$. The $Schedule()$ function works as follows: It passes through the interrupts the priority of which is higher than the current running task or ISR, and non-deterministically calls the ISR function of a higher-priority interrupt which has not yet been triggered. For the sake of presentation, in this subsection, we make the following assumption temporarily: Each statement in the task and ISRs is atomic and a higher-priority interrupt can be triggered at most once before each program statement of the task or a lower-priority interrupt. We use this assumption in this subsection to make the sequentialization method in [Kidd et al. 2010] easy to understand and we will show how to remove this assumption in Sect.3.2.

Fig. 3 shows the sequentialized program of the motivating example by adding explicit $Schedule()$ function before each program statement of the task and ISRs. In Fig. 3, N represents the number of interrupts (and $N = 1$ in this example), $Seq\_ISRs[N]$ (whose indices range from 1 to N) represents the corresponding sequentialized version of a fixed set of ISRs. E.g., $Seq\_ISRs[1].entry()$ represents $ISR1\_seq()$. The function $nondet()$ non-deterministically returns true or false. The function $task\_seq()$ is the entry of the sequentialized program. Besides, in this example, since we have only one interrupt, the $Schedule()$ functions added in $ISR1\_seq()$ can be all omitted and those $Schedule()$ functions added in $task\_seq()$ can be all replaced as

$$\mathbf{if}(nondet()) \quad ISR1\_seq();$$

Note that more generally, the call to $Schedule()$ can be omitted in the interrupt of the highest priority, since it can not be preempted by other interrupts as well as itself.

3.2. Sequentializing IDPs by considering data flow dependency

In Sect. 3.1, we have described an approach to sequentialize IDPs by adding explicit calls to a $Schedule()$ function before each program statement. However, the scale of the resulting sequentialized program may become very large, especially when an IDP contains many interrupts. In this subsection, we will show how to avoid adding unnec-

```
int x, y, z;
//current priority
int Prio = 0;
//ISR entry
ISR Seq_ISRs[N];
void task_seq(){
    int tx, ty;
    Schedule();
    tx = x;
    Schedule();
    ty = y;
    Schedule();
    if(tx < ty){
      Schedule();
      tx = x;
      Schedule();
      ty = y;
      Schedule();
      z = 1/(tx − ty);
    }
    Schedule();
    return;
}
```

```
void ISR1_seq(){
  int tx, ty;
  Schedule(); tx = x;
  Schedule(); tx = tx + 1;
  Schedule(); x = tx;
  Schedule(); ty = y;
  Schedule(); ty = ty − 1;
  Schedule(); y = ty;
  Schedule(); return;
}
 void Schedule(){
   //save current priority
   int prevPrio = Prio;
   for(int i = 1; i ≤ N; i++){
     if(i ≤ Prio) continue;
     if(nondet()){
       Prio = i;
       //call function ISRi_seq
       Seq_ISRs[i].entry();
   } }
   //restore priority
   Prio = prevPrio;
 }
```

Fig. 3. Sequentialization of the motivating example by simulating priority preemptive scheduling

essary calls to the $Schedule()$ function but still guarantee the soundness of sequentialization.

Essentially, in IDPs, the task and interrupts communicate with each other through shared variables. If a program statement does not access any shared variables, it makes no difference whether an interrupt is triggered before or after this statement. For example, suppose that the task consists of $St_1; \ldots; St_n$. Adding a call to $Schedule()$ before each program statement will give: $\{Schedule(); St_1; Schedule(); \ldots; Schedule(); St_n; \}$. However, if for each $i \in [1, n-1]$ the statement St_i does not access any shared variable, transforming to the following sequentialized program is still sound:

```
St_1; St_2; ...; St_{n-1};
for(int i = 1; i ≤ n; i++)
     Schedule();
St_n;
```

Using a loop to wrap a number of calls to the $Schedule()$ function has a key benefit that the resulted sequentialized program will be of much smaller size in code lines. Note that loops can be analyzed efficiently by using extrapolation techniques. Moreover, we could delay widening and even unroll the loop to achieve trade-off between efficiency and precision.

In fact, in practical IDPs, only a very small percentage of program statements will read from/write to shared variables. Moreover, the sets of shared variables between the task and different interrupts are usually different. Before a program statement $l = g$ that reads a shared variable g, we only need to consider those interrupts that, when triggered, will affect the value of g. Similarly, after a program statement $g = l$

that writes to a shared variable g, we only need to consider those ISRs that, when triggered, will be affected by the value of g.

Based on this insight, we could make use of the data flow dependency information over shared variables between the task and interrupts, to avoid certain unnecessary inserted $\mathit{Schedule}()$ function calls during sequentializing IDPs. To this end, we first introduce some notations.

Data flow dependency among interrupts. Let $\mathit{RSVars}(\mathit{ISR}_i)$ ($\mathit{WSVars}(\mathit{ISR}_i)$) be the set of shared variables read (written, respectively) by ISR_i. Interrupt ISR_i is *directly dependent* on ISR_j, denoted as $\mathit{ISR}_i \rightarrow \mathit{ISR}_j$, if $\mathit{RSVars}(\mathit{ISR}_i) \cap \mathit{WSVars}(\mathit{ISR}_j) \neq \emptyset$. Interrupt ISR_i is *transitively dependent* on ISR_j, denoted as $\mathit{ISR}_i \twoheadrightarrow \mathit{ISR}_j$, if $\mathit{ISR}_i \rightarrow \mathit{ISR}_j$ or there exists ISR_k such that $\mathit{ISR}_i \twoheadrightarrow \mathit{ISR}_k \wedge \mathit{ISR}_k \twoheadrightarrow \mathit{ISR}_j$.

The *dependent interrupt group* for interrupt ISR_i is defined as follows, where ISRs represents the set of interrupts $\{\mathit{ISR}_j \mid j \in [1, N]\}$.

$$\mathit{dGroup}[\mathit{ISR}_i] \stackrel{\text{def}}{=} \{I \in \mathit{ISRs} \mid \mathit{ISR}_i \twoheadrightarrow I\}.$$

The *influenced interrupt group* for interrupt ISR_i is defined as follows.

$$\mathit{iGroup}[\mathit{ISR}_i] \stackrel{\text{def}}{=} \{I \in \mathit{ISRs} \mid I \twoheadrightarrow \mathit{ISR}_i\}.$$

Example 1. Suppose that in an IDP, there are two shared variables x, y, three interrupts $\mathit{ISR}_1, \mathit{ISR}_2, \mathit{ISR}_3$, and

$$\begin{aligned}\mathit{RSVars}(\mathit{ISR}_1) &= \mathit{WSVars}(\mathit{ISR}_2) = \{x\}\\ \mathit{RSVars}(\mathit{ISR}_2) &= \mathit{WSVars}(\mathit{ISR}_3) = \{y\}\end{aligned}$$

Then, $\mathit{dGroup}[\mathit{ISR}_2] = \{\mathit{ISR}_3\}$ and $\mathit{iGroup}[\mathit{ISR}_2] = \{\mathit{ISR}_1\}$.

The data flow dependency relationships among ISRs can be described by a directed graph, which we call *dependency graph*. Each vertex of the graph denotes an interrupt and there exists a directed edge from ISR_i to ISR_j if $\mathit{ISR}_i \rightarrow \mathit{ISR}_j$. Then the problem of computing the dependent/influenced interrupt group for ISR_i can be reduced to a reachability problem in a directed graph. We use a matrix $DG \in \{0,1\}^{N \times N}$ to encode the graph where N is the number of interrupts, and

$$DG_{ij} \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } \mathit{ISR}_i \rightarrow \mathit{ISR}_j \\ 0 & \text{otherwise} \end{cases}$$

We use two procedures $\mathit{CompDepGroup}()$ and $\mathit{CompInfGroup}()$ to compute respectively the dependent and influenced interrupt groups for an interrupt, which can be essentially computed by the transitive closure of direct dependency relations among ISRs (encoded by the matrix DG).

Considering only statements that access a shared variable. As we have mentioned, if a program statement does not access any shared variable, it makes no difference whether an interrupt is triggered before or after this statement. For a program statement that reads a shared variable, we need to consider the influence from those ISRs that affect the value of this shared variable. For a program statement that writes into a shared variable, we need to consider the influence of this statement on those interrupts that may be affected by the value change of this shared variable.

Based on this insight, we propose the following strategy for sequentializing IDPs: We only add $\mathit{Schedule}()$ functions *before statements that read shared variables* and *after statements that write to* shared variables. We give the detail as follows:

- Before a statement (in the form of $l = g$) that reads a shared variable g, we consider invoking ISRs in $S_1 \cup S_2$ where

- $S_1 \overset{\text{def}}{=} \{I \in \mathit{ISRs} \mid g \in \mathit{WSVars}(I)\}$
- $S_2 \overset{\text{def}}{=} \{I \in \mathit{dGroup}[I'] \mid I' \in S_1\}$

where S_1 represents the set of ISRs that directly write to shared variable g and S_2 represents the set of ISRs that are in the dependent interrupt groups of any ISR in S_1. We use procedure $\mathit{ReadDepISRs}()$ to compute $S_1 \cup S_2$.

- After a statement (in the form of $g = l$) that writes to a shared variable g, we consider invoking ISRs in $S_3 \cup S_4$ where
  - $S_3 \overset{\text{def}}{=} \{I \in \mathit{ISRs} \mid g \in \mathit{RSVars}(I)\}$
  - $S_4 \overset{\text{def}}{=} \{I \in \mathit{iGroup}[I'] \mid I' \in S_3\}$

  where S_3 represents the set of ISRs that directly read shared variable g and S_4 represents the set of ISRs that are in the influenced interrupt groups of any ISR in S_3. We use procedure $\mathit{WriteInfISRs}()$ to compute $S_3 \cup S_4$.

On this basis, we introduce a new schedule function namely $\mathit{ScheduleG}(\mathit{group})$ to non-deterministically call those ISRs in group, as shown in Fig. 4. According to the above strategy, we consider adding the schedule functions only before and after a statement St that accesses shared variables, but we do not know how many times an interrupt may be triggered before the statement St. In fact, in practical IDPs, an interrupt is never triggered too frequently in each task period, otherwise the program may disobey the real-time restriction. Especially, the system designers of real-time embedded systems often know an upper bound on the number of firing times of each interrupt during one task period. Based on this insight, to guarantee the soundness of the sequentialization following the above strategy, we add the following assumption:

- We assume that the upper bound on the number of firing times of each interrupt during one task period is given by K where K is a positive integer that can be $+\infty$.

The bound K is given by the user, but he can always say $+\infty$ if he does not know. We guarantee that the semantics of the sequentialized program is sound with respect to K. In certain applications, K can be automatically inferred. For example, for nxtOSEK [Chikamasa et al. 2010] programs, some interrupts are periodically triggered and their periods are explicitly specified in an OIL (OSEK Implementation Language) file. In such case, the upper bounds K for these interrupts can be automatically inferred according to their periods.

In Fig. 4, we introduce a function namely $\mathit{ScheduleG\_Ks}(\mathit{group}, K)$ to call the $\mathit{ScheduleG}()$ function K times. In case where K can not be specified, we can put K to $+\infty$ (and then the loop in $\mathit{ScheduleG\_Ks}()$ becomes an unbounded loop that can stop at any time or loop forever), which can still guarantee the soundness of the sequentialization following the above strategy.

Now, we introduce two functions $\mathit{InvokeBefore}(St, \mathit{group})$ and $\mathit{InvokeAfter}(St, \mathit{group})$ to insert the $\mathit{ScheduleG\_Ks}(\mathit{group}, K)$ function respectively before and after a given statement St. During the process of sequentialization, for a statement St^R that reads a shared variable, we use $\mathit{InvokeBefore}(St^R, \mathit{group})$ to obtain the following sequentialized result:

$$St'^R \overset{\text{def}}{=} \mathit{ScheduleG\_Ks}(\mathit{group}, K); St^R;$$

For a statement St^W writing to a shared variable, we use $\mathit{InvokeAfter}(St^W, \mathit{group})$ to obtain the following sequentialized result:

$$St'^W \overset{\text{def}}{=} St^W; \mathit{ScheduleG\_Ks}(\mathit{group}, K);$$

Simplifying the sequentialized programs. We may notice that the sequentialization method based on the above strategy may still introduce unnecessary invocations of the function $\mathit{ScheduleG\_Ks}()$. For example, suppose that the task consists of

```
void $ScheduleG$($group$: int set){
  //save current priority
  int $prevPrio = Prio$;
  for(int $i = 1; i \leq N; i$++){
    if( $i \leq Prio \parallel i \notin group$ ) continue;
    if($nondet()$){
      $Prio = i$;
      $Seq\_ISRs[i].entry(group)$;
 } }
  //restore priority
  $Prio = prevPrio$;
}
//schedule $K$ times
void $ScheduleG\_Ks$($group$: int set, $K$:int){
  for(int $i = 1; i \leq K; i$++)
      $ScheduleG(group)$;
}
```

Fig. 4. Scheduling functions calling only those ISRs in a given interrupt group

$St_1^W; \ldots; St_n^R$, where St_1^W represents a statement that writes to a shared variable g_1 and St_1^R represents a statement that reads a shared variable g_2, while all statements in between do not access any shared variables. Then the sequentialized task will be:

$$\begin{array}{l} St_1; ScheduleG\_Ks(grp_1, K_1); \\ St_2; St_3; \ldots; St_{n-2}; St_{n-1}; \\ ScheduleG\_Ks(grp_2, K_2); St_n \end{array}$$

When the interrupt groups grp_1 and grp_2 are the same, the invoking of $ScheduleG\_Ks(grp_2, K_2)$ is unnecessary.

Based on this insight, we design a simplification procedure $Simplify()$ to remove unnecessary invocations of $ScheduleG\_Ks()$. The $Simplify()$ procedure removes the second invocation of $ScheduleG\_Ks(grp_1, K_1)$ in the following two patterns:

— $St_1^W; ScheduleG\_Ks(grp_1, K_1); \ldots; ScheduleG\_Ks(grp_1, K_1); St_n^R$;
— $ScheduleG\_Ks(grp_1, K_1); St_1^R; \ldots; ScheduleG\_Ks(grp_1, K_1); St_n^R$;

where for all $i \in [2, n-1]$ the statement St_i does not write to any shared variable.

3.3. Sequentializing IDPs by considering IMR

IDPs usually use an interrupt mask register (IMR) to control the interference between tasks and interrupts. Each bit of IMR corresponds to an interrupt and represents whether that interrupt is enabled or disabled. In our IDPs, programmers use $disableISR(i)$ and $enableISR(i)$ to change the value of IMR to disable and enable the i-th interrupt respectively. Hence, the value of IMR may be different at different program points.

Computing data flow dependency considering IMR. The value of IMR may affect the data flow dependency among interrupts. For example, suppose there are two shared variables x, y and three interrupts ISR_i, ISR_j, ISR_k where $RSVars(ISR_i)$ = $WSVars(ISR_j)$ = $\{x\}$ and $RSVars(ISR_j)$ = $WSVars(ISR_k)$ = $\{y\}$. Without considering the value of IMR, we have the following data flow dependency: $ISR_i \rightarrow ISR_j \wedge ISR_j \rightarrow ISR_k \wedge ISR_i \twoheadrightarrow ISR_k$. However, if ISR_j is disabled, there is no data flow dependency among ISR_i, ISR_j, ISR_k.

To obtain a precise analysis of data flow dependency, we need to consider only enabled interrupts when computing the data flow dependency among interrupts. In order to get the set of enabled interrupts, we need to compute the IMR value at each program point.

Pre-analysis for analyzing the value of IMR. In order to obtain the value of IMR at each program point, we use a simple pre-analysis (for each entry function of the task and interrupts separately) prior to the actual sequentialization. As explained in Sect. 2, all function calls are inlined. We do not need to consider function calls during the computation of IMR value. We recall the assumption related to IMR (described in Sect. 2), that is, at the exit-point of an ISR, the IMR is reset to the initial value of IMR at the entry-point of this ISR. Based on this assumption, we omit the effects of interruptions during the computation of IMR value.

ALGORITHM 1: Algorithm computing IMR for each statement

```
procedure $ComputeIMR(st : stmt, pimr : int)$
    $imr, imr_1, imr_2$ : int;
    match $st$ with begin
    | $disableISR(i) \rightarrow$
      $imr \leftarrow pimr \& !(1 << i);$
    | $enableISR(i) \rightarrow$
      $imr \leftarrow pimr | (1 << i);$
    | $S_1; S_2$
      $imr \leftarrow ComputeIMR(S_1, pimr);$
      $imr \leftarrow ComputeIMR(S_2, imr);$
    | if $e$ then $S_1$ else $S_2 \rightarrow$
      $imr_1 \leftarrow ComputeIMR(S_1, pimr);$
      $imr_2 \leftarrow ComputeIMR(S_2, pimr);$
      $imr \leftarrow imr_1 | imr_2;$
    | while $e$ do $S \rightarrow$
      $imr_1 \leftarrow ComputeIMR(S, pimr);$
      $imr \leftarrow pimr | imr_1;$
    | $\_ \rightarrow$
      $imr \leftarrow pimr;$
    end;
    $IMRValTbl.add(st, imr);$
    return $imr;$
end procedure
```

We design a procedure namely $ComputeIMR()$ to compute the value of IMR at each program point for each entry function of the task and interrupts separately. The value of IMR is modeled as a bit vector. For the i-th bit, we use 0 to represent that the i-th interrupt is disabled, 1 to represent that the i-th interrupt is enabled or inconclusive (i.e., either enabled or disabled). Note that when we can not conclude whether the i-th interrupt is enabled or disabled, we assign 1 to the i-th bit of IMR, which means that the interrupt is enabled in this case. As shown in Algorithm. 1, the procedure $ComputeIMR()$ performs a flow-sensitive data flow analysis using a bitwise abstract domain. For $disableISR(i)$ and $enableISR(i)$ statements, we set the i-th bit of the IMR bit-vector to 0 and 1 respectively. For the branch statement, at the control-flow join, we perform the bit-wise OR operation over the two resulting bit-vectors from different branches. In other words, for each bit, the join operation returns 0 if and only if the two corresponding input bits are 0, and otherwise returns 1.

Invoking ISRs for $\mathit{enableISR}(i)$ **and** $\mathit{disableISR}(i)$**.** Until now, we have considered adding calls to the $\mathit{Schedule}()$ function only before statements that read shared variables and after statements that write into shared variables. However, when an IDP includes statements $\mathit{enableISR}(i)$ and $\mathit{disableISR}(i)$, the above strategy may miss some invocations of certain ISRs. For example, Fig. 5 shows an IDP involving statements $\mathit{enableISR}(i)$ and $\mathit{disableISR}(i)$, wherein y is the shared variable. If we add invocations of ISRs only before statements that read shared variables and after statements that write to shared variables, we will not invoke *ISR1* because *ISR1* is disabled when the shared variable y is read (i.e., in line 7). However, this program may cause a division-by-zero error when *ISR1* is triggered between line 5 and line 6 in the $\mathit{task}()$.

```
1:  int y;
2:  void task(){
3:    int x, tmpy, z, c;
4:    c = 1; x = c;
5:    c = 2; y = c;
6:    disableISR(1);
7:    tmpy = y;
8:    z = 1/(x − tmpy);
9:    enableISR(1);
10: }
```

```
1: void ISR1(){
2:   y = 1;
3: }
```

Fig. 5. An example with *disableISR* and *enableISR*

Hence, when handling statements $\mathit{enableISR}(i)$ and $\mathit{disableISR}(i)$, we may also need to add an invocation of certain related ISRs. We use the following strategy: when handling $\mathit{disableISR}(i)$, we presume that all shared variables in $\mathit{WSVars}(\mathit{ISR}_i)$ will be read during the execution of the interrupt masking segment starting at $\mathit{disableISR}(i)$. In this situation, we need to add the $\mathit{Schedule}()$ function to invoke ISR_i and all those interrupts in $\mathit{dGroup}[\mathit{ISR}_i]$ before $\mathit{disableISR}(i)$. Similarly, when handling $\mathit{enableISR}(i)$, we presume that all shared variables in $\mathit{RSVars}(\mathit{ISR}_i)$ have been written to during the execution of the interrupt masking segment ending at $\mathit{enableISR}(i)$. In this situation, we need to add the $\mathit{Schedule}()$ function to invoke ISR_i and all those interrupts in $\mathit{iGroup}[\mathit{ISR}_i]$ after $\mathit{enableISR}(i)$.

Algorithm 2 shows how to insert calls to related ISRs before each statement. *IMRValTbl* represents a map from each program statement to its IMR value, which is computed by $\mathit{computeIMR}()$. $\mathit{InvokeBefore}()$ and $\mathit{InvokeAfter}()$ represent a call to the corresponding sequentialized interrupts before or after a statement. $\mathit{CompDepGroup}(i, p, \mathit{imr})$ (*CompInfGroup*) computes the dependent (influenced) interrupt group of ISR_i for the considered statement which lies in a task or an interrupt with priority p under the IMR value imr. $\mathit{ReadDepISRs}(g, p, \mathit{imr})$ (*WriteInfISRs*) computes the dependent (influenced) interrupt group for the considered statement which reads from (writes to) the shared variable g in a task or an interrupt with priority p under the IMR value imr.

3.4. Sequentializing IDPs by considering ISRs having no data flow dependency with tasks

The resulting sequentialized IDPs given by the above method consist of entry functions of the sequentialized task and the sequentialized ISRs. The entry function of the task is the main entry function of the whole IDP. In most cases, the sequentialized task will invoke all the sequentialized ISRs. However, there may exist special cases

ALGORITHM 2: Algorithm adding invocations of *ISRs* for each statement

```
procedure $StmtInvokeISRs(st : stmt, p : int)$
    $imr$ : int;
    $group$ : int set;
    match $st$ with
    | $l = g$ $\rightarrow$
      $imr \leftarrow IMRValTbl.find(st)$;
      $group \leftarrow ReadDepISRs(g, p, imr)$;
      $InvokeBefore(st, group)$;
    | $g = l$ $\rightarrow$
      $imr \leftarrow IMRValTbl.find(st)$;
      $group \leftarrow WriteInfISRs(g, p, imr)$;
      $InvokeAfter(st, group)$;
    | $disableISR(i) \rightarrow$
      $imr \leftarrow IMRValTbl.find(st)$;
      $group \leftarrow CompDepGroup(i, p, imr)$;
      $InvokeBefore(st, group)$;
    | $enableISR(i) \rightarrow$
      $imr \leftarrow IMRValTbl.find(st)$;
      $group \leftarrow CompInfGroup(i, p, imr)$;
      $InvokeAfter(st, group)$;
    | $S_1; S_2$
    | if $e$ then $S_1$ else $S_2$ $\rightarrow$
      $StmtInvokeISRs(S_1, p); StmtInvokeISRs(S_2, p)$;
    | while $e$ do $S$ $\rightarrow$
      $StmtInvokeISRs(S, p)$;
    | return when $p = 0$ $\rightarrow$
      //for the exit statement of the task
      $group \leftarrow CompNonInvokedISR()$;
      $InvokeBefore(st, group)$;
    | _ $\rightarrow$ ();
end procedure
```

where an interrupt ISR_i may never be invoked in the sequentialized result of the task function when using data flow dependency to optimize the sequentialization process. This is exemplified by Example 2. This situation may happen when there is no (direct or transitive) data flow dependency relations between ISR_i and task. Hence, when sequentializing IDPs, we need to specially consider those *ISRs* that have no data flow dependency relations with task.

Example 2. Suppose that an IDP consists of one task and two interrupts:

- $task$: $\{tmp = x;\}$
- ISR_1 : $\{tmp1 = x; tmp1 = 1/tmp1;\}$
- ISR_2 : $\{tmp2 = 0; x = tmp2;\}$

where ISR_2 is of higher priority than ISR_1, x is a shared variable and tmp, $tmp1$, $tmp2$ are non-shared variables. Following the strategy that we only consider invoking relevant ISRs before statements reading shared variables and after statements writing shared variables, ISR_1 will never be invoked in the sequentialization result of the $task$. In other words, whether ISR_1 happens or not will not affect the running of the task, and the running of the task will not affect the running of ISR_1, when fired. However, in this IDP, there will be a division-by-zero in ISR_1 when ISR_2 is triggered before ISR_1.

To enclose the firing situation of this kind of interrupts, after sequentializing the IDPs, we invoke the interrupts which are not invoked in the sequentialized task before the $return$ statement of task. We use a procedure $CompNonInvokedISR()$ to compute the set of interrupts which are not invoked in the sequentialized task. The simplest way to implement $CompNonInvokedISR()$ is to remember the set of interrupts that are invoked during sequentialization, and then the set of the rest interrupts are not invoked.

3.5. The overall sequentialization algorithm considering data flow dependency and IMR

Algorithm 3 shows the overall sequentialization algorithm considering both the data flow dependency and IMR. In Algorithm 3, the procedure $SeqIDP()$ is the main entry function of the overall sequentialization algorithm. N is the number of interrupts. $task$ and $ISRs[N]$ respectively represent the entry function of the task and interrupts. $IMRValTbl$ is a hash table that maps each program statement to the value of IMR at that statement. $IMRValTbl$ is computed by the procedure $computeIMR()$ discussed in Sect. 3.3. We use the function $SeqEach(entry, p)$ to sequentialize the task and an interrupt with function $entry$ and priority p separately. Without loss of generality, we set the priority of the task to 0.

For an IDP consisting of one task and N interrupts, the overall sequentialization algorithm shown in Algorithm 3 works as follows: First, we compute the value of IMR at each program point and build the data flow dependency graph among ISRs. Then, we sequentialize the task and each interrupt separately. For each program statement in the task and ISRs, we use $StmtInvokeISRs()$ showed in Algorithm. 2 to add the $Schedule()$ function to invoke relevant ISRs before or after that statement. Finally, we use the procedure $Simplify()$ to remove certain unnecessary calls to ISRs in the sequentialized IDPs.

ALGORITHM 3: A sequentialization algorithm considering data flow dependency

```
Require: $task, ISRs[N] : stmt\ list$;//$task$ and $ISR$ entry
  //IMR value for each program point
  $IMRValTbl : (stmt, int)\ Hashtbl$;
  procedure $SeqIDP(\,)$
      $IMRValTbl \leftarrow ComputeIMR()$;
      $BuildDepGraph()$;
      $SeqEach(task, 0)$;
      for ($i = 1\ to\ N$) do
         $SeqEach(ISRs[i], i)$;
      $Simplify()$;
  end procedure
  //sequentialize $task$ and each $ISR$
  procedure $SeqEach(fn : stmt\ list, p : int)$
      for( each $st_i \in fn$) do
         $StmtInvokeISRs(st_i, p)$;
  end procedure
```

3.6. Soundness of sequentialization

Our sequentialization algorithm considering data flow dependency will give a sound sequentialized program for an IDP, which means that the set of the program behaviors of the sequentialized program is an over-approximation of that of the original IDP.

In other words, all the program behaviors of the original IDP are included in that of the sequentialized program. The soundness of the sequentialization can be justified as follows:

(1) As described in Sect. 2, after transformation, an IDP only contains two kinds of statements accessing shared variables: reading from a shared variable (i.e., $l = g$) and writing to a shared variable (i.e., $g = l$). Moreover, we assume that all accesses to shared variables (i.e., $l = g$ or $g = l$) are atomic. Then before each program statement St, we add the following code:

$$\mathit{Schedule\_Ks} \overset{\text{def}}{=} \mathit{for}(\mathit{int}\ i = \mathit{0}; i < K; i{+}{+})\{\ \mathit{Schedule}();\ \}$$

where K is the upper bound of the number of firing times of any interrupt during one task period (we could conservatively set K to $+\infty$ for soundness). If the considered statement St does not access any shared variables, although it may be compiled into several instructions in binary code, when the interrupt is triggered (between any two instructions) during the execution of St will not affect the execution result of St. Hence, the program behaviors caused by triggering interrupts before or during the execution of St are included in the above sequentialized program fragment. If the statement St accesses shared variables (i.e., either $l = g$ or $g = l$), since we have assumed that it is atomic, the program behaviors caused by triggering interrupts before St are also included in the above sequentialized program fragment.

(2) On top of (1), we make two kinds of optimizations to remove unnecessary invocations of the $\mathit{Schedule}()$ function by considering the features of IDPs. First, the program fragment

$$\mathit{Schedule\_Ks}(); St_1; \mathit{Schedule\_Ks}(); St_2; \ldots; \mathit{Schedule\_Ks}(); St_n;$$

can be soundly transformed into

$$St_1; St_2; \ldots; \mathit{Schedule\_Ks}(); St_n;$$

when $St_i, i \in [1, n-1]$ does not access any shared variables and only St_n accesses a shared variable. This is because the situation that an interrupt is triggered before St_i (where $i \in [1, n-1]$) can be simulated by triggering it just before St_n. Note that the happening of the interrupt does not have any effects over the execution result of the statements $\{St_1; \ldots; St_{n-1};\}$. Besides, the number of firing times of any interrupt during the execution of $\{St_1; \ldots; St_{n-1}; St_n;\}$ is less than K. Second, we utilize the data flow dependency to remove unnecessary invocations of certain interrupts in the $\mathit{Schedule}()$ function. To be more clear, in the $\mathit{Schedule}()$ function, we do not invoke those interrupts that have no effects over the execution of the considered statement that accesses a shared variable. Before the statement $\{l = g;\}$, we consider only those interrupts whose triggering will (directly or indirectly) affect the value of shared variable g. Indeed, during the real execution of the IDP, an interrupt ISR_i whose triggering will not affect the value of g may be triggered exactly before $\{l = g;\}$. However, although we do not consider ISR_i before statement $\{l = g;\}$, following our sequentialization approach, we will consider ISR_i before a later statement $\{l' = g';\}$ if ISR_i's triggering will affect the value of shared variable g'. In other words, in the set of program behaviors of the whole sequentialized program, the firing of the interrupt ISR_i is still soundly considered. Similarly, after the statement $\{g = l;\}$, we consider only those interrupts whose triggering will be (directly or indirectly) affected by the value of shared variable g. The soundness of this case is similar to that of the case for $\{l = g;\}$.

To sum up, the program behaviors of the original IDP are over-approximated by the sequentialized program. In other words, our sequentialization approach is sound for the IDP model considered in this paper (under the assumptions listed in Sect. 2).

4. ANALYZING SEQUENTIALIZED IDPS VIA ABSTRACT INTERPRETATION

As we mentioned before, embedded software often involves lots of numerical computations and thus has the potential to contain errors related to numeric computations. In this section, we make use of abstract interpretation [Cousot and Cousot 1977] to analyze numerical properties of IDPs. We leverage existing numerical abstract interpretation techniques for sequential programs to analyze numerical properties of sequentialized IDPs given by the methods described in Sect. 3.

To perform numerical static analysis, there exist a variety of numerical abstract domains in the literature. For example, the interval abstract domain [Cousot and Cousot 1976] is a kind of non-relational abstract domains and can be used to infer numerical bounds for variables, i.e., $x \in [c, d]$. The octagon abstract domain [Miné 2006] is a kind of weakly relational abstract domain and can be used to infer numerical invariants in the form of $\pm x \pm y \leq c$ (where c is a constant). We employ these general numerical abstract domains for analyzing IDPs.

However, sequentialized IDPs also have their own specific features that may be uncommon in generic programs. These features may provide opportunities to get a better balance of precision and scalability of analysis, that is to design certain specific abstract domains according to the specific features of IDPs. In the following, we give two examples of specific abstract domains for IDPs.

Boolean flag abstract domain for specific ISRs. From practical IDPs, we observe that there is a specific family of interrupts which are fired after a fixed time interval. For example, some interrupts are triggered by timers. We call this kind of interrupts *periodic interrupts*. Furthermore, there is a kind of periodic interrupts whose periods are larger than one task period, which means that this kind of periodic interrupts are fired at most once during one task period. In this paper, we call this specific kind of interrupts as *at-most-once fired periodic interrupts*.

During numerical static analysis of IDPs that involve an at-most-once fired periodic interrupt ISR_i, whether ISR_i has been fired or not is an important information for the precision of the analysis. However, numerical abstract interpretation often performs flow-sensitive analysis rather than path-sensitive analysis. E.g., it can not distinguish whether an at-most-once fired interrupt has been fired or not. Consider analyzing $\{\mathbf{if}(nondet())\ ISRi();\}$. Let A_1 denote the abstract state (that is a map from program variables to values) in an abstract domain before this statement. E.g., when using the box abstract domain, an abstract state is a map from program variables to their value ranges. After this statement, abstract interpretation will perform a join operation (which computes the least upper bound of two abstract states) to compute the post abstract state as $A_1 \sqcup A_2$ where $A_2 \overset{\text{def}}{=} [\![ISRi()]\!]^\sharp(A_1)$ wherein $[\![ISRi()]\!]^\sharp$ denotes the abstract transfer function of $ISRi()$. Intuitively, in $A_1 \sqcup A_2$, A_1 denotes the abstract state when ISR_i has never been fired while A_2 denotes the abstract state after ISR_i has been fired. However, after the join operation, most numerical abstract domains will lose the information that the abstract states are different for the case where ISR_i has been fired and for the case where it has not been fired.

To handle this kind of imprecision, our basic idea is using a boolean flag variable f in the abstract domain to distinguish whether an at-most-once fired periodic interrupt has already been fired or not. In the abstract domain, the domain representation is $A \overset{\text{def}}{=} A_1 \times \ldots A_i \times \ldots A_n$ where n represents the number of at-most-once-firing interrupts in the program, $A_i \overset{\text{def}}{=} \langle A_i^f, A_i^{\neg f} \rangle, i \in [1, n]$, wherein A_i^f denotes the abstract state when ISR_i has already been fired and $A_i^{\neg f}$ denotes the abstract state when ISR_i has not been fired.

Most domain operations, such as join and widening, can be applied element-wise as follows:

$$\begin{aligned} A \sqcup A' &= (A_1 \sqcup A'_1) \times \ldots (A_i \sqcup A'_i) \times \ldots (A_n \sqcup A'_n) \\ A \nabla A' &= (A_1 \nabla A'_1) \times \ldots (A_i \nabla A'_i) \times \ldots (A_n \nabla A'_n) \end{aligned}$$

where $A = A_1 \times \ldots A_i \times \ldots A_n$, $A' = A'_1 \times \ldots A'_i \times \ldots A'_n$, $A_i \sqcup A'_i = \langle A_i^f \sqcup A'^f_i, A_i^{\neg f} \sqcup A'^{\neg f}_i \rangle$, $A_i \nabla A'_i = \langle A_i^f \nabla A'^f_i, A_i^{\neg f} \nabla A'^{\neg f}_i \rangle$ and $i \in [1, n]$. The transfer functions for the abstract domain can be applied element-wise like the above domain operations, except the transfer function for the branch statement which non-deterministically calls an at-most-once fired periodic interrupt, i.e., $\{\mathbf{if}(*)\ \mathit{ISRi}();\}$ where we use "$*$" to represent the $\mathit{nondet}()$ function for short. Let $A = A_1 \times \ldots A_i \times \ldots A_n$ denote the abstract state in the boolean flag abstract domain before $\{\mathbf{if}(*)\ \mathit{ISRi}();\}$. For those A_j where $j \neq i$, we apply the transfer function to the branch statement element-wise. For the abstract state A_i (which distinguishes whether *ISRi* has been fired or not), we apply the transfer function as follows: After the then branch, we get the post abstract state $\langle [\![\mathit{ISRi}()]\!]^\sharp(A_i^{\neg f}), \bot \rangle$; After the else branch, nothing changes and thus we get the pair $\langle A_i^f, A_i^{\neg f} \rangle$; Then, at the end of control-flow join, we perform an element-wise join operation to compute the post abstract state. This process can be formulated as:

$$[\![\mathbf{if}(*)\ \mathit{ISRi}()]\!]^\sharp A \stackrel{\text{def}}{=} [\![\mathbf{if}(*)\ \mathit{ISRi}()]\!]^\sharp A_1 \times \ldots [\![\mathbf{if}(*)\ \mathit{ISRi}()]\!]^\sharp A_i \times \ldots [\![\mathbf{if}(*)\ \mathit{ISRi}()]\!]^\sharp A_n$$

where

$$[\![\mathbf{if}(*)\ \mathit{ISRi}()]\!]^\sharp A_j \stackrel{\text{def}}{=} \begin{cases} \langle A_j^f \sqcup [\![\mathit{ISRi}()]\!]^\sharp(A_j^f), A_j^{\neg f} \sqcup [\![\mathit{ISRi}()]\!]^\sharp(A_j^{\neg f}) \rangle & \text{if } i \neq j \\ \langle A_j^f \sqcup [\![\mathit{ISRi}()]\!]^\sharp(A_j^{\neg f}),\ A_j^{\neg f} \rangle & \text{if } i = j \end{cases}$$

The boolean flag domain generally can be considered as a special case of partitioning technique (such as dynamic partitioning [Bourdoncle 1992][Jeannet et al. 1999] and trace partitioning [Mauborgne and Rival 2005]). However, it has its own specialities, tailored to analyze IDPs. The partitioning in this domain is designed according to the semantics of interrupts, rather than derived from the program syntax (e.g., using conditions for partitioning).

There exist two approaches to include boolean flags in our abstract states, providing different cost versus precision trade-offs. One approach is to partition the abstract state with respect to a bit vector where each bit denotes a flag: the bit is set to one if that interrupt has been fired once, and zero if that interrupt has never been fired, under the assumption that such interrupts cannot be fired more than once. In other words, one bit vector represents the firing states of all interrupts. E.g., a bit vector (consisting of 3 bits) 101 means that $\mathit{ISR_1}$ and $\mathit{ISR_3}$ have been fired while $\mathit{ISR_2}$ has not been fired yet. This partitioning is of high precision as it tracks every possible combination of interrupts. However, it may cause an exponential number of partitions. Another approach to include boolean flags is to represent each flag as an independent boolean variable. Assume that the set of periodic interrupts that are fired at-most-once is $\{\mathit{ISR_1}, ..., \mathit{ISR_n}\}$. For each $\mathit{ISR_i}$ we associate a boolean variable f_i to represent whether $\mathit{ISR_i}$ has already been fired (when $f_i = \mathit{true}$) or not (when $f_i = \mathit{false}$). This representation ignores the relationships between the status of ISRs, and is thus less precise. However, it will be more efficient, leading to only $2n$ cases instead of 2^n. E.g., suppose there is a program fragment:

```
if(∗) isr1();
if(∗) isr2();
if(∗) isr3();
```

If we use techniques like trace partitioning to enumerate all the possible paths, there exists 8 paths, which means that after the last statement we need to maintain 8 abstract states. If we only use a flag to represent whether an interrupt has been fired or not, we only need 6 (i.e., 2×3) abstract states. In this paper, we use the later manner for the sake of efficiency.

Example 3. Suppose that an IDP consists of one task and one interrupt where x is a shared variable, all other variables are non-shared variables and ISR is an at-most-once fired periodic interrupt:

— $task : \{x = 0;\ tx = x;\ tx = tx + 1;\ x = tx;\ z = x;\}$
— $ISR : \{tx = x;\ tx = tx + 10;\ x = tx;\}$

If we use only the interval abstract domain to analyze the program, at the end of the task, the resulting variable bounds are $\{x \in [1, 21], z \in [1, 21]\}$. However, if we use the boolean flag abstract domain on top of intervals, at the end of the task, the results will be $\{x \in [11, 11], z \in [11, 11]\}$ when ISR has been fired and $\{x \in [1, 1], z \in [1, 1]\}$ when ISR has not been fired. Obviously, the results given by the boolean flag abstract domain are more precise.

Syntactic equality abstract domain for sequentialized IDPs. From the sequentialized IDPs, we observe that there is a frequently-appearing code pattern that is assigning a shared variable to a temporary variable and then testing whether the temporary variable satisfies some branch conditions. The existence of such a code pattern is due to the fact that we transform all program statements involving shared variables into the form allowing only two kinds of atomic statements that can access shared variables (i.e., $g = l$ and $l = g$), by introducing auxiliary variables as we described in Sect. 2. All shared variables appearing in branch conditions in the original IDPs will be transformed into this code pattern. In Fig. 6(a), the original program tests whether a shared variable $thetaE$ satisfies some branch conditions and updates the shared variables according to the result of the test. In Fig. 6(b), we show the resulting program satisfying our IDP syntax after transformation by introducing a temporary variable $tmpthetaE$.

When using non-relational abstract domains, this kind of code pattern may cause loss of analysis precision. For example, during numerical static analysis of the transformed program in Fig. 6(b), the interval abstract domain can not infer that the invariant $thetaE >= 0x1FF$ holds at line 5. We could use some relational abstract domains, such as octagons and polyhedra, which can infer more precise invariants than the interval abstract domain. However, relational abstract domains are too costly for large-scale IDPs. In this paper, our idea is to solve the imprecision problem caused by this kind of code pattern by using a lightweight abstract domain.

```
1:  unsigned int thetaE;
2:  ...
3:   if (thetaE >= 0x1FF){
4:     thetaE = thetaE − 0x1FF;
5:   }
6:  ...
```

(a) original program fragment

```
1:  unsigned int thetaE, tmpthetaE;
2:  ...
3:  tmpthetaE = thetaE;
4:  if (tmpthetaE >= 0x1FF) {
5:    tmpthetaE = thetaE;
6:    tmpthetaE = tmpthetaE − 0x1FF;
7:    thetaE = tmpthetaE;
8:  }
9:  ...
```

(b) transformed program fragment

Fig. 6. A transformation example following our IDP syntax

Our main idea is tracking the syntactic equality relations between variables besides the value range information inferred by the interval abstract domain. Before testing branch conditions, we leverage the syntactic equality relations between variables to refine the value range in the interval abstract domain. Let A^{EQ} denote the abstract state of syntactic equality relations between variables, which maps each program variable to a set of program variables that have syntactic equality relations with it. Let $\mathbb{E}[\![v]\!](A^{EQ})$ denote the set of variables which are equal to the program variable v (including itself) in the abstract state A^{EQ}. E.g., the syntactic equality relation of Fig. 6(b) at line 4 is $\mathbb{E}[\![thetaE]\!](A^{EQ}) = \{thetaE, tmpthetaE\}$. The transfer function and the join operator for the syntactic equality abstract domain are defined as follows:

$$[\![x = y]\!]^{\sharp}(A^{EQ}) \overset{\text{def}}{=} \{A^{EQ}[x \to \mathbb{E}[\![y]\!](A^{EQ}) \cup \{y\}]\} \tag{1}$$

$$A_1^{EQ} \sqcup A_2^{EQ} \overset{\text{def}}{=} \{\forall x \in SV \cup NV, A_1^{EQ}[x \to \mathbb{E}[\![x]\!](A_1^{EQ}) \cap \mathbb{E}[\![x]\!](A_2^{EQ})]\} \tag{2}$$

where x, y represent program variables. Transfer function (1) represents adding a syntactic equality relation for those assignments whose left and right operands both are variables (rather than contants or compound expressions). For other forms of assignments, such as assigning a constant (e.g., $x = 1$) and assigning an expression (e.g., $x = y + z$), our method clears the syntactic equality relations of the assigned variable for the reason that these assignments destroy the previous syntactic equality relations. Function (2) defines the join operator of the syntactic equality abstract domain. Here we need *must* syntactic equality relations, and thus our method uses the set intersection to get the common syntactic equality relations existing in both abstract states. When testing a branch condition, we enumerate all forms of conditions derived by replacing each of the variable in the test condition by its corresponding syntactic equality variables. And during testing those derived branch conditions, our method propagates the syntactic equality relations to the value range abstract domain. In other words, we make use of syntactic equality relations to update the value ranges of variables. E.g., in Fig. 6, we get $thetaE \geq 0x1FF$ after line 4.

Our syntactic equality abstract domain is similar to the variable equality abstract domain in [Feret 2004], where Feret used an equality abstract domain to refine the digital filter analysis. Here, we make use of our syntactic equality domain to deal with the special code pattern in transformed IDPs.

We use equivalence classes to implement the syntactic equality relations. Moreover, we use a hash table to map each program variable to a equivalence class which is an ordered set of program variables. In order to save the space, we use the Union-Find algorithm to implement the operations of equivalence classes. So the space complexity of the syntactic equality abstract domain is $O(n)$ where n is the number of program variables. The time complexity of assignment transfer function is $O(log_2(n))$ because it has the same time complexity as the problem of inserting an element into an ordered set. While the time complexity for the join operator is $O(n)$ for the reason that we need n times comparison to put n variables into different equivalence classes during the intersect operation on ordered sets. Both the space and time complexity of our syntax abstract domain are $O(n)$. Compared with other relational abstract domains, such as the octagon abstract domain which has space complexity $O(n^2)$ and time complexity $O(n^3)$, our syntactic equality abstract domain is more lightweight.

5. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We have implemented a prototype tool to sequentialize IDPs, which uses CIL [Necula et al. 2002] as its front-end. We have also developed a numerical static analyzer for analyzing sequentialized IDPs based on the front-end CIL and the Apron [Jeannet and Miné 2009] numerical abstract domain library. We use a CIL supported inline tool to handle function calls. For pointers, we first use a flow-insensitive alias analysis as pre-analysis before sequentialization, to obtain an over-approximation result of possible aliases for shared pointer variables which will be used during sequentialization. During the analysis of sequentialized programs, we use a flow-sensitive pointer analysis to capture possible aliases for both shared and non-shared pointer variables. However, we do not allow tasks and interrupts to access the same memory cell through pointer arithmetic over physical addresses without using an explicit variable name to represent that memory cell. For structures, we use an access path based memory model to identify different fields of the structure. For unions, we use an offset based memory model to analyze the potential shared variables. Although the field names of a union type are various, the offset with respect to the starting address is canonical.

Our experiments were conducted on a selection of benchmarks and real-world programs listed in Fig. 7. $Motv\_Ex$ is the motivating example shown in Fig. 1. $DataRace\_Ex$ and $Privatize$ come from a data race detection tool for IDPs namely Goblint [Schwarz et al. 2011]. Nxt_gs[1] is a robot control program from LEGO company samples which implements a two-wheeled robot keeping balance and avoiding obstacles. $UART$ (Universe Asynchronous Receiver and Transmitter)[2] is from an open source website which implements a First-In First-Out (FIFO) buffer. $iRobot3$[3] coming from [Kotker and Dorsa Sadigh 2011] is a simplified control software used in the autonomous robot platform of iRobot Create. The goal of $iRobot3$ is to control a cleaner robot avoiding obstacles and cliffs. HBM (Heart Beat Monitor)[4] is an application for a micro-controller at89s51(8051) which counts the heart beat number by a sensor and displays the number on a LCD screen every minute. The last three benchmarks are coming from aerospace industry applications. $Ping\_pong$ is an implementation of ping-pong buffer (or double buffering that is a technique to use two buffers to speed up a computer that can overlap I/O with processing). ADC_Ctl (Analog-Digital Conversion Control) is an application software in aerospace embedded systems, which samples analog data, converts the analog data to digital data and sends the digital data through a peripheral bus. Sat_Ctl is a control software for an aerocraft which gets data from sensors and controls the flying trace of the aerocraft. Some of these programs originally do not include interrupts, such as Nxt_gs, but the tasks in these programs are scheduled using a priority-based real-time scheduler, which behaves similarly to IDPs. Hence, we adapt them into IDPs during our experiments.

Fig. 7 shows the sequentialization results of all the benchmarks. OLT and OLI represent respectively the original code size in lines of task and interrupts. $\#Vars$ represents the number of variables in programs. $\#ISR$ represents the number of interrupts. SEQ represents the sequentialization method described in Sect. 3.1 which is inspired from [Kidd et al. 2010]. DF_SEQ represents the sequentialization method described in Sect. 3.5 which considers data flow dependency and IMR. From the results, we can see that the code size of the program given by DF_SEQ is much smaller than that given by SEQ. For example, for Sat_Ctl, the code size of the program given by DF_SEQ is around 5% of the code size of that given by SEQ.

Fig. 8 shows the analysis results of analyzing sequentialized IDPs by numerical abstract interpretation. We use the *box* and *octagon* abstract domains in the APRON library to analyze the sequentialized IDPs. We use "-" to represent that the numerical analysis runs out of memory. For $Motv\_Ex$, we find the expected division-by-zero error. For $DataRace\_Ex$ and $Privatize$, our method can prove their assertions. For example,

[1] http://lejos-osek.sourceforge.net/

[2] http://www.mikrocontroller.net/topic/101472#882716

[3] http://uclid.eecs.berkeley.edu/gametime/fmcad11/

[4] https://github.com/pankajshanbhag/heartbeat

Program					Sequentialization				
Name	OLT	OLI	#Vars	#ISR	SEQ		DF_SEQ		DF_SEQ/SEQ
					LOC	Time (s)	LOC	Time (s)	(%LOC)
Motv_Ex	10	7	8	1	158	0.004	134	0.006	84.81
DataRace_Ex	20	40	9	2	385	0.004	242	0.005	62.86
Privatize	25	37	7	2	393	0.006	168	0.004	42.75
Nxt_gs	23	154	27	1	1199	0.005	552	0.006	46.04
UART	129	15	47	1	5940	0.010	1215	0.010	20.45
iRobot3	114	80	55	1	2986	0.035	793	0.034	26.56
HBM	500	85	36	2	9832	0.056	1312	0.053	13.34
Ping_pong	130	53	21	1	3159	0.006	842	0.006	26.65
ADC_Ctl	1950	2880	334	1	1.2M	0.541	404K	0.520	33.67
Sat_Ctl	33885	1227	1352	1	10M	16.1	534K	1.6	5.34

Fig. 7. Experimental results on sequentializing IDPs

$Privatize$ asserts that a shared variable is always equal to 1. For Nxt_gs, our analysis issues a number of integer overflow alarms. This is due to the fact that in Nxt_gs many variables are assigned data from sensors. For soundness, our analysis sets these variables to the full range of the data type and then arithmetic operations over these variables may cause integer overflow. For $UART$ and $Ping\_Pong$, our method can prove that there is no array-out-of-bound error. For $iRobot3$, our method issues two integral overflow alarms due to the same reason as Nxt_gs.

Program	Analysis time of SEQ (s)		Analysis time of DF_SEQ (s)		Found properties or errors
	BOX	OCT	BOX	OCT	
Motv_Ex	0.007	0.011	0.006	0.007	division-by-zero error
DataRace_Ex	0.042	0.053	0.011	0.015	assertions hold
Privatize	0.029	0.036	0.005	0.007	assertions hold
Nxt_gs	0.113	0.140	0.040	0.046	2 integer overflow alarms
UART	0.732	5.782	0.128	1.177	no array-out-of-bound
iRobot3	0.303	2.979	0.069	0.636	2 integer overflow alarms
HBM	0.887	5.661	0.112	1.076	4 integer overflow alarms
Ping_pong	0.429	2.434	0.054	0.251	no array-out-of-bound
ADC_Ctl	-	-	343.5	-	70 arithmetic overflow alarms
Sat_Ctl	-	-	5325	-	538 alarms

Fig. 8. Experimental results on analyzing sequentialized IDPs

For HBM, using the *box* abstract domain issues 4 integer overflow alarms. Without considering the application scenario, two of the alarms are false alarms which can be eliminated using the octagon abstract domain. When we consider the application scenario, the other two alarms are also false alarms and can be eliminated by adding some assumptions. As we explained before, HBM is an application for heart beat monitor, which consists of one task and two interrupts. The lower priority interrupt reads some sensors and increments a program variable bt by 1 to store the heart beat number. The higher priority interrupt is a timer and increments a program variable min by 1 every 60 seconds. Our method issues two integer overflow alarms for the incrementation of variables bt and min for the reason that the upper bounds on the firing number of these two interrupts are unknown. However, we can infer reasonable upper bounds on the firing number of these two interrupts by considering the real world application scenario. For instance, suppose that the upper bound of heart beat counting interrupt can be set to 200 (i.e. assuming heart beat number is less than 200 times in one minute) and the upper bound of the timer interrupt is 60 in every minute. In such a case, these two integer overflow alarms can be eliminated by using the octagon abstract domain combined with these upper bound assumptions.

For ADC_Ctl, our method issues 70 arithmetic overflow alarms. After analyzing these alarms, we can divide these alarms into two classes : 20 potential overflow alarms and 50 false positives. For those potential overflow alarms, there are two kinds of typical alarms. One kind of such alarms are caused by the program incrementing a global variable by 1 without resetting it. In Fig.9(a), when the state variable $stateAD$ is abnormal, the variable emg is incremented by 1 at line 9, but without being reset. During the execution, there may be integer overflow over this variable because this program fragment may be executed for an unbounded number of times. Such kind of integer overflow may cause serious accident in critical embedded systems. E.g., an integer overflow bug found in Boeing 787 control software may result in loss of control of the airplane [Goodin 2015]. According to [Goodin 2015], the integer overflow is caused by continuously adding a counter but without reseting it, which is quite similar to the case of integer overflow alarms found in ADC_Ctl. Another set of alarms are caused by assigning negative values to unsigned variables as shown in Fig.9(b). At line 7, the program variable $speed$ with unsigned integer type will be assigned by a negative value. In fact, for the same value (i.e., binary representation), the results are quite different, when considering signed and unsigned. E.g., in Fig.9(b), for unsigned integer the value of speed after line 7 is about 4×10^9 while for signed integer the value is -839. Although, in the program, we found that all accesses of $speed$ only manipulate the low 24 bits, this is still an unsafe use. The last 50 false positives are caused by the over approximation of our static analysis method. Furthermore, we will show how to remove part of those false positives by our syntactic equality abstract domain in the later part.

For Sat_Ctl, our method issues 538 warnings, including 473 arithmetic overflow alarms, 19 division by zero alarms and 46 array out of bounds alarms. Most of arithmetic overflow alarms are caused by type cast assignments, such as assigning an unsigned integer to a char. All of the division by zero alarms are caused due to the fact that the divider is a result of a non-linear function. E.g., when the program uses the result of $sin(x)$ as the divider, our analysis will return the interval $[-1, 1]$ as the result of $sin(x)$, which contains zero and thus causes division by zero alarms. Some of the array out of bound alarms are due to the fact that our analysis of array indices in nested loops is not precise enough.

From the analysis time, we can see that analyzing the sequentialized program given by DF_SEQ is much faster than analyzing the one given by SEQ. This is due to the fact that the code size of the resulting sequentialized IDPs given by DF_SEQ is much smaller than that given by SEQ. Although the resulting sequentialized IDPs given by DF_SEQ may contain more loops, abstract interpretation can deal with loops efficiently using extrapolation techniques such as widening.

In order to know how many warnings are caused by interrupts, we also analyze the IDPs after disabling all interrupts (i.e., only analyzing the tasks) and enabling all interrupts (i.e., removing all the interrupt-disabling statements in the programs). The results are shown in Fig. 10, where OF represents arithmetic overflow, DivZ represents division by zero and AOB represents array out of bound. We select ADC_Ctl and Sat_Ctl as the benchmarks because the other programs do not contain enable and disable interrupt statements ($Motv\_Ex$) or there is no interrupt-disabling statements after the initialization process of the main application ($DataRace\_Ex$,$Privatize$,$UART$,$Ping\_pong$). For ADC_Ctl, comparing the analysis results of disabling all interrupts and enabling all interrupts, we see that most of the warnings are caused by interrupts. Compared with the analysis results of DF_SEQ, enabling all interrupts finds 2 more warnings including 1 true division by zero error

```
int $emg, stateAD$;
...
 if ($stateAD == 0x0$)
   $stateAD = 0x1$;
 else if ($stateAD == 0x1$)
   $stateAD = 0x2$;
 else if ($stateAD == 0x2$)
   $stateAD = 0x0$;
 else $emg{+}{+}$;
 ...
```

(a) integer overflow

```
unsigned int $speed$;
float $avg$;
 ...
 if ($avg > 1.57f$)
  $speed$ =(unsigned int)((int)($1.57 * 534.72$)));
 else if ($avg < -1.57f$)
  $speed$ =(unsigned int)((int)($-1.57 * 534.72$)));
 else
   $speed$ =(unsigned int)((int)($avg * 534.72$)));
 ...
```

(b) float overflow

Fig. 9. Arithmetic overflow program fragments in *ADC_Ctl*

Program	all-*ISRs*-enabled warnings				all-*ISRs*-disabled warnings				*DF_SEQ* warnings			
	OF	DivZ	AOB	total	OF	DivZ	AOB	total	OF	DivZ	AOB	total
ADC_Ctl	71	1	0	72	8	0	0	8	70	0	0	70
Sat_Ctl	477	19	46	542	302	11	14	327	473	19	46	538

Fig. 10. Experimental results on analyzing programs with enabling and disabling interrupts

and 1 arithmetic overflow false alarm. For *Sat_Ctl*, compared with the analysis results of *DF_SEQ*, enabling all interrupts issues only few more warnings. The reason is that the purpose of disabling interrupts statements in *Sat_Ctl* is to avoid functional incorrectness, e.g., to avoid data inconsistency, whereas our method focuses on finding run-time errors.

Experimental results on boolean flag abstract domain. Fig. 11 shows the analysis results of analyzing IDPs with at-most-once fired periodic interrupts. In Fig. 11, BF denotes the boolean flag abstract domain described in Sect. 4. #FP denotes the number of false alarms. *Example_4* is an adapted version of the program in Example 4 in Section 4 by adding an assertion $x \leq 20$ at the end of the task. *Division_Ex* is an example that involves a division operation in the task. For *Example_4* and *Division_Ex*, the analysis using only the octagon domain issues false alarms, while using our boolean flag abstract domain on top of the octagon domain (denoted by BF+OCT) can eliminate these false alarms. This is because our boolean flag abstract domain can make use of the information that the interrupt is an at-most-once fired periodic interrupt. *SeekRobot*[5] is a nxtOSEK platform application software, which consists of priority tasks. Moreover, tasks in *SeekRobot* are periodic tasks and satisfy at-most-once fired constraints. E.g., given one task with priority 1 and period 50 ms while another task with priority 5 and period 100 ms , we know that during one execution of the lower priority task, the higher priority task can be triggered at-most-once. As we mentioned before, our method supports this kind of program by considering tasks with priority as interrupts. We adapt the lower priority task to contain a loop which increments a shared variable of type *char* by 1 until 126 (which is the upper bound of type *char* minus one). For the higher priority task, we add a statement which increments the shared variable by 1. According to the at-most-once fired constraints, there will be no overflow for this shared variable. When performing analysis using the octagon abstract domain, there will be 1 overflow alarm for the incrementation of the shared variable. Our boolean flag abstract domain on top of the octagon abstract domain can eliminate this false alarm by considering that the highest priority task is fired at-most-once during one execution of the lower priority task.

[5] https://github.com/smiler/nxt-osek-sumo

$Posix\_Ex$[6] is an application in Trampoline [Trampoline 2015] which is a static RTOS for small embedded systems. $Posix\_Ex$ consists of 2 tasks. The period of the lower priority task is 100 ms while the period of the higher priority task is 1000 ms. So the higher priority task of $Posix\_Ex$ satisfies the at-most-once fired constraints. We adapt the lower priority task to contain a loop which reads data from a shared buffer and increments the index of the buffer. For the higher priority task, we add a statement which reads the shared buffer and increments the index of the buffer. When we perform the analysis using the octagon abstract domain, there will be 1 array out of bound false alarm over the accessing of shared buffer. Our boolean flag abstract domain on top of the octagon abstract domain can eliminate this false alarm by taking the at-most-once fired constraints into consideration.

For the analysis time, we can see that analyzing the sequentialized program given by BF+OCT is slower than OCT. This is due to the fact that our boolean flag abstract domain needs to maintain extra information for at-most-once fired interrupts.

Program			Analysis of SEQ (s)					Analysis of DF_SEQ (s)				
Name	OLT	OLI	LOC	OCT		BF+OCT		LOC	OCT		BF+OCT	
				time (s)	#FP	time (s)	#FP		time (s)	#FP	time (s)	#FP
Example_4	6	11	158	0.007	1	0.012	0	122	0.005	1	0.010	0
Division_Ex	8	10	189	0.007	1	0.013	0	99	0.004	1	0.007	0
SeekRobot	56	710	15921	4.852	1	6.570	0	5241	1.552	1	3.418	0
Posix_Ex	15	10	255	0.028	1	0.040	0	111	0.016	1	0.022	0

Fig. 11. Experimental results on analyzing IDPs with at-most-once fired periodic interrupts

Experimental results on syntactic equality abstract domain. Fig. 12 shows the analysis results of analyzing IDPs using syntactic equality abstract domain. In Fig. 12, EQ denotes the syntactic equality abstract domain described in Sect. 4. $\#Warnings$ denotes the number of warnings issued by each abstract domain. For $UART$ and $Ping\_pong$, our syntactic equality abstract domain combined with the interval abstract domain (denoted by EQ+BOX) issues no warning, the same as the analysis results given by the interval and octagon abstract domain. For $iRobot$, as we mentioned before, these two warnings are caused by unknown values from sensors, which can not be eliminated by improving the ability of abstract domain. For HBM, the syntactic equality abstract domain can eliminate one false alarm due to the fact that in HBM, a shared variable in a branch condition is replaced with temporary variables. As a non-relational abstract domain, the interval abstract domain can not find the equality relations between these variables, while our syntactic equality abstract domain maintains these equality relations and eliminates this false alarm. The octagon abstract domain is a weakly relational abstract domain, which can infer equality relationship invariants (such as $x - y \leq 0 \wedge y - x \leq 0$), so the octagon abstract domain can eliminate this false alarm too. Moreover, when adding the upper bound on the firing number of each interrupt, the octagon abstract domain infers the relations between shared variables and the upper bounds (e.g., $x \leq y \wedge 0 \leq y \leq c$) and eliminates the other two false alarms. However, the interval and the syntactic equality abstract domain can not infer this kind of invariants. For ADC_Ctl, compared with only using the interval abstract domain, the syntactic equality abstract domain eliminates 8 false alarms for the similar reason as HBM. For Sat_Ctl, compared with using only the interval abstract domain, the syntactic equality abstract domain eliminates 3 false alarms including 1 arithmetic

[6] https://github.com/TrampolineRTOS/trampoline/tree/master/examples/posix

overflow alarms and 2 array out of bound alarms for the similar reason as HBM. However, the octagon abstract domain runs out of memory for these two programs ADC_Ctl and Sat_Ctl.

For the analysis time, compared with using only the interval abstract domain, the syntactic equality abstract domain causes some overhead. For small scale programs, these overhead can almost be neglected. For ADC_Ctl and Sat_Ctl, the overhead is less than 1.4 times to the analysis time of using the interval abstract domain. Compared with the time consumption of the octagon abstract domain, the syntactic equality abstract domain has a speed of the same order of magnitude as using only the interval abstract domain.

Program	Analysis of BOX		Analysis of EQ + BOX		Analysis of OCT	
Name	Time(s)	#Warnings	Time(s)	#Warnings	Time(s)	#Warnings
UART	0.128	0	0.178	0	1.177	0
iRobot3	0.069	2	0.073	2	0.636	2
HBM	0.112	4	0.127	3	1.076	0
Ping_pong	0.054	0	0.125	0	0.251	0
ADC_Ctl	343.5	70	480.6	62	-	-
Sat_Ctl	5325	538	5664	535	-	-

Fig. 12. Experimental results on analyzing IDPs with syntactic equality abstract domain

6. RELATED WORK

Sequentialization. Much work has been done on sequentializing concurrent programs. Qadeer et al. [Qadeer and Wu 2004] propose a context bounded analysis (CBA) method for concurrent programs via sequentialization. Their sequentialization method adds non-deterministic calling function of other threads and non-deterministic return statements to the previous executing thread before each program statement. Their method finds numerous bugs in device drivers with a fixed context bound 2. However, their method cannot be generalized to an arbitrary context bound. Lal et al. [Lal et al. 2008] propose a CBA method based on sequentialization for concurrent programs with an arbitrary given context bound. Their sequentialization method uses different copies of the shared global memory in different context and uses assumptions to prune infeasible runs. However, their method uses a large number of extra variables which cause a high degree of nondeterminism. Inverso et al. [Inverso et al. 2014] propose a CBA method based on sequentialization for concurrent programs. Their method reduces the nondeterminism of sequentialized programs to avoid exponentially growing formula sizes in the model checking of sequentialized programs. Recently, Chaki et al. [Chaki et al. 2013] present a CBA method for analyzing periodic programs based on sequentialization.

Kidd et al. [Kidd et al. 2010] propose a sequentialization method for priority preemptive scheduling systems in which each task is periodic. The key idea is to use a single stack for all tasks and to model preemptions by function calls. Edwards [Edwards 2003] surveys a variety of approaches for translating concurrent specifications (these concurrent specifications are more abstract than concurrent programs) into sequential code which can be efficiently executed.

Compared with the above work, our sequentialization method is specifically designed for IDPs. Moreover, our method makes use of the data flow dependency among tasks and interrupts to reduce the size of the sequentialized program. In addition, we consider analyzing numerical properties of the sequentialized programs using numerical abstract interpretation.

Numerical static analysis of embedded software. Most of the existing numerical static analysis approaches consider sequential programs. Astrée [Blanchet et al. 2003] is one of the famous numerical static analyzers for sequential programs, which has been successfully used in analyzing flight control software.

Mine [Miné 2011] proposes a numerical static analysis method for parallel embedded software. Their method iterates each thread in turn until all threads interferences stabilize. The interference is used to abstract the effects of a thread on the shared variables. One main difference between interferences and the data flow dependency is that interferences consider the influences of each thread to shared variables, whereas our data flow dependency considers both the dependency and influence relations among interrupts, linked through shared variables. Recently their work extends to support relational numerical properties [Miné 2014]. The scalability of their method is pretty good due to the fact that the method is thread-modular. Compared with their work, our work targets IDPs and we take into consideration the priority and firing times of interrupts, so we can get more precise analysis results for IDPs.

Cooprider et al. [Cooprider and Regehr 2006] propose a static analysis method for embedded software to reduce the code size. Beckschulze et al. [Beckschulze et al. 2012] propose a data race analysis method for lockless micro-controller programs considering hardware architecture. Compared with their work, our method focuses on numerical properties of IDPs and considers data flow dependency among tasks and interrupts. Monniaux [Monniaux 2007] proposes a numerical static analysis method for a concurrent USB driver. Their method dynamically invokes interrupts for every access to the shared memory in tasks. Compared with their method, our method first sequentializes the concurrent program to sequential program, which has a key benefit that is the ability to leverage the power of the state-of-the-art analysis and verification techniques for sequential programs to analyze IDPs.

Analysis of interrupt-driven programs. In the literature, there are a few work on analyzing and verifying IDPs.

Brylow et al. [Brylow et al. 2001] propose a static analysis method for interrupt-driven Z86-based software. Their method first generates the control flow graph of the IDPs which considers the effects from interrupts. Based on the control flow graph, the approach uses a model checking algorithm of pushdown systems to analyze the upper bounds of stack sizes and interrupt latencies of IDPs. Most of the existing work focus on object code and consider problems such as interrupt latency [Brylow and Palsberg 2004], stack size [Regehr et al. 2005], data race [Schwarz et al. 2011].

Compared with the above work, our method analyzes the source code of IDPs rather than object code and focuses on numerical properties of IDPs.

7. CONCLUSION

We have presented a sound numerical static analysis approach for IDPs. The key idea is to sequentialize IDPs into sequential programs before analysis. The idea of sequentializing IDPs into sequential programs enables the use of existing analysis and verification techniques (e.g., bounded model checking, symbolic execution, etc.) for sequential programs to analyze and verify IDPs. We have proposed a sequentialization algorithm specifically for IDPs, by considering the data flow dependency among ISRs and specific hardware features of IDPs. After that, we have shown how to use numerical abstract interpretation to analyze numerical properties of the sequentialized IDPs. By considering specific features of the sequentialized IDPs, we design and make use of specific abstract domains to analyze the sequentialized IDPs. The preliminary results show that our method is promising.

For future work, we will consider designing more specific abstract domains that fit IDPs and conducting more experiments on large realistic IDPs. We plan to extend our IDP model and our sequentialization method to support mixing of interrupts and multi-threads. Besides, we also plan to extend our IDP model to support weak memory models.

REFERENCES

Eva Beckschulze, Sebastian Biallas, and Stefan Kowalewski. 2012. Static Analysis of Lockless Microcontroller C Programs. In *SSV'12*. 103–114.

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *PLDI'03*. ACM, 196–207.

François Bourdoncle. 1992. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming* 2, 04 (1992), 407–435.

Dennis Brylow, Niels Damgaard, and Jens Palsberg. 2001. Static Checking of Interrupt-Driven Software. In *ICSE'01*. IEEE, 47–56.

Dennis Brylow and Jens Palsberg. 2004. Deadline Analysis of Interrupt-Driven Software. *IEEE Trans. Software Eng.* 30, 10 (2004), 634–655.

Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. 2013. Verifying periodic programs with priority inheritance locks. In *FMCAD'13*. 137–144.

Takashi Chikamasa, Eiji Sato, and Koji Shimizu. 2010. OSEK platform for LEGO ® MINDSTORMS ®. http://lejos-osek.sourceforge.net/. (2010).

Nathan Cooprider and John Regehr. 2006. Pluggable abstract domains for analyzing embedded software. In *LCTES'06*. ACM, 44–53.

Patrick Cousot and Radhia Cousot. 1976. Static determination of dynamic properties of programs. In *Proc. of the 2nd International Symposium on Programming*. Dunod, Paris, 106–130.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*. ACM, 238–252.

Stephen A. Edwards. 2003. Tutorial: Compiling concurrent languages for sequential processors. *ACM Trans. Design Autom. Electr. Syst.* 8, 2 (2003), 141–187.

Jérôme Feret. 2004. Static analysis of digital filters. In *ESOP'04*. Springer, 33–48.

Dan Goodin. 2015. Boeing 787 Dreamliners contain a potentially catastrophic software bug. http://arstechnica.com/information-technology/2015/05/boeing-787-dreamliners-contain-a-potentially-catastrophic-software-bug/. (2015).

Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2014. Bounded Model Checking of Multi-threaded C Programs via Lazy Sequentialization. In *CAV'14 (LNCS)*, Vol. 8559. Springer, 585–602.

Bertrand Jeannet, Nicolas Halbwachs, and Pascal Raymond. 1999. Dynamic partitioning in analyses of numerical properties. In *SAS'99*. Springer, 39–50.

Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV'09 (LNCS)*, Vol. 5643. Springer, 661–667.

Nicholas Kidd, Suresh Jagannathan, and Jan Vitek. 2010. One Stack to Run Them All - Reducing Concurrent Analysis to Sequential Analysis under Priority Scheduling. In *SPIN'10 (LNCS)*, Vol. 6349. Springer, 245–261.

Jonathan Kotker and Sanjit A. Seshia Dorsa Sadigh. 2011. Timing Analysis of Interrupt-Driven Programs under Context Bounds. In *FMCAD '11*. IEEE Press, 81–90.

Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas W. Reps. 2008. Interprocedural Analysis of Concurrent Programs Under a Context Bound. In *TACAS'08 (LNCS)*, Vol. 4963. Springer, 282–298.

Laurent Mauborgne and Xavier Rival. 2005. Trace partitioning in abstract interpretation based static analyzers. In *ESOP'05*. Springer, 5–20.

Antoine Miné. 2006. The octagon abstract domain. *Higher-Order and Symbolic Computation* 19, 1 (2006), 31–100.

Antoine Miné. 2011. Static Analysis of Run-Time Errors in Embedded Critical Parallel C Programs. In *ESOP'11 (LNCS)*, Vol. 6602. Springer, 398–418.

Antoine Miné. 2014. Relational Thread-Modular Static Value Analysis by Abstract Interpretation. In *VMCAI'14 (LNCS)*, Vol. 8318. Springer, 39–58.

David Monniaux. 2007. Verification of device drivers and intelligent controllers: a case study. In *EMSOFT'07*. ACM, 30–36.

George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC'02 (LNCS)*, Vol. 2304. Springer, 213–228.

Shaz Qadeer and Dinghao Wu. 2004. KISS: keep it simple and sequential. In *PLDI'04*. ACM, 14–24.

John Regehr, Alastair Reid, and Kirk Webb. 2005. Eliminating stack overflow by abstract interpretation. *ACM Trans. Embedded Comput. Syst.* 4, 4 (2005), 751–778.

Martin D. Schwarz, Helmut Seidl, Vesal Vojdani, Peter Lammich, and Markus Müller-Olm. 2011. Static analysis of interrupt-driven programs synchronized via the priority ceiling protocol. In *POPL'11*. ACM, 93–104.

Trampoline. 2015. OpenSource RTOS project. http://trampoline.rts-software.org/. (2015).

Xueguang Wu, Liqian Chen, Antoine Miné, Wei Dong, and Ji Wang. 2015. Numerical Static Analysis of Interrupt-driven Programs via Sequentialization. In *EMSOFT '15*. IEEE Press, 55–64.

Wenhua Yang, Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2015. A survey on dependability improvement techniques for pervasive computing systems. *SCIENCE CHINA Information Sciences* 58, 5 (2015), 1–14.