The 2016 Workshop on Static Analysis of Concurrent Software, Edinburgh, Scotland

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

areaspace

medical equipment

automobile

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

- 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

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

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

Sequential program analysis: no division-by-zero

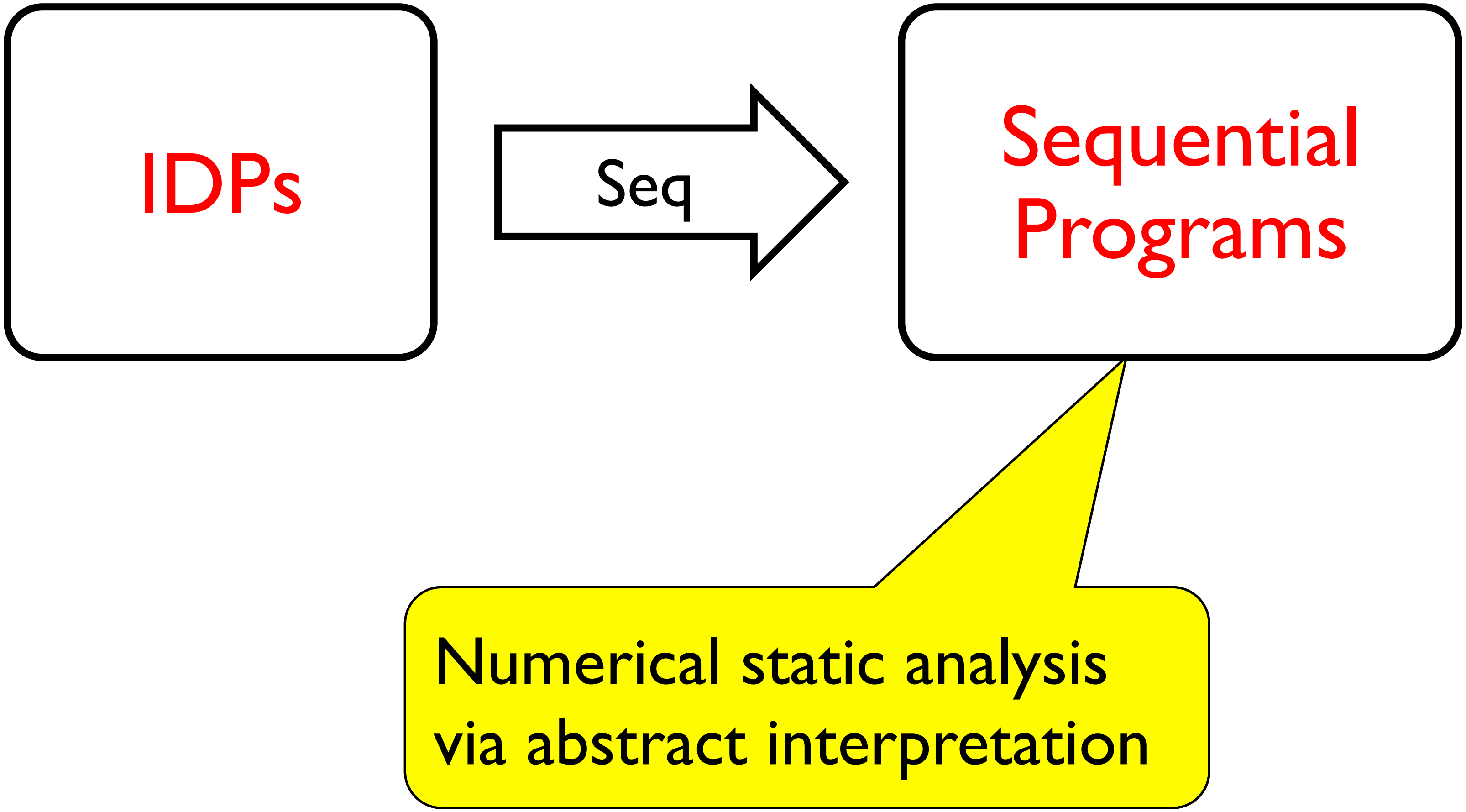
UNSOUND !

Interrupt semantics: Given x=1,y=3 , if ISR fires at ❶, there is a division-by-zero error at ❷

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 unicore 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

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

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, $E_1, E_2 \in \mathit{Expr}$ and $\diamond \in \{+, -, \times, \div\}$)
Stmt	$:=$	$l = g \mid g = l \mid l = e \mid S_1; S_2 \mid \textbf{skip} \mid \mathit{enableISR}(i)$ $\mid \mathit{disableISR}(i) \mid \textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2$ $\mid \textbf{while } e \textbf{ do } S$ (where $l \in NV, g \in SV, e \in \mathit{Expr}, i \in [1, N]$, $S_1, S_2, S \in \mathit{Stmt}$)
Task	$:=$	entry (where $\mathit{entry} \in \mathit{Stmt}$)

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. $IMR_{ISR_i^{entry}} = IMR_{ISR_i^{exit}}$

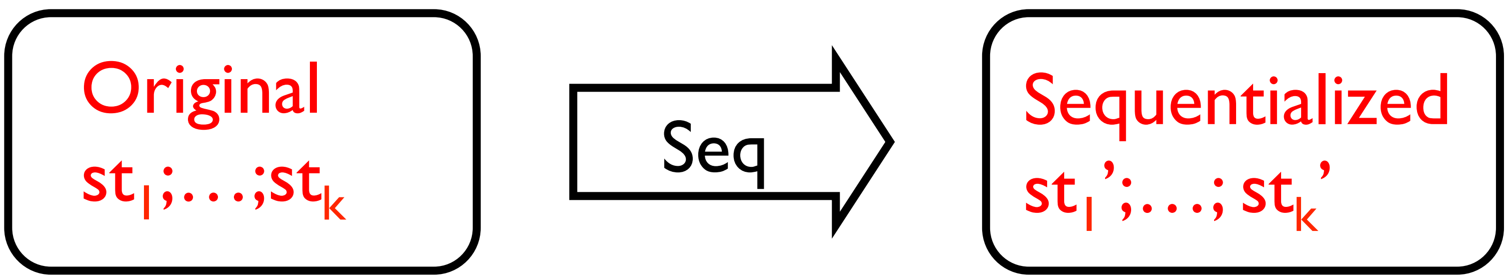
     keeping IMR intact holds for practical IDPs, e.g., satellite control programs

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



where st_i' = schedule(); st_i

Example

only allow l=g and g = l

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

```
int x, y, z;
void task'(){
  int tx, ty;
  tx = x;
  ty = y;
  if(tx < ty){
   tx = x;
   ty = y;
   z = 1/(tx-ty);
  }
  return ;
}
```

```
void ISR'(){
    int tx, ty;
    tx = x;
    tx = tx + 1;
    x = tx;
    ty = y;
    ty = ty + 1;
    y = ty;
    return ;
}
```

Example

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

Add schedule() before each program statement

```
int Prio=0;
//current priority
ISR ISRs_seq[N];
//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 ;
}
```

```
    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(){
    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

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

```
int x, y, z;
int Prio=0;
//current priority
ISR ISRs_seq[N];
//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(); return ;
}
```

```
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;}
```

```
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;
}
```

Non-deterministically schedule higher priority interrupts

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; …; St_n}, where only St_n reads shared variables (SVs)

  - basic sequentialization

```
{ schedule(); St_1; schedule(); St_2; … ; schedule() ; St_n }
```

  - consider SVs

```
{ St_1; St_2; …; St_n-1 ;
   for(int i=0;i<n;i++)
     schedule();
   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

```
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 +oo

Example

```
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; }
```

These statements access shared variables

Example

```
int x,y,z;
void task(){
  int t, t
  x = 1(
  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;}
```

only invoke *scheduleG_K()* before reading or after writing SVs

Seq

```
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; 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();
```

Example

```
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;}
```

only invoke relevant ISRs

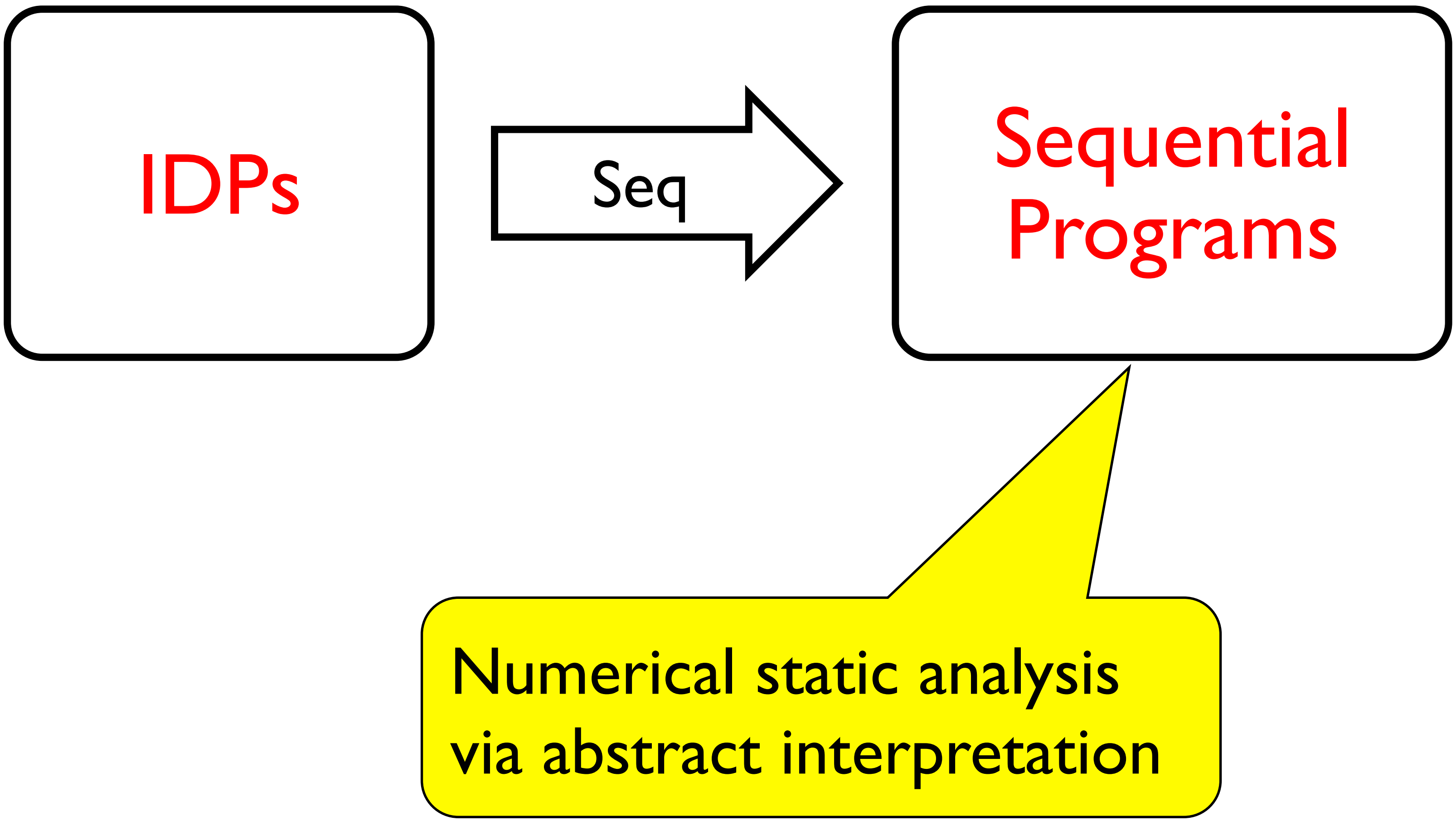
Seq

```
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; 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();
```

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



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

```
int x;
void task(){
   int tx,z;
   x=0;
   tx=x;
   tx=tx+1;
   x=tx;
   z=1/(x-5);
}
```

```
void ISR1(){
   int tx;
   tx = x;
   tx = tx+10;
   x = tx;
}
```

ISR1 hasn't fired

ISR1 has fired

```
int x;
void task(){
   int tx,z;
   x=0;              /* x^nf ∈ [0,0], x^f ∈ [0,0]     */
   if(*)  ISR1();    /* x^nf ∈ [0,0], x^f ∈ [10,10]   */
   tx=x;
   tx=tx+1;          /* x^nf ∈ [0,0], x^f ∈ [10,10]   */
   x=tx;             /* x^nf ∈ [1,1], x^f ∈ [11,11]   */
   if(*)  ISR1();    /* x^nf ∈ [1,1], x^f ∈ [11,11]   */
   z=1/(x-5);        /*      division is safe         */
}
```

If only using interval domain: $x \in [1,21]$ 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

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

orginal program fragment

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

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

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

The scale of sequentialized program by DF_SEQ is smaller than SEQ

Experiments

- Experiment of numerical static analysis

Program	Analysis of SEQ (s)		Analysis of DF_SEQ(s)		Warnings & Proved Properties
Name	BOX	OCT	BOX	OCT	
iRobot3	0.303	2.979	0.069	0.636	2 int overflow alarms
UART	0.732	5.782	0.128	1.177	No ArrayOutofBound
HBM	0.887	5.661	0.112	1.076	4 int overflow alarms
PingPong	0.429	2.434	0.054	0.251	No ArrayOutofBound
ADC	MemOut	MemOut	343.5	MemOut	70 overflow alarms
S_Control	MemOut	MemOut	5325	MemOut	538(473o/19d/46a)

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

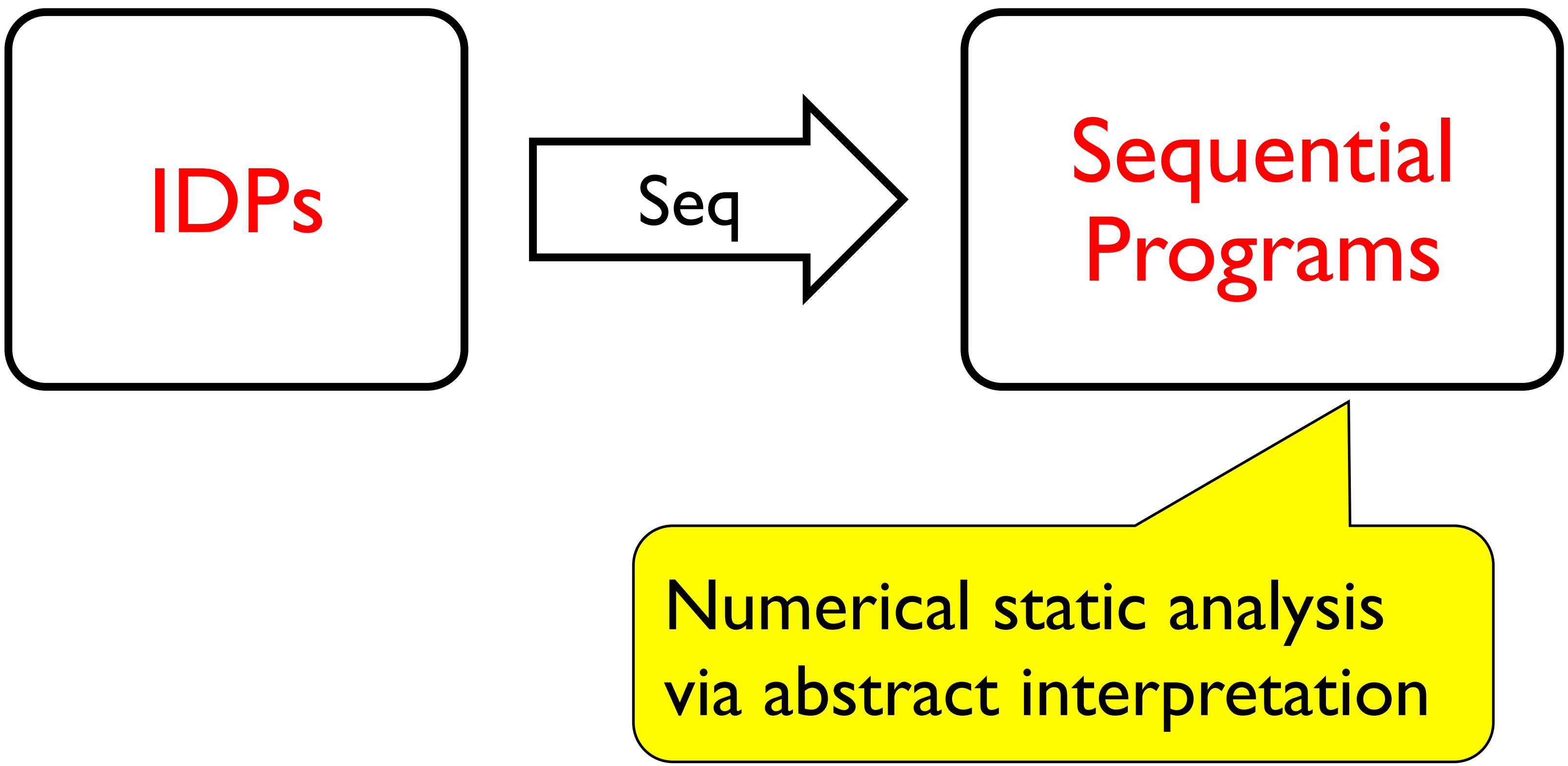
Precision of SEQ&DF_SEQ is the same and the scalability of DF_SEQ is much better

Overview

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

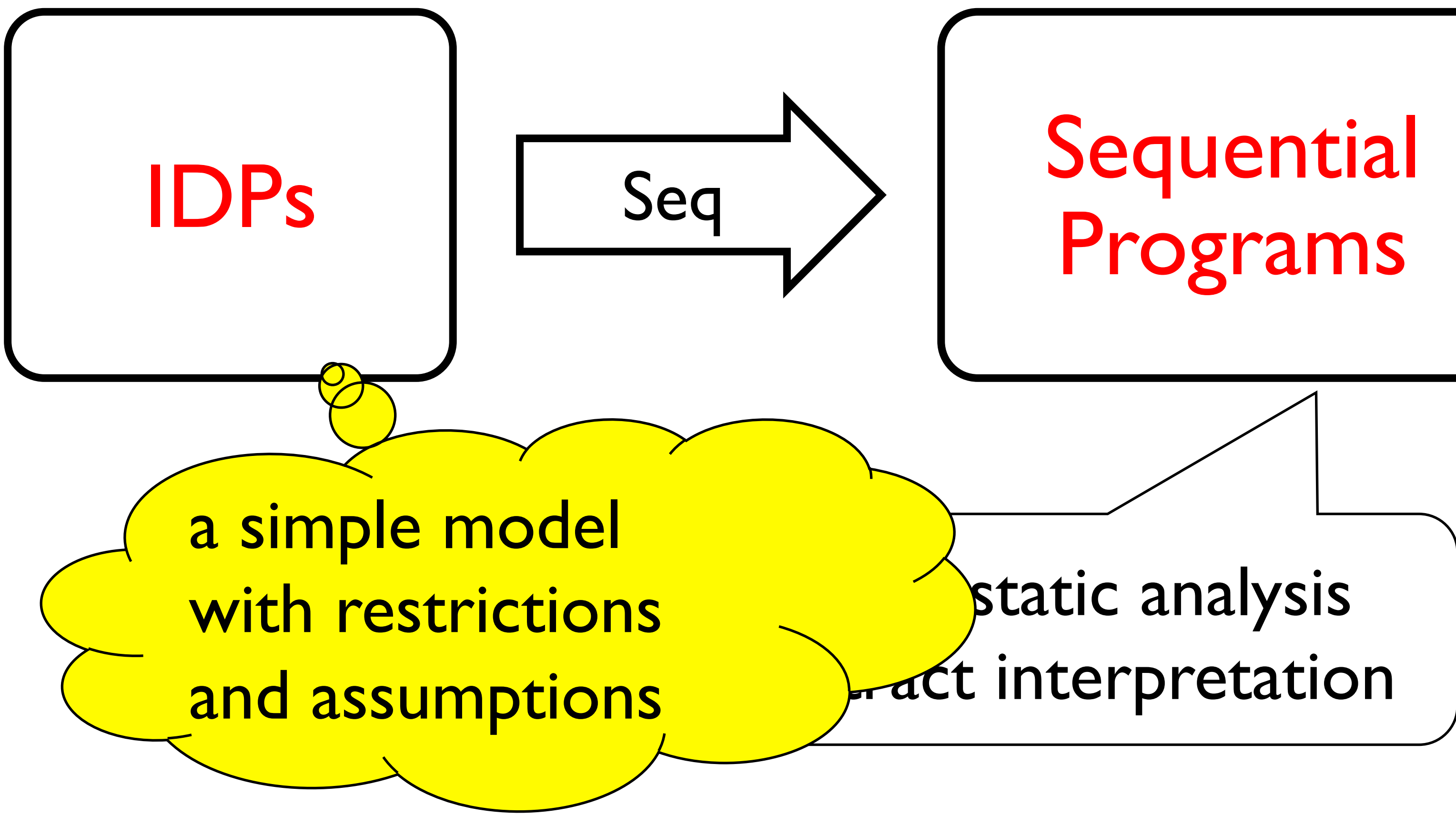
Conclusion

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



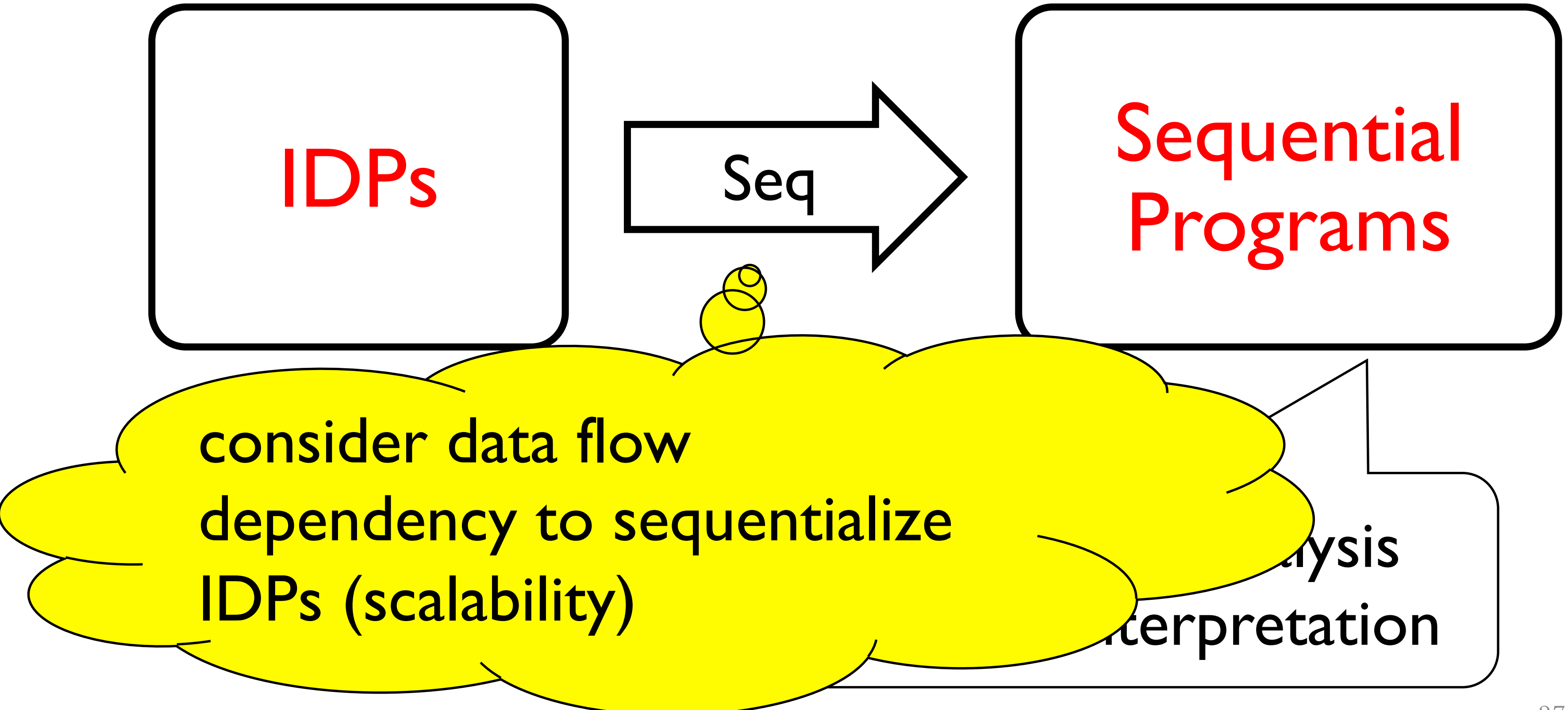
Conclusion

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



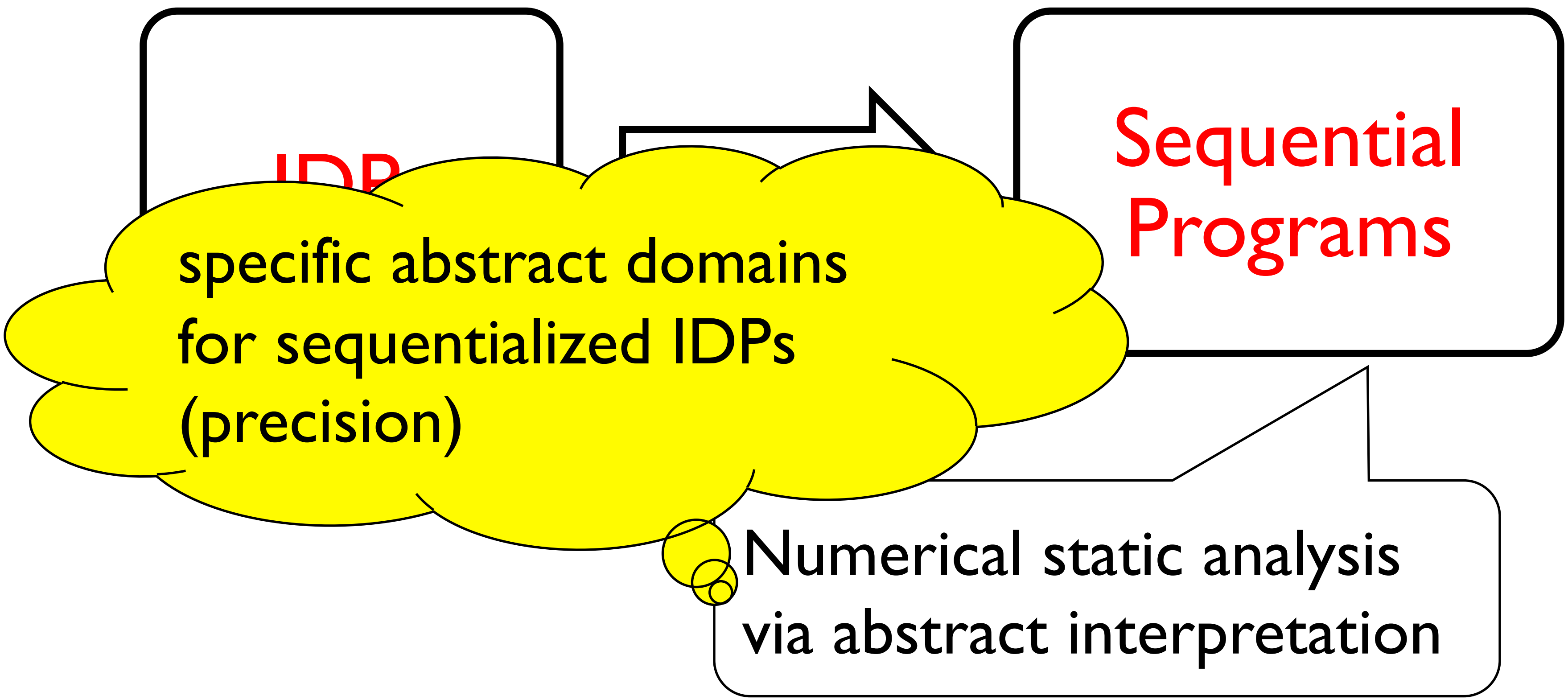
Conclusion

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



Conclusion

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



Thank you

Any

Questions?