# Numerical Static Analysis of Interrupt-Driven Programs via Sequentialization

Xueguang Wu, Liqian Chen, Antoine Miné, Wei Dong, Ji Wang

# Numerical Static Analysis of Interrupt-Driven Programs via Sequentialization

Xueguang Wu [*†] Liqian Chen [†] Antoine Miné [‡] Wei Dong [†] Ji Wang [*†]
{xueguangwu, lqchen}@nudt.edu.cn Mine@di.ens.fr {wdong, wj}@nudt.edu.cn

[*] State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China
[†] College of Computer Science, National University of Defense Technology, Changsha, China
[‡] Computer Science Lab, École Normale Supérieure, Paris, France

## ABSTRACT

Embedded software often involves intensive numerical computations and thus can contain a number of numerical runtime 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 technique that introduces concurrency in embedded systems. To this end, a numerical static analysis approach is desired for embedded software with interrupts. In this paper, we propose a sound numerical static analysis approach specifically for interrupt-driven programs based on sequentialization techniques. A 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. To be more clear, we first propose a sequentialization algorithm to sequentialize interrupt-driven programs into non-deterministic sequential programs according to the semantics of interrupts. On this basis, we leverage the power of numerical abstract interpretation to analyze numerical properties of the sequentialized programs. Moreover, to improve the analysis precision, 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.

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

## Keywords

Embedded Software, Interrupt-Driven Programs, Static Analysis, Abstract Interpretation, Sequentialization

.

## 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 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 executions and even unexpected erroneous behaviors. Therefore, there is a great need in practice to ensure that IDPs work correct 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) [2]. 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 ISR(){
2:   x++;
3:   y−−;
4:   return;
5: }
```

**Figure 1: A motivating example**

Fig. 1 shows a motivating example, where the functions *task*() and *ISR*() 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 $y$. $ISR()$ 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 fires 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 [15, 16], but very few approaches have considered the specific features of IDPs [6, 17]. 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. The main idea is as follows. We first sequentialize IDPs into non-deterministic sequential programs according to the semantics of interrupts and the interaction between tasks and interrupts. Sequentialization then enables the use of existing analysis and verification techniques for sequential programs to verify IDPs. After that we make use of numerical abstract interpretation to analyze the numerical properties of the sequentialized IDPs. Moreover, by considering the specific features of sequentialized interrupt-driven programs, we design specific abstract domains to improve the precision of numerical static analysis. 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.

## 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. In other words, higher-priority interrupts can interrupt lower-priority interrupts and tasks, but the opposite preemption can not happen. For the sake of presentation, 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.

$$
\begin{array}{lcl}
\mathit{Expr} & := & l \mid C \mid E_1 \diamond E_2 \text{ (where } l \in NV,\ C \text{ is a constant,} \\
 & & E_1, E_2 \in \mathit{Expr} \text{ and } \diamond \in \{+, -, \times, \div\}) \\
\mathit{Stmt} & := & l = g \mid g = l \mid l = e \mid S_1; S_2 \mid \textbf{skip} \mid \mathit{enableISR}(i) \\
 & & \mid \mathit{disableISR}(i) \mid \textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2 \\
 & & \mid \textbf{while } e \textbf{ do } S \\
 & & (\text{where } l \in NV, g \in SV, e \in \mathit{Expr}, i \in [1, N], \\
 & & S_1, S_2, S \in \mathit{Stmt}\ ) \\
\mathit{Task} & := & \mathit{entry} \text{ (where } \mathit{entry} \in \mathit{Stmt}) \\
\mathit{ISR} & := & \langle \mathit{entry}, p \rangle \text{ (where } \mathit{entry} \in \mathit{Stmt},\ p \in [1, N]) \\
\mathit{Prog} & := & \mathit{Task} \parallel \mathit{ISR}_1 \parallel \ldots \parallel \mathit{ISR}_N
\end{array}
$$

**Figure 2: Syntax of interrupt-driven programs**

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 assume that the priority $p$ is equal to the index of the corresponding interrupt and use $ISR_i$ to represent the interrupt with priority $p = i$ where $i \in [1, N]$. $enableISR(i)$ and $disableISR(i)$ represent respectively the instructions that enable and disable an interrupt by writing to the interrupt mask register (IMR). We use $SV$ and $NV$ respectively represent the set of shared and non-shared variables.

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 this form by introducing auxiliary variables. Moreover, we assume that the statements $g = l$ and $l = g$ are *atomic*.

In this paper, we consider analyzing IDPs in the level of source code rather than machine code, and one program statement in the source code can be translated into several machine instructions. Hence, an interrupt may happen 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 happens 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 happen during the running of this statement at machine instruction level. For example, the interrupt may happen after reading $x$ and before reading

$y$, and then the value of variable $z$ can be 1 in this case. This is the reason why we only allow two kinds of statements (i.e., $l = g$ and $g = l$) that are atomic to access a share 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 point to remove unnecessary scheduling in Sect. 3.3.

## 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 fired before any program statement of the task. If the $i$-th interrupt is fired, 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 fired. Similarly, before each program statement of the $i$-th interrupt, if the $j$-th interrupt with higher priority (i.e., satisfying $i<j$) is fired, $ISR_i$ 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 fired. 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. In other words, in IDPs, interrupt preemption can be modeled as merely a function call.

Based on this insight, inspired from [12] where Kidd et al. made use of a *Schedule*() function to simulate the priority preemptive scheduler for sequentializing multi-tasking programs, we add an explicit *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: $St' \stackrel{\text{def}}{=} Schedule(); St$. The *Schedule*() function works as follows: It passes through the interrupts of which the priority 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 happened. In this subsection, now we make the following assumption: Each statement in the task and ISRs is atomic and a higher-priority interrupt can happen 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 [12] 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 the *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), $ISRs\_seq[N]$ (whose indices range from 1 to $N$) represents the corresponding sequentialized version of a fixed set of ISRs and the function *nondet*() non-deterministically returns true or false. The function *task_seq* is the entrance of the sequentialized program. Besides, in this example, since we have only one interrupt, the *Schedule*() functions added in *ISR_seq*() can be all omitted and those *Schedule*() functions added in *task_seq*() can be all replaced as

**if**(*nondet*()) *ISR_seq*();

```
1:  int x, y, z;
2:  //Current priority
3:  int Prio = 0;
4:  //ISR entry
5:  ISR ISRs_seq[N];
6:  void task_seq(){
7:      int tx, ty;
8:      Schedule();
9:      tx = x;
10:     Schedule();
11:     ty = y;
12:     Schedule();
13:     if(tx < ty){
14:       Schedule();
15:       tx = x;
16:       Schedule();
17:       ty = y;
18:       Schedule();
19:       z = 1/(tx − ty);
20:     }
21:     Schedule();
22:     return;
23:   }
```

```
1:  void ISR_seq(){
2:    int tx, ty;
3:    Schedule(); tx = x;
4:    Schedule(); tx = tx + 1;
5:    Schedule(); x = tx;
6:    Schedule(); ty = y;
7:    Schedule(); ty = ty − 1;
8:    Schedule(); y = ty;
9:    Schedule(); return;
10:  }
11:  void Schedule(){
12:    //Save current priority
13:    int prevPrio = Prio;
14:    for(int i = 1; i ≤ N; i++){
15:      if(i ≤ Prio) continue;
16:      if(nondet()){
17:        Prio = i;
18:        ISRs_seq[i].entry();
19:      } }
20:    //Restore priority
21:    Prio = prevPrio;
22:  }
```

**Figure 3: Sequentialization of the motivating example by simulating priority preemptive scheduling**

## 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 unnecessary 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 variable, it makes no difference whether an interrupt happens before or after this statement. For example, suppose that the task is comprised 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, the following sequentialized program is still sound:

$$
\begin{array}{l}
St_1; St_2; \ldots; St_{n-1}; \\
for(int\ i = 1; i < n; i{+}{+}) \\
\quad Schedule(); \\
St_n;
\end{array}
$$

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. And loops can be analyzed very fast, for example, by using extrapolation techniques such as widening in abstract interpretation.

In fact, in practical IDPs, only a very small percentage of program statements will read/write 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 whose happening will affect the value of $g$. Similarly, after a program statement $g = l$ that writes a shared variable $g$, we only need to consider those ISRs whose execution 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 *Schedule*() function calls during sequentializing IDPs. To this end, we first introduce some notations.

**Data flow dependency among interrupts.** For each $ISR_i$, we introduce $RSVars(ISR_i)$ and $WSVars(ISR_i)$ to respectively denote the sets of shared variables that are read and written by $ISR_i$. Given two interrupts $ISR_i, ISR_j$, if $RSVars(ISR_i) \cap WSVars(ISR_j) \neq \emptyset$, we say that $ISR_i$ is *directly dependent* on $ISR_j$, denoted as $ISR_i \rightarrow ISR_j$. Given two interrupts $ISR_i, ISR_j$, we say that $ISR_i$ is *transitively dependent* on $ISR_j$, denoted as $ISR_i \twoheadrightarrow ISR_j$, if there exists $ISR_k$ such that $(ISR_i \rightarrow ISR_k \vee ISR_i \twoheadrightarrow ISR_k) \wedge (ISR_k \rightarrow ISR_j \vee ISR_k \twoheadrightarrow ISR_j)$. Given an interrupt $ISR_i$, we define a so-called *dependent interrupt group* for $ISR_i$ as $dGroup[ISR_i] \stackrel{\text{def}}{=} \{I \in ISRs \mid I \rightarrow ISR_i \vee I \twoheadrightarrow ISR_i\}$, and a so-called *influenced interrupt group* for $ISR_i$ as $iGroup[ISR_i] \stackrel{\text{def}}{=} \{I \in ISRs \mid ISR_i \rightarrow I \vee ISR_i \twoheadrightarrow I\}$.

*Example 1.* Suppose that in an IDP, there are two shared variables $x, y$, three interrupts $ISR_1, ISR_2, ISR_3$, and

$$RSVars(ISR_1) = WSVars(ISR_2) = \{x\}$$
$$RSVars(ISR_2) = WSVars(ISR_3) = \{y\}$$

Then, $dGroup[ISR_2] = \{ISR_3\}$ and $iGroup[ISR_2] = \{ISR_1\}$.

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 $ISR_i \rightarrow ISR_j$. 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

$$DG_{ij} \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } ISR_i \rightarrow ISR_j \\ 0 & \text{otherwise} \end{cases}$$

We use a procedure *BuildDepGraph*() to construct the dependency graph for an IDP. And we use two procedures *CompDepGroup*() and *CompInfGroup*() to compute respectively the dependent and influenced interrupt groups for $ISR_i$, as shown in Algorithm 1. Basically, Algorithm 1 computes the two groups by computing the transitive closure of direct dependency relations among ISRs.

**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 happens before or after this statement.

**Algorithm 1** Algorithms for computing dependent/influenced group for $ISR_i$

```
procedure CompDepGroup(i: int, p: int, imr: int)
    ws, rs: int set;
    ws = {i}, rs = {};
    j, k: int;
    while( ws ≠ ∅)
      //Get an element and remove it from a set
      k = GetAndRemove(ws);
      rs ← rs ∪ {k};
      for each (j ∈ [p, N] ∧ imr|_j == 1) do
        if (DG_kj == 1 && j ∉ rs) do
          ws ← ws ∪ {j};
    return rs;
end procedure
procedure CompInfGroup(i: int, p: int, imr: int)
    ws, rs: int set;
    ws = {i}, rs = {};
    j, k: int;
    while ws ≠ ∅ do
      k = GetAndRemove(ws);
      rs ← rs ∪ {k};
      for(each j ∈ [p, N] ∧ imr|_j = 1) do
        if (DG_jr == 1 && rs ∩ {j} == ∅) then
          ws ← ws ∪ {j};
    return rs;
end procedure
```

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 write into a shared variable, we need to consider the influence of this statement to those interrupts whose executions 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 *Schedule*() functions *before statements that read shared variables* and *after statements that write* shared variables. To be more clear, we give the details 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
  - $S_1 \stackrel{\text{def}}{=} \{I \in ISRs \mid g \in WSVars(I)\}$
  - $S_2 \stackrel{\text{def}}{=} \{I \in dGroup[I'] \mid I' \in S_1\}$

  where $S_1$ represents the set of ISRs that directly write 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 *ReadDepISRs*() to compute $S_1 \cup S_2$, as shown in Algorithm 2.

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

  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

**Algorithm 2** Algorithms for computing sets of ISRs that need to be considered w.r.t. a shared variable

```
//Compute the set of ISRs that need to be considered
before a statement that reads a shared variable g
procedure ReadDepISRs(g: vars, p: int, imr: int)
    directDepSet = {}: int set;
    depSet = {}: int set;
    i, j: int;
    for (each i ∈ [p, N] ∧ imr|_i == 1) do
      if (g ∈ WSVars(ISR_i)) then
        directDepSet ← directDepSet ∪ {i};
    for (each j ∈ directDepSet) do
      depSet ← depSet ∪ CompDepGroup(j, p, imr);
    return depSet;
end procedure
//Compute the set of ISRs that need to be considered
after a statement that writes a shared variable g
procedure WriteInfISRs(g: vars, p: int, imr: int)
    directInfSet = {}: int set;
    infSet = {}: int set;
    i, j: int;
    for (each i ∈ [p, N] ∧ imr|_i == 1) do
      if (g ∈ RSVars(ISR_i)) then
        directInfSet ← directInfSet ∪ {i};
    for(each j ∈ directInfSet) do
        infSet ← infSet ∪ CompInfGroup(j, p, imr);
    return infSet;
end procedure
```

in $S_3$. We use procedure *WriteInfISRs*() to compute $S_3 \cup S_4$, as shown in Algorithm 2.

On this basis, we introduce a new schedule function namely *ScheduleG*(*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 fired before the statement $St$. In fact, in practical IDPs, an interrupt never fires 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 $+\infty$.

In Fig. 4, we introduce a function namely *ScheduleG_K*(*group*) to call the *ScheduleG*() function $K$ times. In case where $K$ can not be specified, we can put $K$ to $+\infty$ (and then the loop in *ScheduleG_K*() 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 functions namely *InvokeBefore*($St$, *group*) and *InvokeAfter*($St$, *group*) to insert the *ScheduleG_K*(*group*) 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 *InvokeBefore*($St^R$, *group*) to obtain the following sequentialized result:

$$St'^R \stackrel{\text{def}}{=} \mathit{ScheduleG\_K}(group); St^R;$$

For a statement $St^W$ that writes a shared variable, we will use *InvokeAfter*($St^W$, *group*) to obtain the following sequentialized result:

$$St'^W \stackrel{\text{def}}{=} St^W; \mathit{ScheduleG\_K}(group);$$

```
1:  void ScheduleG(group: int set){
2:    //Save current priority
3:    int prevPrio = Prio;
4:    for(int i = 1; i ≤ N; i++){
5:      if( i ≤ Prio ∥ i ∉ group ) continue;
6:      if(nondet()){
7:        Prio = i;
8:        ISRs_seq[i].entry(group);
9:    } }
10:   //Restore priority
11:   Prio = prevPrio;
12: }
13: //schedule K times
14: void ScheduleG_K(group: int set){
15:     for(int i = 1; i ≤ K; i++)
16:         ScheduleG(group);
17: }
```

**Figure 4: Scheduling functions calling only those ISRs in a given interrupt group**

**Simplifying the sequentialized programs.** We may notice that the sequentialization method based on the above strategy may still introduce unnecessary invoking of the function *ScheduleG_K*(). For example, suppose the task is comprised of $St_1^W; \ldots; St_n^R$, where $St_1^W$ represents a statement that write a shared variable $g_1$ and $St_1^R$ represents a statement that read a shared variable $g_2$, while all statements in between do not access any shared variables. Then the sequentialized task will be:

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

When the interrupt groups $grp_1$ and $grp_2$ are the same, the invoking of *ScheduleG_K*($grp_2$) is unnecessary.

Based on this insight, we design a simplification procedure *Simplify*() to remove unnecessary invoking of *ScheduleG_K*(). The *Simplify*() procedure removes the second invoking of *ScheduleG_K*($grp_1$) in the following two patterns:

- $St_1^W; \mathit{ScheduleG\_K}(grp_1); \ldots; \mathit{ScheduleG\_K}(grp_1); St_n^R;$
- $\mathit{ScheduleG\_K}(grp_1); St_1^R; \ldots; \mathit{ScheduleG\_K}(grp_1); St_n^R;$

where for all $i \in [2, n-1]$ the statement $St_i$ does not write 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 can 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: $ISR_i \rightarrow ISR_j \wedge ISR_j \rightarrow ISR_k \wedge ISR_i \nrightarrow ISR_k$. 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. As shown in Algorithm 1 and Algorithm 2, in procedures *CompDepGroup*(), *CompInfGroup*(), *ReadDepISRs*(), *WriteInfISRs*(), we have considered the value of IMR. Essentially, these procedures consider data flow dependency among enabled interrupts. In these two algorithms, we use $imr|_j$ to represent the $j$-th bit of the $imr$ value which indicates whether $ISR_j$ is enabled (when the bit is 1) or disabled (when the bit is 0).

**Pre-analysis for analyzing the value of IMR.** In order to obtain the value of IMR at each program point, we need a pre-analysis to analyze the value of IMR (for each entry function of the task and interrupts separately) before sequentializing an IDP. In other words, we need a pre-analysis for analyzing the value of IMR without considering the interleaving between tasks and interrupts. However, inside an ISR function, programmers may also call *disableISR*($i$) and *enableISR*($i$) to change the value of IMR. Hence, the value of IMR at the entry-point of an ISR may be different from the value at the exit-point of the same ISR. Thus, the pre-analysis that analyzes the value of IMR for each entry function of the task and interrupts separately may be not sound in general. In this paper, we essentially model interrupt preemption as a function call to the corresponding ISR. To this end, throughout this paper, we presume the following assumption:

- 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.

In fact, this assumption is followed in most practical IDPs.

Based on this assumption, 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 we assume the interrupt is enabled in this case. Essentially, 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 *enableISR*($i$) and *disableISR*($i$).** Until now, we have considered adding calls to the schedule function only before statements that read shared variables and after statements that write into shared variables. However, when if an IDP includes statements *enableISR*($i$) and *disableISR*($i$), the above strategy may miss some invoking of related ISRs. For example, Fig. 5 shows an IDP involving statements *enableISR*($i$) and *disableISR*($i$), wherein $y$ is the shared variable. If we add invocation of ISRs only before statements that read shared variables and after statements that write 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* fires between line 5 and line 6 in the *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: }
```

**Figure 5: An example with *disableISR* and *enableISR***

Hence, when dealing with statements *enableISR*($i$) and *disableISR*($i$), we may also need to add an invocation of certain related ISRs. We use the following strategy to add invocation of certain related ISRs during dealing with statements *enableISR*($i$) and *disableISR*($i$).

When dealing with *disableISR*($i$), we presume that all shared variables in $WSVars(ISR_i)$ are read during the execution of the statement *disableISR*($i$). In this situation, we need add the schedule function to invoke $ISR_i$ and all those interrupts in $dGroup[ISR_i]$ before *disableISR*($i$). Similarly, when dealing with *enableISR*($i$), we presume that all shared variables in $RSVars(ISR_i)$ are written during the execution of the statement *enableISR*($i$). In this situation, we need add the schedule function to invoke $ISR_i$ and all those interrupts in $iGroup[ISR_i]$ after *enableISR*($i$).

Algorithm 3 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 *computeIMR*(). *InvokeBefore*() and *InvokeAfter*() represent a call to the corresponding interrupts before or after a statement.

## 3.4 The overall sequentialization algorithm considering data flow dependency and IMR

Algorithm 4 shows the overall sequentialization algorithm considering both the data flow dependency and IMR. In Algorithm 4, 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.

For an IDP consisting of one task and $N$ interrupts, the overall sequentialization algorithm shown in Algorithm 4 works as follows: First, we compute the value of IMR at each program point and build the data flow dependency graph

**Algorithm 3** Algorithm calling *ISRs* for each statement

```
procedure StmtInvokeISRs(st : stmt, p : int)
    imr : int;
    group : int set;
    match st with
    | l = g →
      imr ← IMRValTbl.find(st);
      group ← ReadDepISRs(g, p, imr);
      InvokeBefore(st, group);
    | g = l →
      imr ← IMRValTbl.find(st);
      group ← WriteInfISRs(g, p, imr);
      InvokeAfter(st, group);
    | disableISR(i) →
      imr ← IMRValTbl.find(st);
      group ← CompDepGroup(i, p, imr);
      InvokeBefore(st, group);
    | enableISR(i) →
      imr ← IMRValTbl.find(st);
      group ← CompInfGroup(i, p, imr);
      InvokeAfter(st, group);
    | S1; S2
    | if e then S1 else S2 →
      StmtInvokeISRs(S1, p); StmtInvokeISRs(S2, p);
    | while e do S →
      StmtInvokeISRs(S, p);
    | _ → ();
end procedure
```

among ISRs. Then, we sequentialize the task and each interrupt separately. For each program statement in the task and ISRs, we first get the value of IMR from *IMRValTbl*, compute the dependent/influenced interrupt group, and then add the schedule function to invoke ISRs in the corresponding group before or after that statement. Finally, we use the procedure *Simplify*() to remove certain unnecessary calls to ISRs in the sequentialized IDPs.

# 4. ANALYZING SEQUENTIALIZED IDPS VIA ABSTRACT INTERPRETATION

As we mentioned before, embedded software often involves lots of numerical computations and thus have the potential to contain numerical related program errors. To this end, in this section, we make use of abstract interpretation [8] to analyze numerical properties of IDPs. To be more clear, we would like to leverage existing numerical abstract interpretation techniques for sequential programs to analyze numerical properties of sequentialized IDPs given by the methods described in Sect. 3.

## 4.1 Analyzing entry functions of sequentialized IDPs

The resulting sequentialized IDPs given by the methods described in Sect. 3 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. Hence, we need to analyze the entry function of the sequentialized task first. In most cases, the sequentialized task will invoke all the sequentialized ISRs. For the sequentialized ISR that is invoked in the sequentialized task, we do not need

**Algorithm 4** 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 ← ComputeIMR();
      BuildDepGraph();
      SeqEach(task);
      for (i = 1 to N) do
        SeqEach(ISRs[i − 1], i);
      Simplify();
  end procedure
  //Sequentialize task and each ISR
  procedure SeqEach(fn : stmt list, p : int)
      for( each st_i ∈ fn) do
        StmtInvokeISRs(st_i, p);
  end procedure
```

to analyze the entry function of this ISR again after analyzing the task. However, there may exist special cases where an interrupt $ISR_i$ may never be invoked in the sequentialized result of the *task* function by the algorithm described in Sect. 3.4. 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 analyzing the sequentialized IDPs, we need to analyze not only the entry function of the sequentialized *task*, but also the entry functions of those sequentialized *ISRs* that are never invoked in the sequentialized *task*.

*Example 2.* Suppose that an IDP is comprised 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*. However, in this IDP, there will be a division-by-zero in $ISR_1$ when $ISR_2$ fires before $ISR_1$. Hence, when performing static analysis over sequentialized IDPs, we need to analyze not only the entry function of sequentialized *task* but also the entry function of sequentialized $ISR_1$. If we analyze the entry function of sequentialized $ISR_1$, the division-by-zero error will be detected, since in the sequentialized $ISR_1$ a non- deterministic call to the higher-priority $ISR_2$ is added before the statement $\{tmp1 = x;\}$.

Hence, given a sequentialized IDP, we first perform numerical static analysis to analyze the entry function of the sequentialized task. And then we separately analyze entry functions of those sequentialized *ISRs* that are never invoked in the sequentialized *task*.

## 4.2 Specific abstract domains for IDPs

To perform numerical static analysis, there exist a variety of numerical abstract domains in the literature. For

example, the interval abstract domain [7] 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 [14] 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 are not common in generic programs. To improve the precision of numerical static analysis of sequentialized IDPs, we need to design certain specific abstract domains according to the specific features of IDPs. In the following, we give an example of specific abstract domains for IDPs.

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 happened 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. Consider analyzing $\{\textbf{if}(notdet())\ ISRi();\}$. Let $A_1$ denote the abstract value in an abstract domain before this statement. After this statement, abstract interpretation will perform a join operation to compute the post abstract value as $A_1 \sqcup A_2$ where $A_2 \stackrel{\text{def}}{=} [\![ISRi()]\!]^\sharp(A_1)$ wherein $[\![ISRi()]\!]^\sharp$ denotes the abstract transfer function of $ISRi()$. Intuitively, in $A_1 \sqcup A_2$, $A_1$ denotes the abstract value when $ISR_i$ has never been fired while $A_2$ denotes the abstract value after $ISR_i$ has been fired. However, after the join operation, most numerical abstract domains will lose the information that the abstract values are different for the case where $ISR_i$ has been fired or not.

Our basic idea is to use a boolean flag variable $f$ in the abstract domain to distinguish whether an at-most-once fired periodic interrupt $ISR_i$ has already been fired or not. In the abstract domain, for each boolean flag variable, we maintain a pair of abstract values $(A^f, A^{\neg f})$ where $A^f$ denotes the abstract value when $ISR_i$ has already been fired and $A^{\neg f}$ denotes the abstract value when $ISR_i$ has not been fired. As an example, we now use the boolean flag abstract domain to analyze $\{\textbf{if}(notdet())\ ISRi();\}$. Let $(A_1^f, A_1^{\neg f})$ denote the abstract value in the boolean flag abstract domain before this statement. After the **then** branch, we get the post abstract value $([\![ISRi()]\!]^\sharp(A_1^{\neg f}), \bot)$. After the **else** branch, nothing changes and thus we get the pair $(A_1^f, A_1^{\neg f})$. Then, at the control-flow join, we perform a join operation to compute the post abstract value as

$$([\![ISRi()]\!]^\sharp(A_1^{\neg f}), \bot) \sqcup (A_1^f, A_1^{\neg f})$$

and then by element-wise join, it will result in

$$(A_1^f \sqcup [\![ISRi()]\!]^\sharp(A_1^{\neg f}),\ A_1^{\neg f})$$

*Example 3.* Suppose that an IDP is comprised 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 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.

# 5. IMPLEMENTATION AND EXPERIMENT RESULTS

We have implemented a prototype tool to sequentialize IDPs, which uses CIL [18] as its front-end. We use a CIL supported inline tool to deal with function calls. We have also developed a numerical static analyzer for analyzing sequentialized IDPs based on the front-end CIL and the Apron [11] numerical abstract domain library.

Our experiments were conducted on a selection of benchmark examples listed in Fig. 6. *Motv_Ex* is the motivating example shown in Fig. 1. *DataRace_Ex* and *Privatize* come from a data race detection tool Goblint [21]. *Nxt_gs* is a robot controller program from LEGO company samples[1]. *UART* (Universe Asynchronous Receiver and Transmitter)[2] is from an open source website which implements a First-In First-Out (FIFO) buffer. *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). Some of these examples originally do not include interrupts, such as *Nxt_gs*, but tasks in these programs are scheduled by the priorities of tasks, which is quite similar to IDPs. Thus we adapt them into IDPs.

Fig. 6 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 [12]. *DF_SEQ* represents the sequentialization method described in Sect. 3.4 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 *UART*, the code size of the program given by *DF_SEQ* is around 20% of the code size of that given by *SEQ*.

Fig. 7 shows the analysis results of analyzing sequentialized IDPs by numerical abstract interpretation. We use *box* and *octagon* abstract domains to analyze the sequentialized IDPs. For all the examples in Fig. 7, using the box domain and the octagon domain for both the programs given by *SEQ* and *DF_SEQ* can find the same properties or errors in the program. For *Motv_Ex*, we find the expected division-by-zero error. For *DataRace_Ex* and *Privatize*, our method can prove their assertions. For example, *Privatize* asserts that a shared variable is always equal to 1. For

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

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

| Program | | | | | Sequentialization | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Name | OLT | OLI | #Vars | #ISR | SEQ | | DF_SEQ | | DF_SEQ/SEQ (%LOC) |
| | | | | | LOC | Time (s) | LOC | Time (s) | |
| 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 |
| Ping_pong | 130 | 53 | 21 | 1 | 3159 | 0.006 | 842 | 0.006 | 26.65 |

**Figure 6: Experimental results on Sequentializing IDPs**

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

Fig. 8 shows the analysis results of analyzing IDPs with at-most-once fired periodic interrupts. In Fig. 8, we use BF to denote the boolean flag abstract domain described in Sect. 4. #FP denote the number of false alarms. *Example_3* is an adapted version of the program in Example 3 in Section 4.2 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_3* 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.

# 6. RELATED WORK

**Sequentialization.** Much work has been done on sequentializing concurrent programs. Qadeer et al. [19] propose a context bounded analysis (CBA) method for concurrent programs via sequentialization. Their sequentialization method employs a non-deterministic scheduler model for two threads and two context switches. Lal et al. [13] propose a CBA method based on sequentialization with an arbitrary given context bound. Inverso et al. [10] propose a lazy sequentialization method that reduces the nondeterminism of sequentialized programs to avoid exponentially growing formula sizes during model checking the sequentialized programs. Recently, Chaki et al. [5] present a CBA method for analyzing periodic programs based on sequentialization.

Kidd et al. [12] 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 model preemptions by function calls. Edwards [9] 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 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. Astree [2] is one of the famous numerical static analyzers for sequential programs, which has been successfully used in analyzing flight control software.

Miné [15, 16] proposes a numerical static analysis method for parallel embedded software. The main idea of his method is to iterate each thread in turn until all thread interferences stabilize. Compared with his work, our work is specially for IDPs and we can get more precise analysis results for IDPs.

Cooprider et al. [6] propose a static analysis method for embedded software to reduce the code size. Beckschulze et al. [1] 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 [17] proposes a static analysis method for a concurrent USB driver, which is limited to two threads. Compared with his method, our method supports multiple tasks and interrupts.

**Analysis of interrupt-driven programs.** In the literature, there are a few work on analyzing and verifying IDPs [3, 4, 20, 21]. Brylow et al. [3] propose a static analysis method for interrupt-driven Z86-based software. Their method uses model checking to analyze upper bounds of stack size and interrupt latencies of IDPs. Most of the existing work focus on object code and consider problems such as interrupt latency, stack size, and data race.

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

| Program | Analysis time of SEQ (s) | | Analysis time of DF_SEQ (s) | | Founded 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 | integer overflow alarms |
| UART | 0.732 | 5.782 | 0.128 | 1.177 | no array-out-of-bound |
| Ping_pong | 0.429 | 2.434 | 0.054 | 0.251 | no array-out-of-bound |

**Figure 7: Experimental results on analyzing sequentialized IDPs**

| Program | | | | Analysis of SEQ (s) | | | | Analysis of DF_SEQ (s) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | OLT | OLI | SL | OCT | | BF+OCT | | DF_SL | OCT | | BF+OCT | |
| | | | | time (s) | #FP | time (s) | #FP | | time (s) | #FP | time (s) | #FP |
| Example_3 | 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 |

**Figure 8: Experimental results on analyzing IDPs with at-most-once fired periodic interrupts**

properties of the sequentialized IDPs. By considering specific features of sequentialized IDPs, we design and make use of specific abstract domains to analyze 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 also plan to handle shared variables involving pointers during sequentialization.

**Acknowledgments.** This work is supported by the 973 Program under Grant No. 2014CB340703, the NSFC under Grant Nos. 61120106006, 61202120, 91318301.

# 8. REFERENCES

[1] E. Beckschulze, S. Biallas, and S. Kowalewski. Static analysis of lockless microcontroller C programs. In *SSV'12*, pages 103–114, 2012.

[2] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI'03*, pages 196–207. ACM, 2003.

[3] D. Brylow, N. Damgaard, and J. Palsberg. Static checking of interrupt-driven software. In *ICSE'01*, pages 47–56. IEEE, 2001.

[4] D. Brylow and J. Palsberg. Deadline analysis of interrupt-driven software. *IEEE Trans. Software Eng.*, 30(10):634–655, 2004.

[5] S. Chaki, A. Gurfinkel, and O. Strichman. Verifying periodic programs with priority inheritance locks. In *FMCAD'13*, pages 137–144, 2013.

[6] N. Cooprider and J. Regehr. Pluggable abstract domains for analyzing embedded software. In *LCTES'06*, pages 44–53. ACM, 2006.

[7] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. of the 2nd International Symposium on Programming*, pages 106–130. Dunod, Paris, 1976.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM, 1977.

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

[10] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. In *CAV'14*, volume 8559 of *LNCS*, pages 585–602. Springer, 2014.

[11] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV'09*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.

[12] N. Kidd, S. Jagannathan, and J. Vitek. One stack to run them all - reducing concurrent analysis to sequential analysis under priority scheduling. In *SPIN'10*, volume 6349 of *LNCS*, pages 245–261. Springer, 2010.

[13] A. Lal, T. Touili, N. Kidd, and T. W. Reps. Interprocedural analysis of concurrent programs under a context bound. In *TACAS'08*, volume 4963 of *LNCS*, pages 282–298. Springer, 2008.

[14] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[15] A. Miné. Static analysis of run-time errors in embedded critical parallel C programs. In *ESOP'11*, volume 6602 of *LNCS*, pages 398–418. Springer, 2011.

[16] A. Miné. Relational thread-modular static value analysis by abstract interpretation. In *VMCAI'14*, volume 8318 of *LNCS*, pages 39–58. Springer, 2014.

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

[18] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: intermediate language and tools for analysis and transformation of C programs. In *CC'02*, volume 2304 of *LNCS*, pages 213–228. Springer, 2002.

[19] S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *PLDI'04*, pages 14–24. ACM, 2004.

[20] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. *ACM Trans. Embedded Comput. Syst.*, 4(4):751–778, 2005.

[21] M. D. Schwarz, H. Seidl, V. Vojdani, P. Lammich, and M. Müller-Olm. Static analysis of interrupt-driven programs synchronized via the priority ceiling protocol. In *POPL'11*, pages 93–104. ACM, 2011.