Numerical Static Analysis of Embedded Software with Interrupts

Liqian Chen
National University of Defense Technology, Changsha, China
11/09/2016

(Joint work with Xueguang Wu, Wei Dong, Ji Wang from NUDT, and Antoine Miné from UPMC)
Overview

• Motivation
• Interrupt-driven programs (IDPs)
• Sequentialization of IDPs
• Analysis of sequentialized IDPs via abstract interpretation
• Experiments
• Conclusion
Interrupts in Embedded Software

- **Interrupts** are a commonly used technique that introduce **concurrency** in embedded software.

- Data race detection
  - too many false alarms
  - harmful or unhelpful

- Numerical static analysis to find run-time errors
  - embedded software often contain intensive **numerical** computations which are **error prone**
Motivation

• Without considering the interleaving, sequential program analysis results may be **unsound**

```c
int x, y, z;
void TASK(){
    if(x<y){           // 1
        z = 1/(x-y);  // 2
    }           
    return;         
}
```

```c
void ISR(){
    x++;            
    y--;            
    return;        
}
```

**Interrupt semantics:**
Given \(x=1, y=3\), if ISR fires at \(\boxed{1}\), there is a division-by-zero error at \(\boxed{2}\)
Our Goal

• Goal
  • a sound approach for numerical static analysis of embedded C programs with interrupts

• Challenges
  • analyzing source code rather than machine code (soundness)
  • interleaving state space can grow exponentially w.r.t. the number of interrupts (scalability)
  • interrupts are controlled by hardware (precision)
    • e.g., periodic interrupts, interrupt mask register (IMR)
Basic Idea

IDPs → Seq → Sequential Programs

Numerical static analysis via abstract interpretation
Overview

- Motivation
- Interrupt-driven programs (IDPs)
- Sequentialization of IDPs
- Analysis of sequentialized IDPs via abstract interpretation
- Experiments
- Conclusion
Interrupt-Driven Programs

• Our target interrupt-driven programs (IDPs)
  • an IDP consists of a fixed finite set of tasks and interrupts
  • tasks are scheduled cooperatively, while interrupts are scheduled preemptively by priority (on a uncore processor)
Interrupt-Driven Programs

• Model of interrupt-driven programs
  • 1 task + N interrupts
    • each interrupt priority with at most one interrupt
  • only 2 forms of statements accessing shared variables
    • $l=g$ //read from a shared variable $g$
    • $g=l$ //write to a shared variable $g$

| Expr | := | $l \mid C \mid E_1 \diamond E_2$ (where $l \in NV$, $C$ is a constant, $E_1, E_2 \in Expr$ and $\diamond \in \{+, -, \times, \div\}$) |
| Stmt | := | $l=g \mid g=l \mid l=e \mid S_1; S_2 \mid \text{skip} \mid \text{enableISR}(i)$ |
  | | $\mid \text{disableISR}(i) \mid \text{if } e \text{ then } S_1 \text{ else } S_2$ |
  | | $\mid \text{while } e \text{ do } S$ |
  | | (where $l \in NV$, $g \in SV$, $e \in Expr$, $i \in [1, N]$, $S_1, S_2, S \in Stmt$) |
| Task | := | entry (where entry $\in Stmt$) |
| ISR | := | $\langle\text{entry}, p\rangle$ (where entry $\in Stmt$, $p \in [1, N]$) |
| Prog | := | Task $\parallel ISR_1 \parallel \ldots \parallel ISR_N$ |
Interrupt-Driven Programs

- Model of interrupt-driven programs
  - 1 task + N interrupts
    - each interrupt priority with at most one interrupt
  - only 2 forms of statements accessing shared variables
    - \( l=g \) //read from a shared variable \( g \)
    - \( g=l \) //write to a shared variable \( g \)

expr \( := l \mid C \mid E_1 \diamond E_2 \) (where \( l \in NV, C \) is a constant, \( E_1, E_2 \in Expr \) and \( \diamond \in \{+, -, \times, \div\} \))

stmt \( := l = g \mid g = l \mid l = e \mid S_1; S_2 \mid \text{skip} \mid \text{enableISR}(i) \)
  \mid \text{disableISR}(i) \mid \text{if } e \text{ then } S_1 \text{ else } S_2
  \mid \text{while } e \text{ do } S
  \) (where \( l \in NV, g \in SV, e \in Expr, i \in [1, N],\)
  \( S_1, S_2, S \in Stmt \))

task \( := \text{entry} \) (where \( \text{entry} \in Stmt \))

This model simplifies IDPs without losing generality
Interrupt-Driven Programs

• Assumptions over the model

1. all accesses to shared variables \((l=g\) and \(g=l)\) are **atomic**.

   this assumption exists in most of concurrent program analysis, e.g., Cseq [ASE’13], AstréeA[ESOP’11], KISS [PLDI’04]

2. the IMR is **intact** inside an ISR, i.e. \(IMR_{ISR_i^{entry}} = IMR_{ISR_i^{exit}}\)

   keeping IMR intact holds for practical IDPs, e.g., satellite control programs
Overview

• Motivation
• Interrupt-driven programs (IDPs)

• Sequentialization of IDPs

• Analysis of sequentialized IDPs via abstract interpretation

• Experiments

• Conclusion
Basic Idea of Sequentialization

• **Observation**: firing of interrupts can be simulated by function calls

• **Basic idea**: add a `schedule()` function before each (atomic) program statement of the task and interrupts
  • the `schedule()` function non-deterministically schedules higher priority interrupts

Original

\[ st_1; \ldots; st_k \]

Sequentialized

\[ st_1'; \ldots; st_k' \]

where \( st_i' = \text{schedule}(); \ st_i \)
Example

```
int x, y, z;
void task() {
  if (x < y) {
    z = 1 / (x - y);
  }
  return;
}
void ISR() {
  x++;
  y--;
  return;
}
```
Example

```c
int x, y, z;
void task(){
    if(x<y){
        z = 1/(x-y);
    }
    return;
}
void ISR(){
    x++;
    y--;
    return;
}
int Prio=0;
ISR ISRs_seq[N];
//current priority
//ISR entry
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 schedule(){
    int prevPrio = Prio;
    for(int i<=1;i<=N;i++){
        if(i<=Prio) continue;
        if(nondet()){  
            Prio = i;
            ISRs_seq[i].entry();
        }
    }
    Prio = prevPrio;
    }
```
Example

```c
int x, y, z;
void task(){
  if(x<y){
    z = 1/(x-y);
  }
  return;
}
void ISR(){
  x++;
  y--;
  return;
}
```

```c
void task_seq(){
  int tx, ty;
  schedule(); tx = x;
  schedule(); ty = y;
  schedule();
  if(tx < ty){
    schedule(); tx = x;
    schedule(); ty = y;
    schedule();
    return;
  }
}
```

```c
void ISR_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;
}
```

```c
void schedule(){
  int prevPrio = Prio;
  for(int i<=1;i<=N;i++){
    if(i<=Prio) continue;
    if(nondet()){
      Prio = i;
      ISRs_seq[i].entry();
    }
  }
  Prio = prevPrio;
}
```
Basic Idea of Sequentialization

- **Disadvantages** of the basic sequentialization method
  - the resulting sequentialized program becomes large
    - too many `schedule()` functions are invoked
- **Further observation**
  - interrupts and tasks communicate with each other by shared variables
    - interrupts only affect those statements which access shared variables

**Further idea:** utilize data flow dependency to reduce the size of sequentialized programs
Sequentialization by Considering Data Flow Dependency

- Example: Program \{ St_1; St_2; \ldots; St_n \}, where only \( St_n \) reads shared variables (SVs)
  - basic sequentialization

\[
\{ \text{schedule(); } St_1; \text{schedule(); } St_2; \ldots; \text{schedule(); } St_n \}
\]

- consider SVs

\[
\{ St_1; St_2; \ldots; St_{n-1}; \\
\quad \text{for(int } i=0; i<n; i++) \\
\quad \text{ schedule();} \\
\quad St_n \\
\}
\]
Sequentialization by Considering Data Flow Dependency

• Key idea: schedule relevant interrupts only for those statements accessing shared variables
  • before \( l = g \) (i.e., reading a shared variable)
    • schedule those interrupts which may affect the value of shared variable \( g \)
  • after \( g = l \) (i.e., writing a shared variable)
    • schedule those interrupts of which the execution results may be affected by shared variable \( g \)
Sequentialization by Considering Data Flow Dependency

• Need to consider the firing number of interrupts, otherwise the analysis results may be not sound

```c
void scheduleG_K(group: int set){
    for(int i=1;i<=K;i++)
        scheduleG(group);
}
```

K is the upper bound of the firing times of each ISR, which can be a specific value or $+\infty$. 
int x, y, z;
void task(){
    int t, tx, ty, tz;
    x = 10;
    y = 0;
    tx = x;
    ty = y;
    t = tx+ty;
    ty = y;
    tx = t-ty;
    x = tx;
    tz = t*2;
    z = tz;
    ty = y;
    ty = t-ty;
    y = ty;
}

void ISR1(){
    int tx, ty;
    ty = y;
    ty = ty + 1; y = ty;
    tx = x;
    tx = tx - 1; x = tx;
}

void ISR2(){
    int tz;
    tz = z;
    tz = tz + 1; z = tz;
}
int x,y,z;
void task(){
    int t, tx, ty, tz;
    x = 10; //scheduleG_K({1});
    y = 0; //scheduleG_K({1});
    tx = x;
    ty = y;
    t = tx+ty;
    ty = y;
    tx = t-ty;
    x = tx;
    scheduleG_K({1});
    tz = t*2;
    z = tz;
    ty = y;
    ty = t-ty;
    y = ty;
}
void ISR1(){
    int tx, ty;
    ty = y; ty = ty + 1; y = ty;
    tx = x; tx = tx -1; x = tx;
} void ISR2(){
    int tz;
    tz = z; tz = tz+1; z=tz;
}
```c
int x, y, z;
void task(){
    int t, tx, ty, tz;
    x = 10;
    y = 0;
    tx = x;
    ty = y;
    t = tx + ty;
    ty = y;
    tx = t - ty;
    x = tx;
    tz = t * 2;
    z = tz;
    ty = y;
    ty = t - ty;
    y = ty;
}
void ISR1(){
    int tx, ty;
    ty = y;
    ty = ty + 1;
    y = ty;
    tx = x;
    tx = tx - 1;
    x = tx;
}
void ISR2(){
    int tz;
    tz = z;
    tz = tz + 1;
}
}
```

only invoke relevant ISRs

```c
void task(){
    int t, tx, ty, tz;
    x = 10; scheduleG_K({1});
    y = 0; scheduleG_K({1});
    tx = x; ty = y;
    t = tx + ty;
    ty = y;
    tx = t - ty;
    x = tx;
    tz = t * 2;
    z = tz; scheduleG_K({2});
    scheduleG_K({1});
    ty = y;
    ty = t - ty;
    y = ty; scheduleG_K({1});
}
void ISR1_seq(){ //Same as ISR1
}
void ISR2_seq(){ //Same as ISR2
    //scheduleG_K({1}) gives:
    for(int i=0; i<K; i++)
        if(nondet()) ISR1_seq();
    //scheduleG_K({2}) gives:
    for(int i=0; i<K; i++)
        if(nondet()) ISR2_seq();
```

only invoke relevant ISRs

```c
```
Overview

• Motivation
• Interrupt-driven programs (IDPs)
• Sequentialization of IDPs
  • Analysis of sequentialized IDPs via abstract interpretation
• Experiments
• Conclusion
Analysis of Sequentialized IDPs via Abstract Interpretation

IDPs \rightarrow Seq \rightarrow Sequential Programs

Numerical static analysis via abstract interpretation
Analysis of Sequentialized IDPs via Abstract Interpretation

• Analysis of sequentialized IDPs
  • using generic numerical abstract domains

• Need to consider specific features of sequentialized IDPs
  • firing number of interrupts affects the analysis result
  • interrupts with period

Need specific abstract domains to consider interrupt features
Specific Abstract Domains for IDPs

• At-most-once firing periodic interrupts
  • periodic interrupts: firing with a fixed time interval
  • the period of interrupts is larger than one task period
• An abstract domain for at-most-once firing periodic interrupts
  • use boolean flag variables to distinguish whether ISRs have happened or not
Specific Abstract Domain for IDPs

- Example of boolean flag abstract domain

```c
int x;
void task(){
    int tx,z;
    x=0;
    tx=x;
    tx=tx+1;
    x=tx;
    z=1/(x-5);
}
void ISR1(){
    int tx,z;
    x=0;
    tx = x;
    tx = tx+10;
    x = tx;
    tx=tx+1;
    x=tx;
    if(*) ISR1();
    tx=x;
    if(*) ISR1();
    z=1/(x-5);
    // division is safe
}

If only using interval domain: x ∈ [1,21] and there will be a division by zero false alarm
```
Specific Abstract Domains for IDPs

• An abstract domain for tracing **syntactic equalities** in transformed IDPs
  
  • IDPs after transformation allow only 2 forms of statements accessing shared variables

```c
unsigned int thetaE; //shared variable
if(thetaE>0x1FF){
    thetaE = thetaE - 0x1FF;
    ......
}
```

```c
unsigned int thetaE; //shared variable
tmpthetaE = thetaE;
if(tmpthetaE>0x1FF){
    tmpthetaE = thetaE;
    tmpthetaE = tmpthetaE - 0x1FF;
    thetaE = tmpthetaE;
    ......
}
```

original program fragment

transformed program fragment
Overview

• Motivation

• Interrupt-driven programs (IDPs)

• Sequentialization of IDPs

• Analysis of sequentialized IDPs via abstract interpretation

• Experiments

• Conclusion
Experiments

• Benchmarks
  • control program used in the autonomous robot platform
  • universal asynchronous receive and transmitter (UART)
  • Heart Beat Monitor (HBM) for a micro-controller
  • some programs from industry
Experiments

• Experiment of sequentialization

<table>
<thead>
<tr>
<th>Program</th>
<th>Loc__task</th>
<th>Loc__ISR</th>
<th>#Vars</th>
<th>#ISR</th>
<th>SEQ LOC</th>
<th>SEQ Time (s)</th>
<th>DF_SEQ LOC</th>
<th>DF_SEQ Time(s)</th>
<th>DF_SEQ /SEQ (%LOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>iRobot3</td>
<td>114</td>
<td>80</td>
<td>55</td>
<td>1</td>
<td>2986</td>
<td>0.035</td>
<td>793</td>
<td>0.034</td>
<td>26.56</td>
</tr>
<tr>
<td>UART</td>
<td>129</td>
<td>15</td>
<td>47</td>
<td>1</td>
<td>5940</td>
<td>0.010</td>
<td>1215</td>
<td>0.010</td>
<td>20.45</td>
</tr>
<tr>
<td>HBM</td>
<td>500</td>
<td>85</td>
<td>36</td>
<td>2</td>
<td>9832</td>
<td>0.056</td>
<td>1312</td>
<td>0.053</td>
<td>13.34</td>
</tr>
<tr>
<td>PingPong</td>
<td>130</td>
<td>53</td>
<td>21</td>
<td>1</td>
<td>3159</td>
<td>0.006</td>
<td>842</td>
<td>0.006</td>
<td>26.65</td>
</tr>
<tr>
<td>ADC</td>
<td>1870</td>
<td>2989</td>
<td>312</td>
<td>1</td>
<td>123K</td>
<td>0.449</td>
<td>23K</td>
<td>0.8</td>
<td>18.70</td>
</tr>
<tr>
<td>S_Control</td>
<td>33885</td>
<td>1227</td>
<td>1352</td>
<td>1</td>
<td>10M</td>
<td>16.1</td>
<td>534K</td>
<td>1.6</td>
<td>5.34</td>
</tr>
</tbody>
</table>

The scale of sequentialized program by DF_SEQ is smaller than SEQ
Experiments

- Experiment of numerical static analysis

<table>
<thead>
<tr>
<th>Program</th>
<th>Analysis of SEQ (s)</th>
<th>Analysis of DF_SEQ(s)</th>
<th>Warnings &amp; Proved Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>BOX</td>
<td>BOX</td>
<td></td>
</tr>
<tr>
<td>iRobot3</td>
<td>0.303</td>
<td>0.069</td>
<td>2 int overflow alarms</td>
</tr>
<tr>
<td>UART</td>
<td>0.732</td>
<td>0.128</td>
<td>No ArrayOutOfBound</td>
</tr>
<tr>
<td>HBM</td>
<td>0.887</td>
<td>0.112</td>
<td>4 int overflow alarms</td>
</tr>
<tr>
<td>PingPong</td>
<td>0.429</td>
<td>0.054</td>
<td>No ArrayOutOfBound</td>
</tr>
<tr>
<td>ADC</td>
<td>MemOut</td>
<td>MemOut</td>
<td>70 overflow alarms</td>
</tr>
<tr>
<td>S_Control</td>
<td>MemOut</td>
<td>MemOut</td>
<td>538(473o/19d/46a)</td>
</tr>
</tbody>
</table>

Some alarms can be further removed if considering the application scenario (such as timing constraints)

Precision of SEQ&DF_SEQ is the same and the scalability of DF_SEQ is much better
Overview

• Motivation
• Interrupt-driven programs
• Sequentialization of IDPs
• Analysis of sequentialized IDPs via abstract interpretation
• Experiments
• Conclusion
Conclusion

• **Contribution:** a *sound* approach for numerical static analysis of embedded C software with interrupts

[Diagram showing IDPs leading to Sequential Programs with a note on numerical static analysis via abstract interpretation]
Conclusion

• **Contribution:** a *sound* approach for numerical static analysis of embedded C software with interrupts

IDPs \(\rightarrow\) Seq \(\rightarrow\) Sequential Programs

- a simple model with restrictions and assumptions
- static analysis (abstract interpretation)
Conclusion

• **Contribution:** a *sound* approach for numerical static analysis of embedded C software with interrupts

IDPs \(\xrightarrow{Seq}\) Sequential Programs

consider data flow dependency to sequentialize IDPs (scalability)
Conclusion

• **Contribution:** a **sound** approach for numerical static analysis of embedded C software with interrupts

  - IDPs
  - Specific abstract domains for sequentialized IDPs (precision)
  - Numerical static analysis via abstract interpretation

  - Sequential Programs
Thank you Any Questions?