Numerical Static Analysis of Interrupt-Driven Programs via Sequentialization

Xueguang Wu[1] Liqian Chen[1] Antoine Miné[2] Wei Dong[1] Ji Wang[1]

[1]National University of Defense Technology, Changsha, China

[2]CNRS & LIP6, UPMC, Paris, France

05/10/2015 – EMSOFT 2015

Overview

- Motivation
- Interrupt-driven programs (IDPs)
- Sequentialization of IDPs
- Analysis of sequentialized IDPs via abstract interpretation
- Implementation and experiments
- Conclusion

Interrupts in Embedded Software

- Interrupts are a commonly used technique that introduce concurrency in embedded software
- Embedded software may contain intensive numerical computations which are error prone

satellite

medical equipment

automobile

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;
}
```

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

Sequential program analysis:
no division-by-zero

UNSOUND !

Existing Work

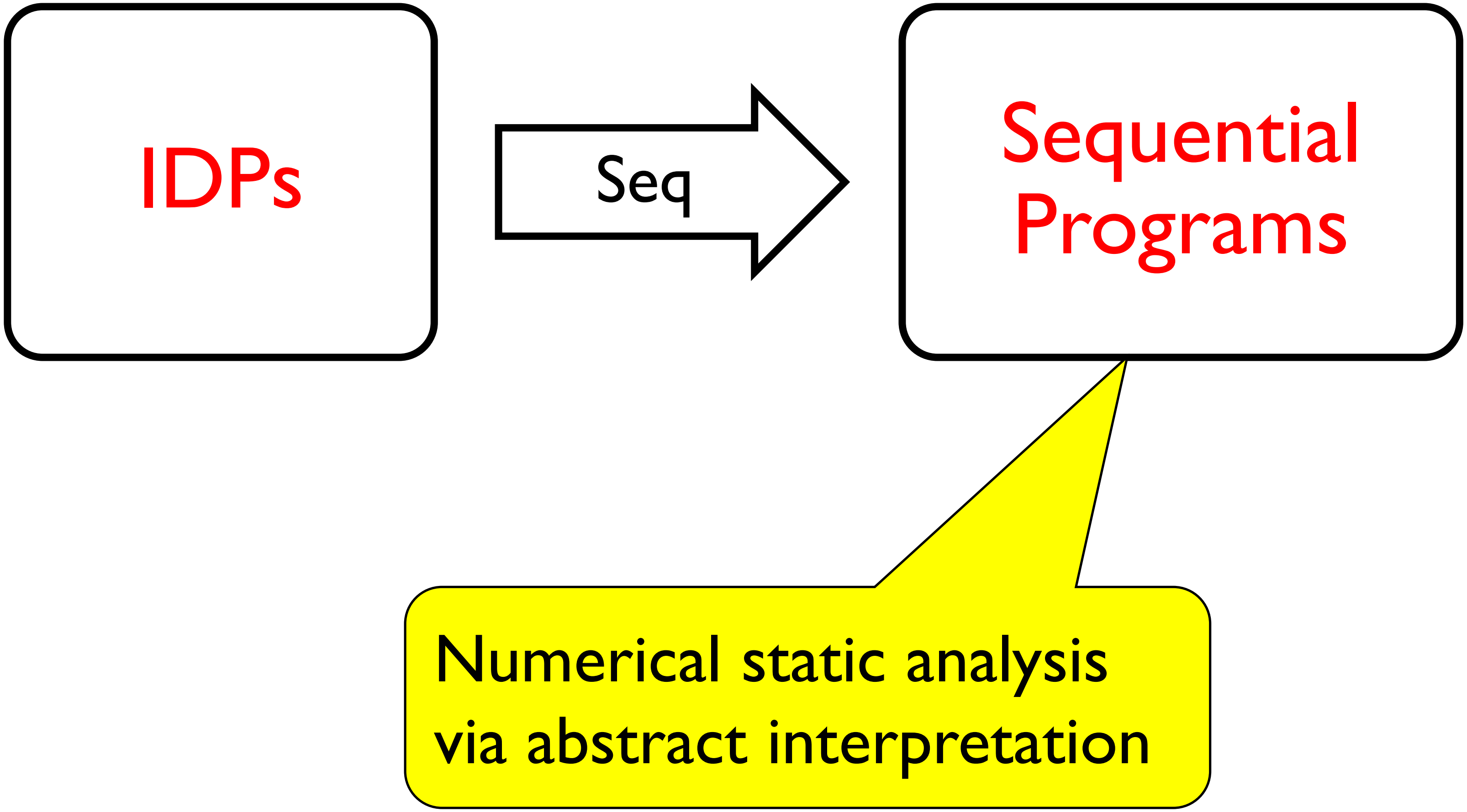
- Sequentialization methods for concurrent programs
  - KISS [PLDI'04], Kidd et al. [SPIN'10], REKH [VMCAI'13], Cseq [ASE'13], …
- Numerical static analysis of concurrent embedded software
  - cXprop [LCTES'06], Monniaux [EMSOFT'07], AstréeA[ESOP'11] …

Few existing numerical static analysis methods consider interrupts

Our Goal

- Challenges of analyzing IDPs
  - interleaving state space can grow exponentially with the number of interrupts (scalability)
  - interrupts are controlled by hardware (precision)
    - e.g., periodic interrupts, interrupt mask register (IMR)
- Goal
  - a sound approach for numerical static analysis of embedded C programs with **interrupts**

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
- Implementation and 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
- Application scenarios

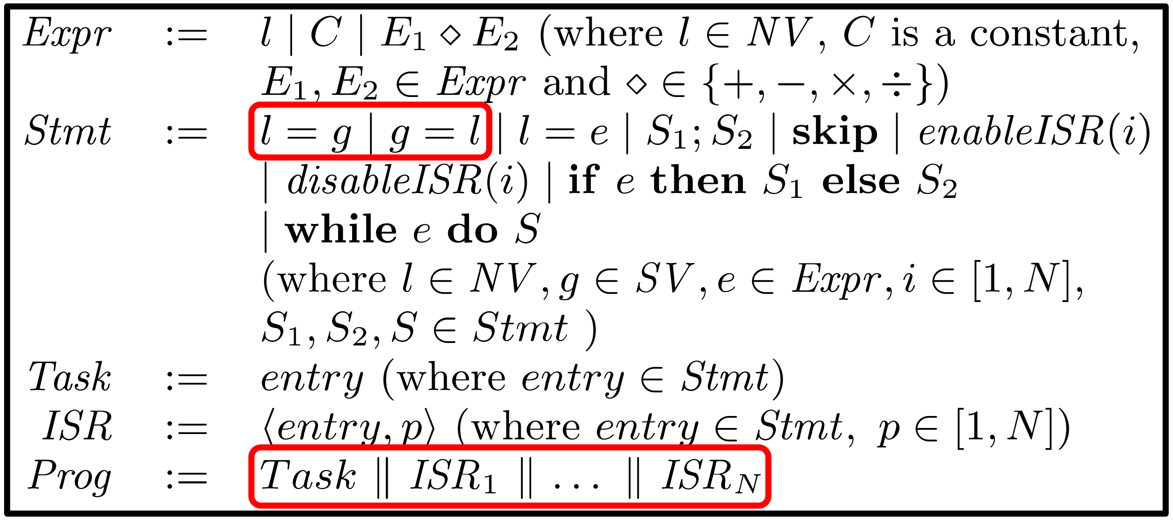
Satellite

Wireless network OS

LEGO robotics (OSEK)

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

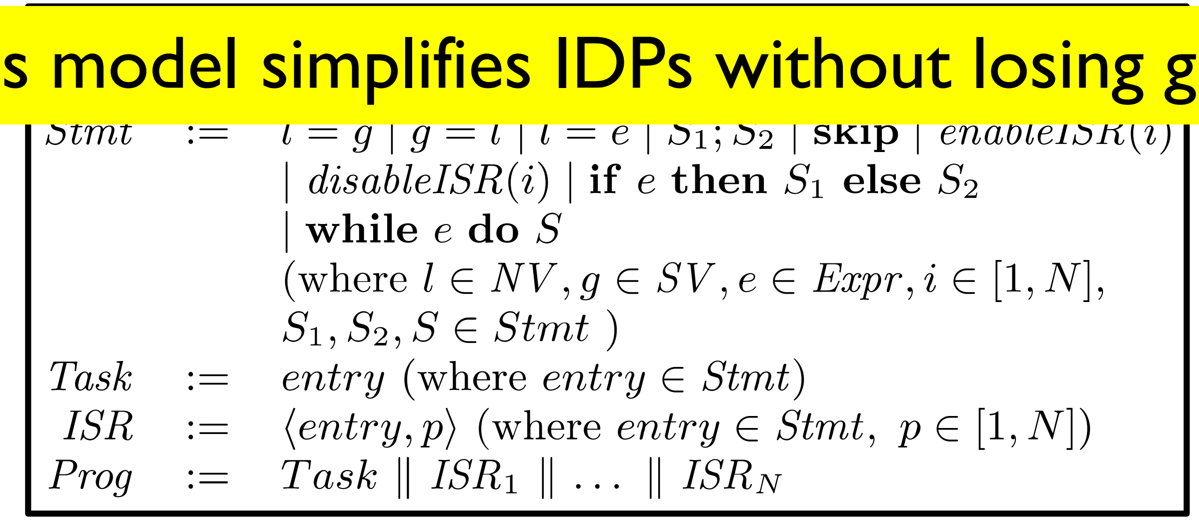


$$
\begin{array}{lcl}
\mathit{Expr} & := & l \mid C \mid E_1 \diamond E_2 \text{ (where } l \in NV,\ C \text{ is a constant,} \\
 & & E_1, E_2 \in \mathit{Expr} \text{ and } \diamond \in \{+,-,\times,\div\}) \\
\mathit{Stmt} & := & \boxed{l = g \mid g = l} \mid l = e \mid S_1; S_2 \mid \textbf{skip}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

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

This model simplifies IDPs without losing generality



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

Interrupt-Driven Programs

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

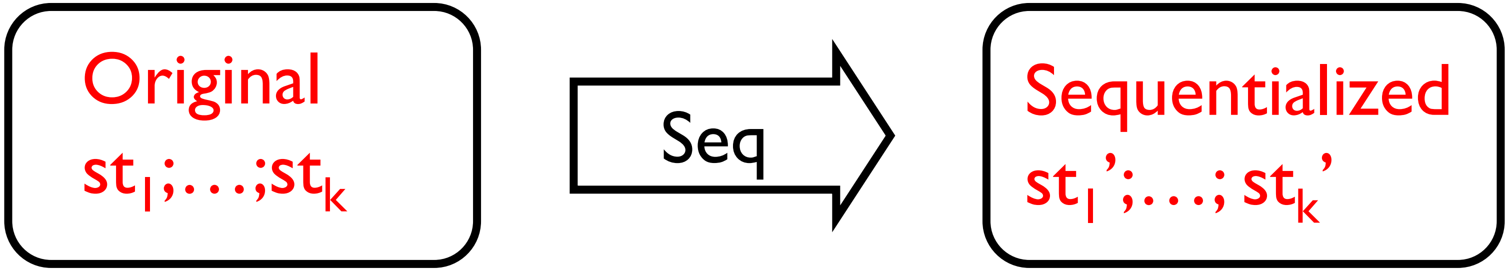
keeping IMR intact holds for practical IDPs, e.g., satellite control programs

Overview

- Motivation
- Interrupt-driven programs (IDPs)
- Sequentialization of IDPs
- Analysis of sequentialized IDPs via abstract interpretation
- Implementation and 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

$st_1;\ldots;st_k$

Seq

Sequentialized

$st_1';\ldots;st_k'$

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

- **The disadvantage** 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<K;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,
  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

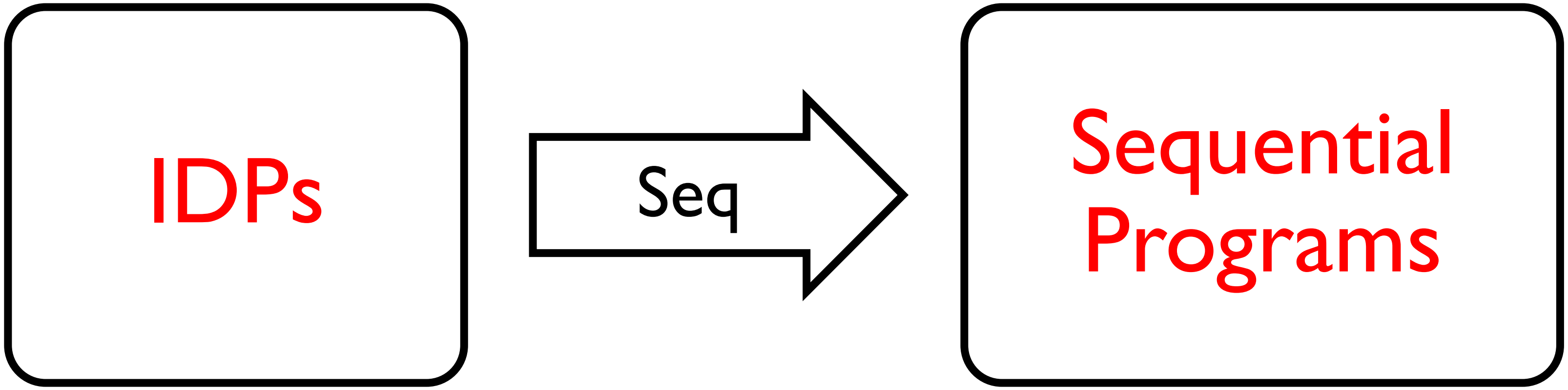
Seq

```
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
- Implementation and experiments
- Conclusion

Analysis of Sequentialized IDPs via Abstract Interpretation



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

A Specific Abstract Domain 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

A 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;             /* $x^{nf} \in [0,0], x^{f} \in [0,0]$     */
   if(*)  ISR1();   /* $x^{nf} \in [0,0], x^{f} \in [10,10]$ */
   tx=x;
   tx=tx+1;         /* $x^{nf} \in [0,0], x^{f} \in [10,10]$  */
   x=tx;            /* $x^{nf} \in [1,1], x^{f} \in [11,11]$  */
   if(*)  ISR1();   /* $x^{nf} \in [1,1], x^{f} \in [11,11]$  */
   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

Overview

- Motivation
- Interrupt-driven programs (IDPs)
- Sequentialization of IDPs
- Analysis of sequentialized IDPs via abstract interpretation
- Implementation and experiments
- Conclusion

Implementation and Experiments

- Implementation
  - frontend: CIL
  - numerical abstract domain library: Apron
- Benchmarks
  - OSEK programs from Goblint [Schwarz et al. POPL11]
  - LEGO robotic control program (Nxt_gs)
  - universal asynchronous receive and transmitter (UART)
  - ping pong buffer program from satellite application program
  - ADC controller from satellite application program
  - a satellite control program

Implementation and Experiments

- Aims of the experiments
  - check run time errors of IDPs
  - compare the generated program size and the time consumption of sequentialization methods with and without considering data flow dependency
  - compare the scalability and precision of numerical static analysis for sequentialization methods with and without considering data flow dependency

Implementation and Experiments

- Experiments of sequentialization

Program					**Sequentialization**				
Name	Loc_ task	Loc_ ISR	#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
PingPong_Sate	130	53	21	1	3159	0.006	842	0.006	26.65
ADC_Sate	1870	2989	312	1	123K	0.449	23K	0.8	18.70
Satellite_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

Implementation and Experiments

- Experiment of numerical static analysis

Program	**Analysis of SEQ (s)**		**Analysis of DF_SEQ (s)**		**Warnings & Proved Properties**
Name	BOX	OCT	BOX	OCT	
Motv_Ex	0.007	0.011	0.006	0.007	Div-by-zero
DataRace_Ex	0.042	0.053	0.011	0.015	Assertion holds
Privatize	0.029	0.036	0.005	0.007	Assertion holds
Nxt_gs	0.113	0.140	0.040	0.046	Integer overflow
UART	0.732	5.782	0.128	1.177	No ArrayOutofBound
Ping_Pong	0.429	2.434	0.054	0.251	No ArrayOutofBound
ADC_Sate	MemOut	MemOut	80.5	MemOut	143(109/0/34)
Satellite Control	MemOut	MemOut	5190	MemOut	544(479/19/46)

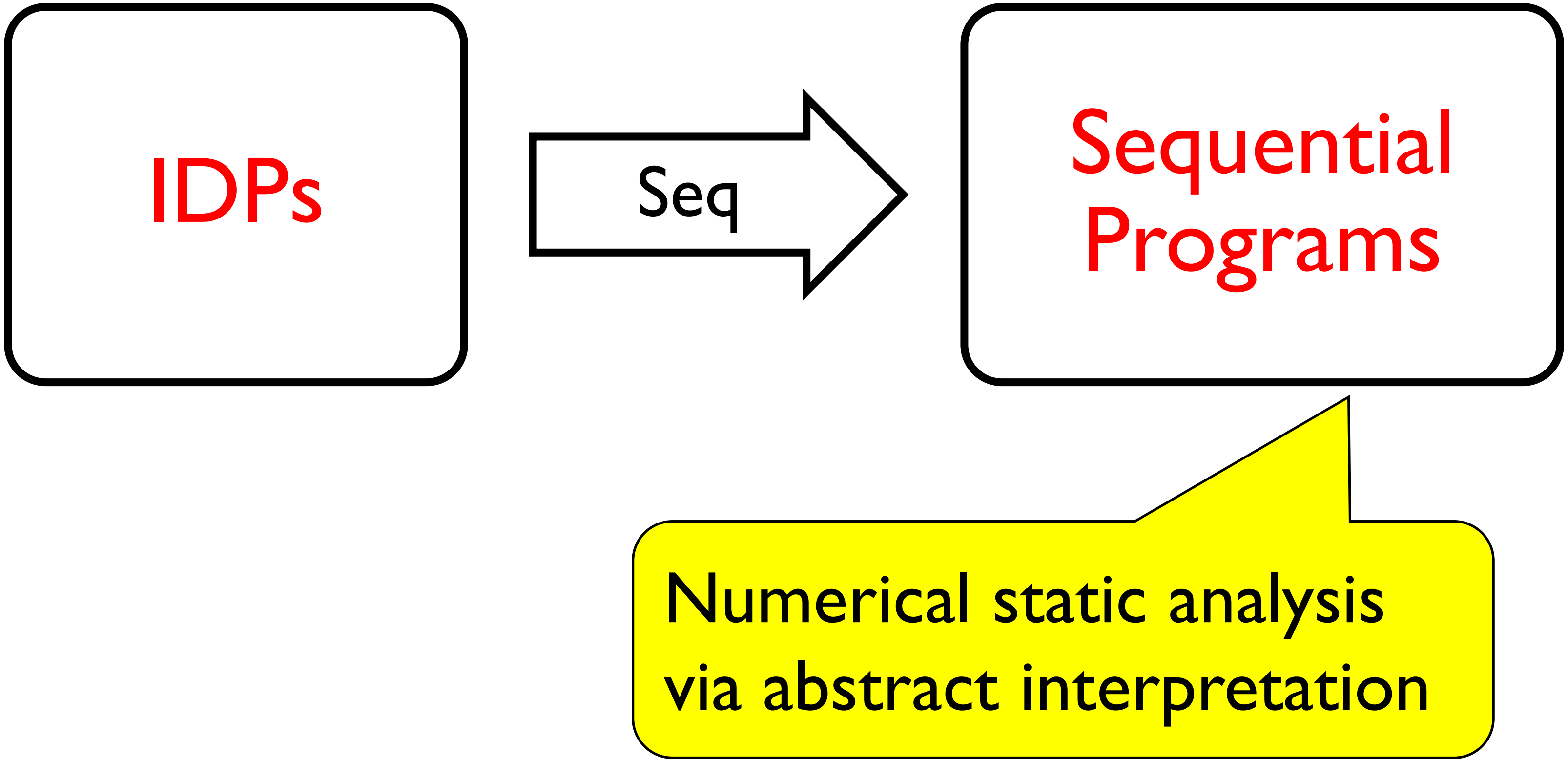
Precision of SEQ&DF_SEQ is the same and the scalability of DF_SEQ is much better

Overview

- Motivation
- Interrupt-driven programs
- Sequentialization of IDPs
- Analysis of sequentialized IDPs via abstract interpretation
- Implementation and experiments
- Conclusion

Conclusion

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



IDPs

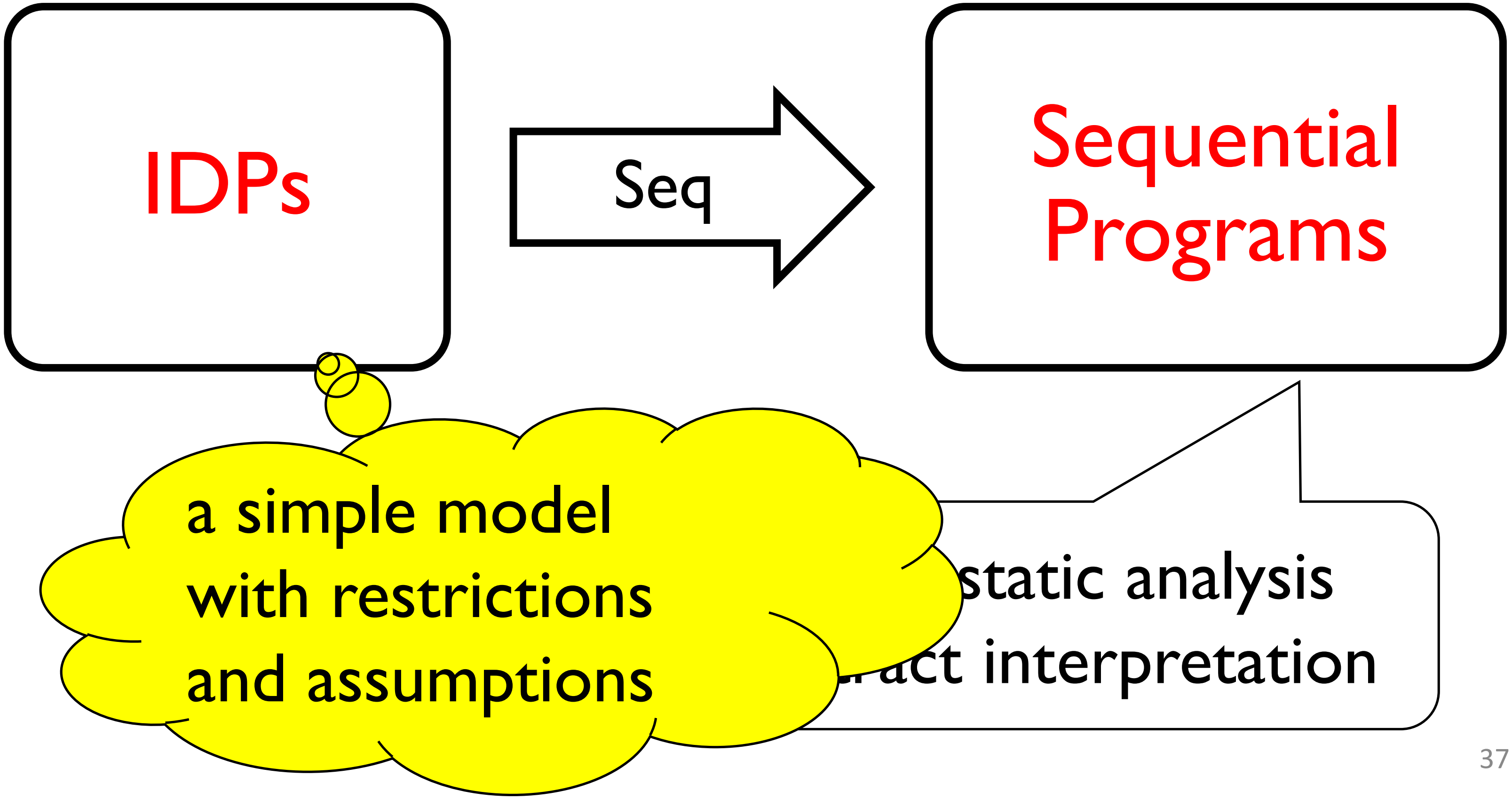
Seq

Sequential Programs

Numerical static analysis via abstract interpretation

Conclusion

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



IDPs

Seq

Sequential Programs

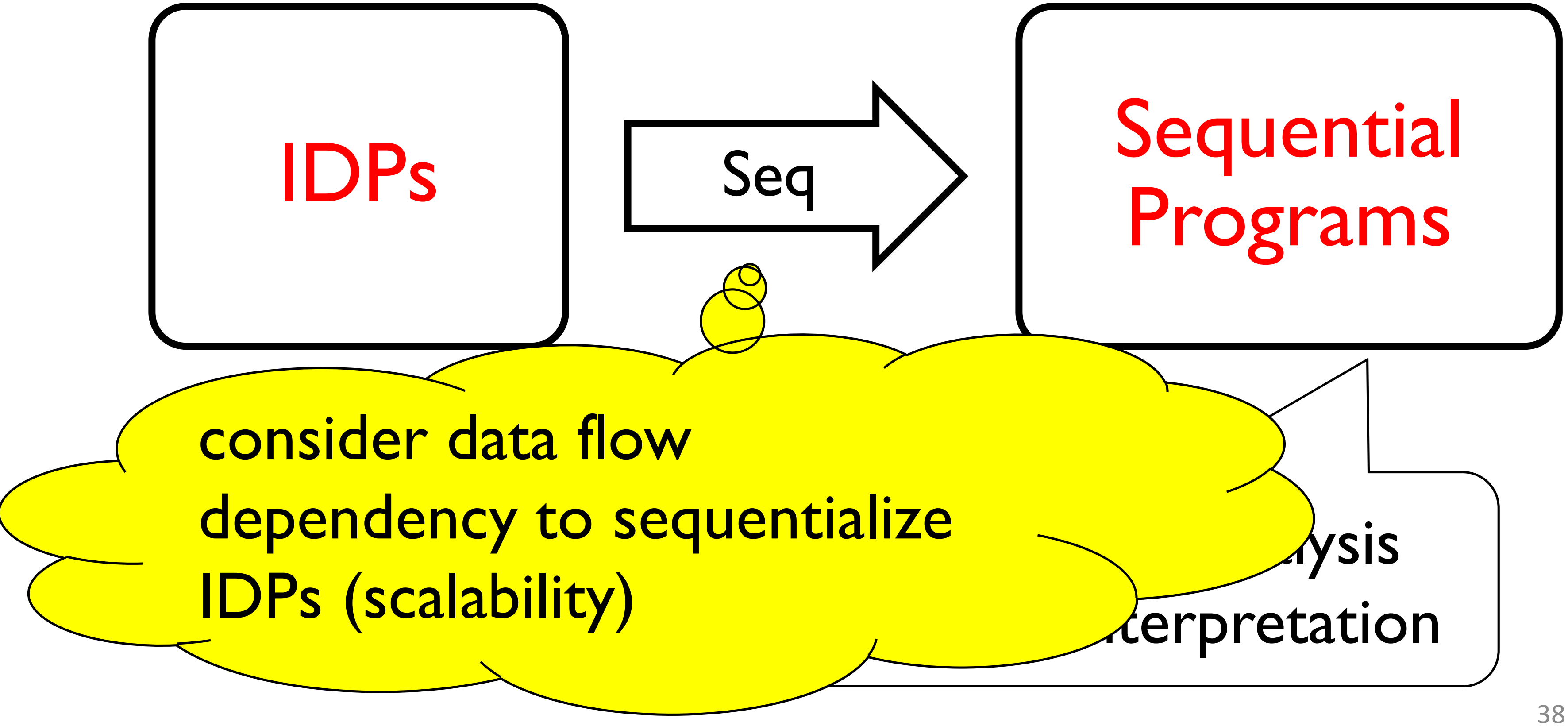
a simple model with restrictions and assumptions

static analysis

act interpretation

Conclusion

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



IDPs

Seq

Sequential Programs

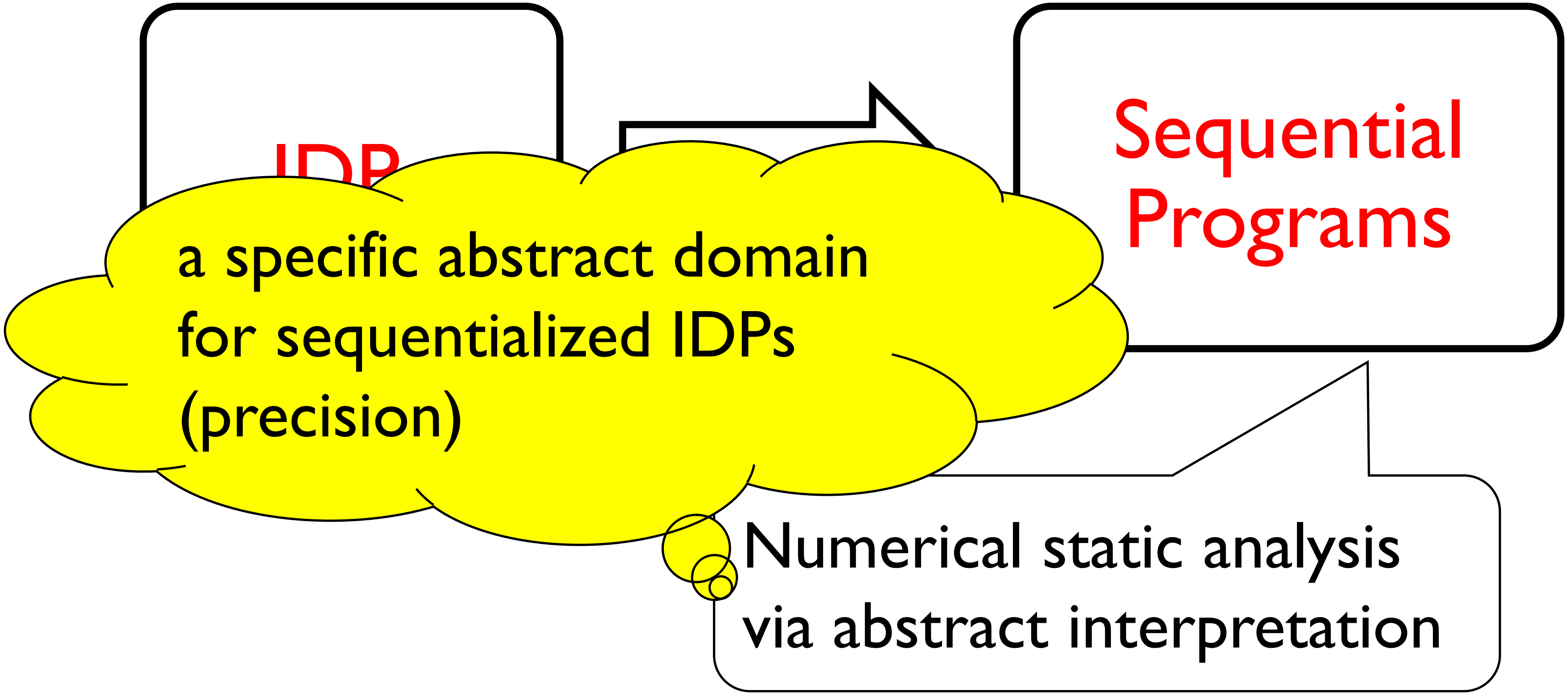
ysis

terpretation

consider data flow dependency to sequentialize IDPs (scalability)

Conclusion

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



IDP

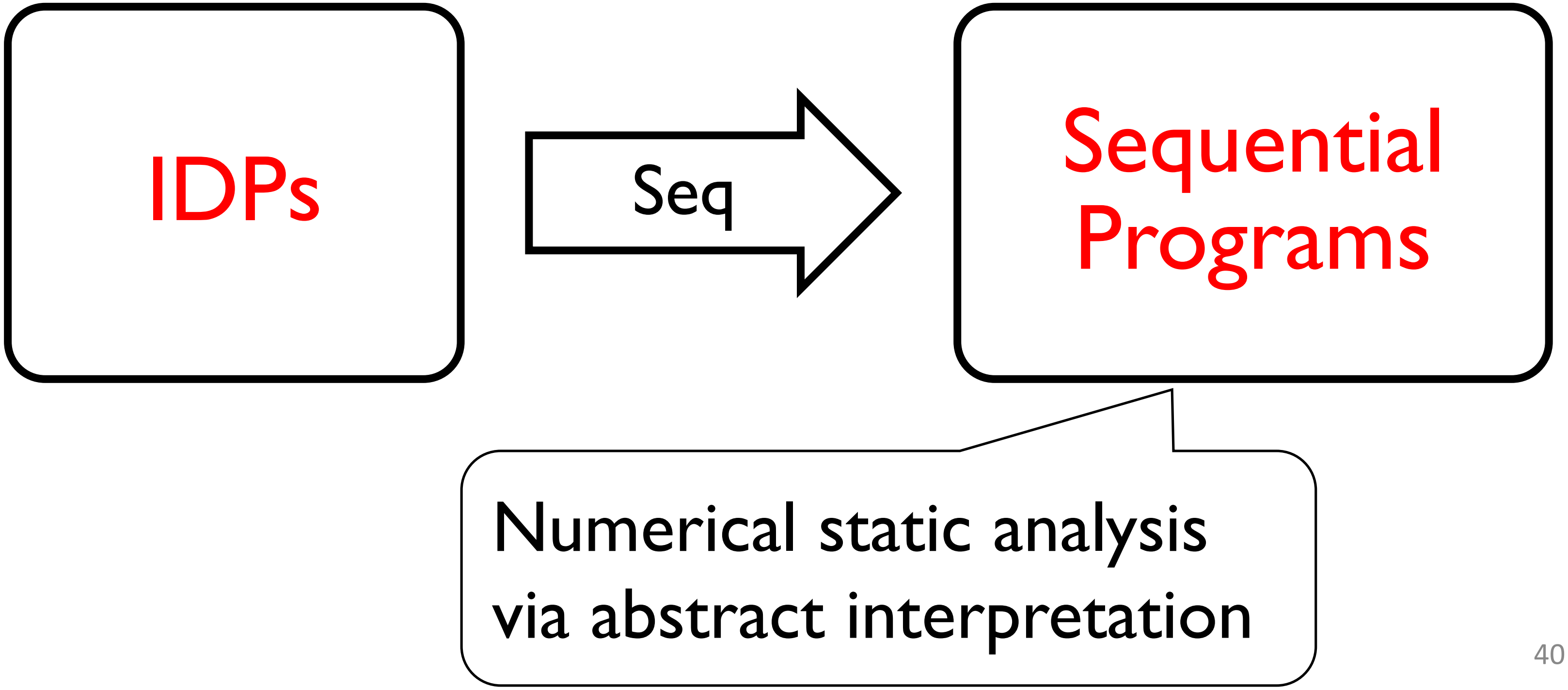
Sequential Programs

a specific abstract domain for sequentialized IDPs (precision)

Numerical static analysis via abstract interpretation

Conclusion

- Future work
  - extending the model to support IDPs with tasks preemption tasks
  - designing more specific abstract domains that fit IDPs



IDPs

Seq

Sequential Programs

Numerical static analysis via abstract interpretation

Thank you
Any Questions?