Analyzing and Disentangling Interleaved Interrupt-driven IoT Programs

YUXIA SUN, MEMBER, IEEE, SONG GUO, SENIOR MEMBER, IEEE, SHING-CHI CHEUNG, SENIOR MEMBER, IEEE, AND YONG TANG, MEMBER, IEEE

IEEE INTERNET OF THINGS JOURNAL

Introduction

- In the IoT community, Wireless Sensor network (WSN) is a key technique to enable ubiquitous sensing of environments and provide reliable services to applications
- WSN program are interrupt-driven in order to reduce energy comsumption
- WSN concurrency mechanism involves interrupt preemption and task scheduling. For instance, an interrupt processing logic consist of one interrupt handler (which is execute immediately) and several interrupt-processing task (which is deferred)
- Due to the concurrency mechanism of WSN, program's behavior is difficult to predict and test.
- With the reasons above, using static analyses to WSN is not effective. In contrast, dynamic analyses can precisely examine the actual behavior of program
- Also, WSN program's behavior consist of collaborative Interrupt Procedure Instances (IPI), so IPI-based analyses is indispensable.

Introduction

- Furthermore, online (real-time) analyses can also help uncover time-related issue
- Conclude the reasons above, this paper makes the following contribution:
  - Present a formal definition of Interrupt Procedure instance
  - Propose a generic algorithm for identifying IPIs of WSN programs
  - Prove the correctness、efficiency and real-time of the algorithm
  - Implement a prototype of the algorithm and compare to existing ones

Interrupt Procedure Instances (IPI) – Fundemental

- In this paper, they use TinyOS, an mainstream operating system for WSN programming, as the basis of IPI’s definition.
- In a nesC (programming language) module m, a task t() and its task-posting statement post(t) will compiled to two function taskName.runTask() and taskName.postTask(), where taskName denotes m.t
- taskName.postTask(): It will post the task into OS task queue
- taskName.runTask(): If a task is successfully pushed, it will be scheduled in a FIFO manner.

Interrupt Procedure Instances (IPI) – Definition

- Let IH be the interrupt handler of an interrupt i
- Definition 1: The interrupt-procedure of IH consists of the static codes of three nesC modules, IH, the callees of IH(or i), and the tasks of IH where
  (1) A callee of IH is a function that is called by IH, a callee of IH, or a task of IH.
  (2) A task of IH is a task that is posted by IH, a callee of IH, or a task of IH.
- Definition 2. An interrupt-procedure-instance (abbr. IPI) of IH(or i) is one execution of the interrupt procedure of IH.
  The callees of the instance are the callees of IH that are executed in the instance.
  The tasks of the instance are the tasks of IH that are executed (i.e., successfully posted) in the instance.

Interrupt Procedure Instances (IPI) – Execution point & scenario

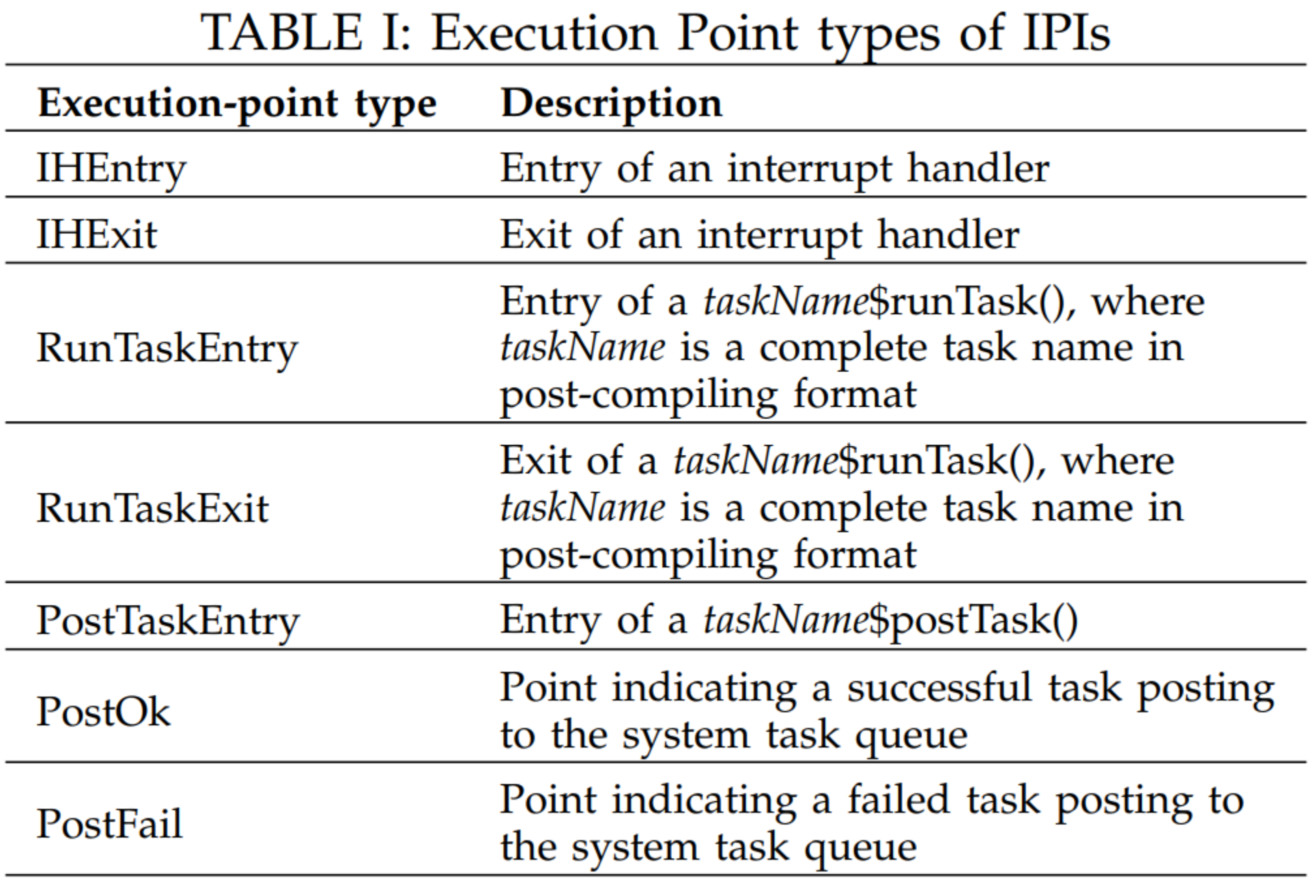


TABLE I: Execution Point types of IPIs

Execution-point type	**Description**
IHEntry	Entry of an interrupt handler
IHExit	Exit of an interrupt handler
RunTaskEntry	Entry of a *taskName*$runTask(), where *taskName* is a complete task name in post-compiling format
RunTaskExit	Exit of a *taskName*$runTask(), where *taskName* is a complete task name in post-compiling format
PostTaskEntry	Entry of a *taskName*$postTask()
PostOk	Point indicating a successful task posting to the system task queue
PostFail	Point indicating a failed task posting to the system task queue

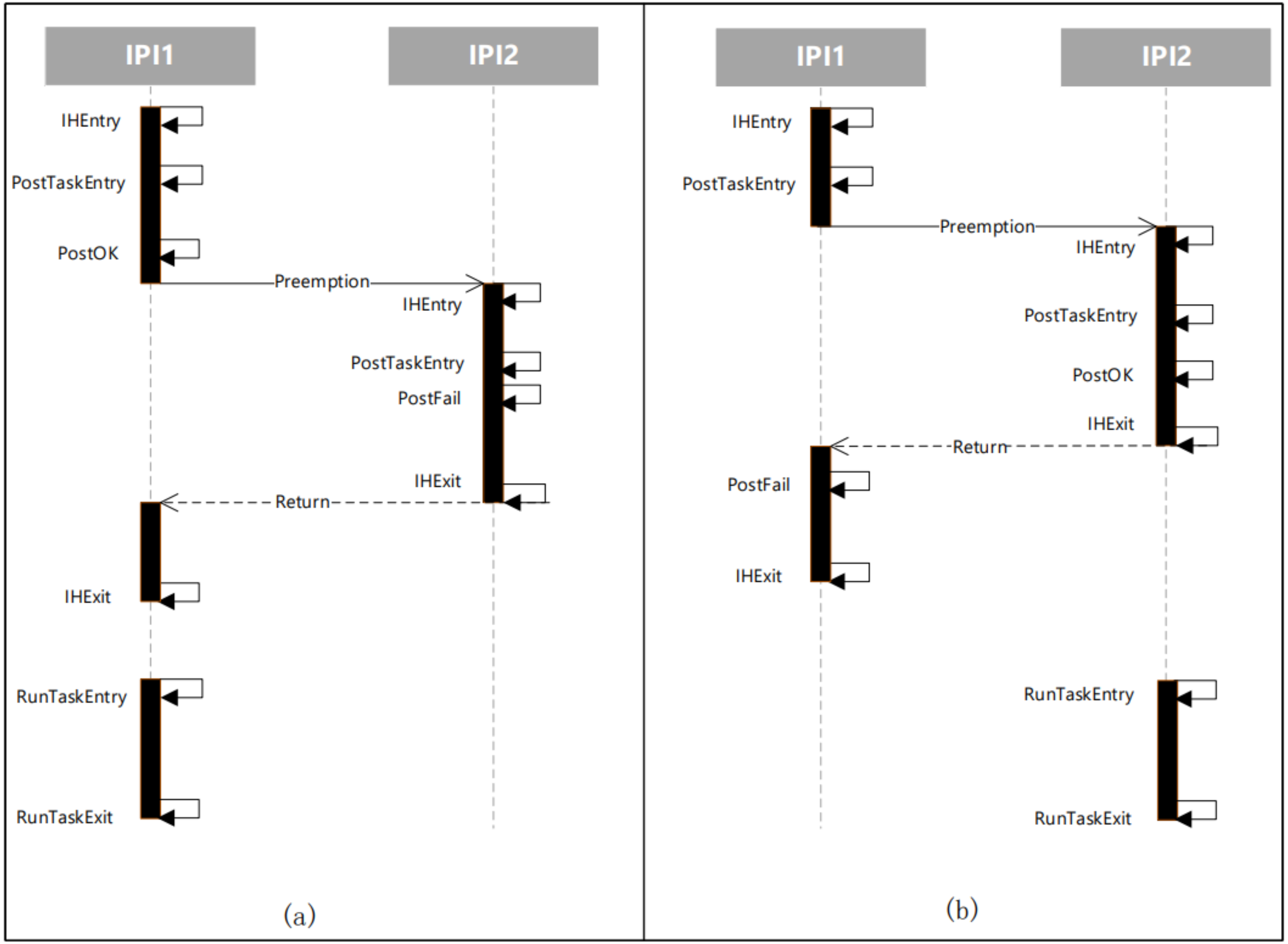


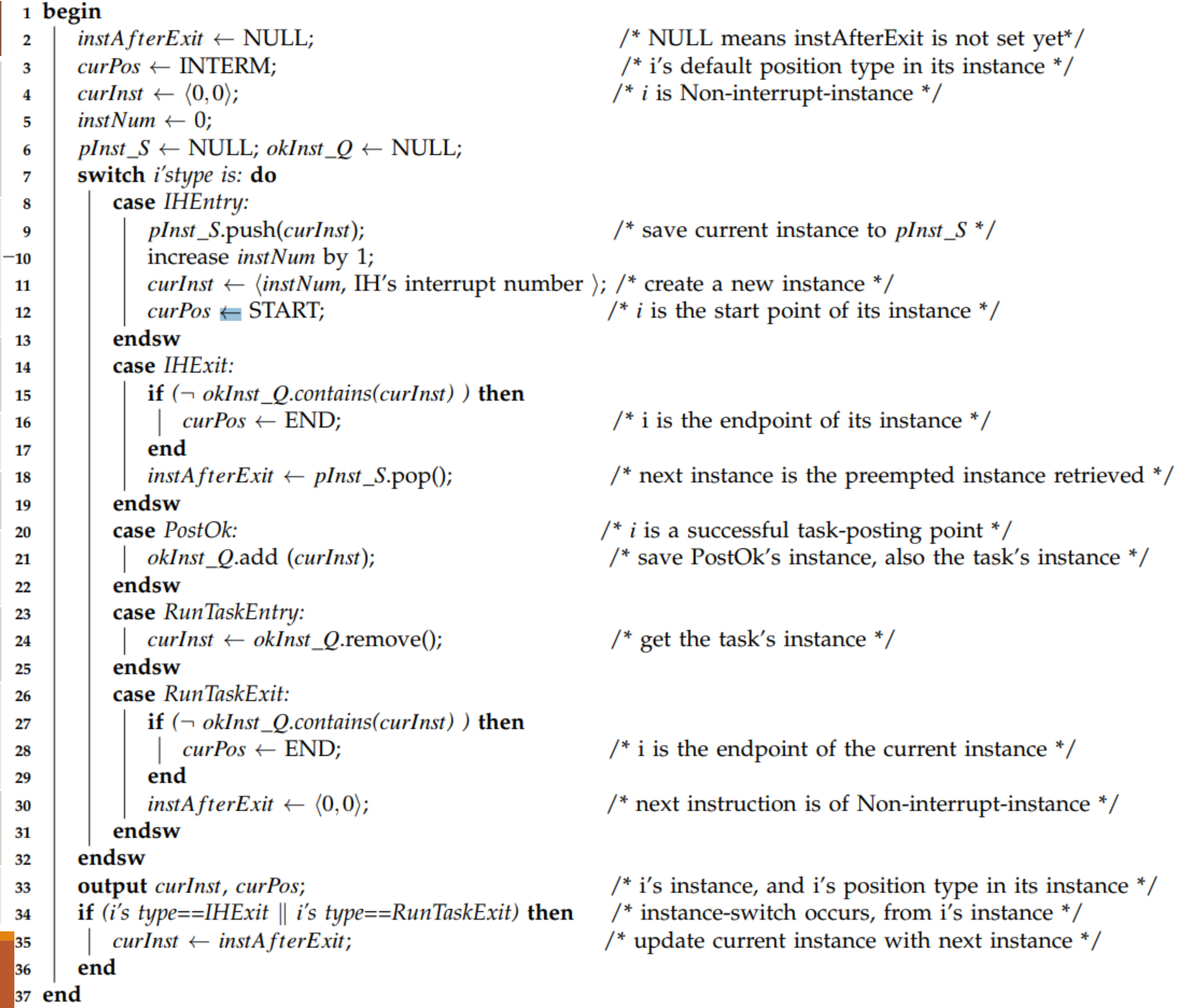
Fig. 1: Examples of Interleaving IPIs

IPI-Identification algorithm

- Non-interrupt instance : System operation such as system initialization and system scheduling between task-executions are not driven by interrupt. It doesn’t belong to any IPI and be regarded as Non-interrupt instance.

- Theorem 1: During the execution of a TinyOS program, instance switches only occur in one of the following execution points: IHEntry points, immediate successor points of IHExit points, RunTaskEntry points, and immediate successor points of RunTaskExit points.
- Proof: TinyOS program switch into either IPI or Non-interrupt instance.
- Only 3 cases that a program will switch to IPI
  - An interrupt occurs -> start Interrupt Handler (IHEntry)
  - A task scheduling occurs -> start to run the task (RunTaskEntry)
  - Preempted IPI ended -> return to previous IPI (immediate successor of IHExit)
- Only 2 cases that a program switch non-interrupt instance
  - Preempted IPI ended -> return to previous non-interrupt instance (immediate successor of IHExit)
  - A task is ended -> continue running non-interrupt instance (immediate successor of RunTaskExit)
- Proved!

Variable	Type	Description
INST <id,type>	Data structure	An IPI, where id is instance id and type is interrupt number of the instance's triggering interrupt
POSTYPE	enum	Includes START END and INTERM, indicating the position point of the instance
i	Input	Current instruction being executed
curInst	Global	i's instance, type is INST
instNum	Global	Instance counter
pInst_S	Global	Stack of INST, preempted instances by His
okInst_Q	Global	Queue of INST, pending tasks' instances
instAfter Exit	Local	Next instruction's instance that is different from i's instance
curPos	Local	i's position type in its instance, type is POSTYPE
	Output	curInst, curPos



```
begin
    instAfterExit ← NULL;                               /* NULL means instAfterExit is not set yet*/
    curPos ← INTERM;                                    /* i's default position type in its instance */
    curInst ← ⟨0,0⟩;                                    /* i is Non-interrupt-instance */
    instNum ← 0;
    pInst_S ← NULL; okInst_Q ← NULL;
    switch i'stype is: do
        case IHEntry:
            pInst_S.push(curInst);                      /* save current instance to pInst_S */
            increase instNum by 1;
            curInst ← ⟨instNum, IH's interrupt number ⟩; /* create a new instance */
            curPos ← START;                             /* i is the start point of its instance */
        endsw
        case IHExit:
            if (¬ okInst_Q.contains(curInst) ) then
                curPos ← END;                           /* i is the endpoint of its instance */
            end
            instAfterExit ← pInst_S.pop();              /* next instance is the preempted instance retrieved */
        endsw
        case PostOk:                                    /* i is a successful task-posting point */
            okInst_Q.add (curInst);                     /* save PostOk's instance, also the task's instance */
        endsw
        case RunTaskEntry:
            curInst ← okInst_Q.remove();                /* get the task's instance */
        endsw
        case RunTaskExit:
            if (¬ okInst_Q.contains(curInst) ) then
                curPos ← END;                           /* i is the endpoint of the current instance */
            end
            instAfterExit ← ⟨0,0⟩;                      /* next instruction is of Non-interrupt-instance */
        endsw
    endsw
    output curInst, curPos;                             /* i's instance, and i's position type in its instance */
    if (i's type==IHExit || i's type==RunTaskExit) then /* instance-switch occurs, from i's instance */
        curInst ← instAfterExit;                        /* update current instance with next instance */
    end
end
```

Algorithm Analysis

- Lemma 1. When Algorithm 1 is processing an IHExit execution point, the popped INST value from the stack pInst_S is the instance information of the immediate successor of the IHExit point.
  - ✓ When enter IHEntry, system will push the instruction been preempted, and pInst_S will push instance information at the same time.
  - ✓ When enter IHExit, system will pop the preempted instruction, and pInst_S will pop the instance information at the same time.

- Lemma 2. When Algorithm 1 is processing a RunTaskEntry execution point, the removed INST value from the queue okInst_Q is the instance information of the immediate successor of the RunTaskEntry point.
  - ✓ When enter PostOK, system will enqueue the task, and okInst_Q will enqueue the instance information at the same time.
  - ✓ When enter RunTaskEntry, system will dequeue the task, and okInst_Q will dequeue the instance information at the same time.

- Lemma 3. When a tested TinyOS program is executing an IHExit or RunTaskExit point, if the queue okInst_Q of Algorihtm 1 does not contain the point's instance information, the point is the endpoint of the instance.
  - ✓ According to Lemma 2, the instance information in okInst_Q has one-to-one mapping relation with the task queue in TinyOS.
  - ✓ If an instance have no instance information in okInst_Q, meaning this instance have no pending task in the task queue.
  - ✓ When the instance moves to IHExit or RunTaskExit point, if there is no instance information in okInst_Q, meaning that the instance has arrived to its endpoint .

Algorithm Analysis

- Corollary 1. The IPI-identification of Algorithm 1 is correct and real-time.
  - ✓ Correctness : Taking Theorem 1, Lemma 1 and 2 together, we conclude this algorithm can trace the switch correctly.
    With Lemma 3, we conclude that this algorithm can identify startpoint and endpoint correctly
  - ✓ Real-time: each instruction i's can be identified its instance information and position in the instance by this algorithm before next instruction is executed.
- Corollary 2. Both the space complexity and the time complexity of Algorithm 1 are constant O(1).
  - ✓ Space: mostly static variables(curInst, curPos …).
    For pInst_S and okInst_Q, the size are depends on the system, which is a small constant, so it can be considered $\Theta(1)$
  - ✓ Time: Mainly switch statement.
    For queue searching ok_InstQ.contains(curInst), because size of the queue is considered $\Theta(1)$, so the operation is constant time.
    So the time complexity is O(n), where n is the total executed instruction, which will increase with time.
  - ✓ Let $O(n) = O(t*N) = O(t)$, where t is execution time and N is # of executed instruction per time unit t, which is a constant.
  - ✓ Because a program's running time is limited, namely $t < C$, so let $O(t) = O(C)$, where C is large constant.
  - ✓ Finally, $O(C)=O(1)$

Experimental Study

- Experiment Setup:
  - Implemented in Java, utilizing probe mechanism of Avrora, a cycle-accurate instruction level simulator for sensor network.
  - The existing instance-identification technique (called the old tool) that is used for comparison is Sentomist(or T-Morph)
  - Perfoming experiment on Avrora with simulated Mica2(wireless sensor) platform and ATmega128 microcontroller, with TinyOS 2.1 in Cygwin and Windows XP, which runs on desktop computer that conatins Intel 2.7Ghz dual-core processor and 1GB RAM.

Experiment test case

- Sub1-3 is a sensor data collection program using single-hop packet transmission. Sub 4 is multi-hop packet transmission. Sub 5 using collection tree protocol (CTP).
- Each run group Rn will run 4 times with different running time {10, 50, 100, 150}(in second)
- In Sub 5, there is a bug of stopping packet-sending. When the bug occurs, the number of concerned instances on the buggy node might stop increasing, and it may increase the overhead's increment with the running time.

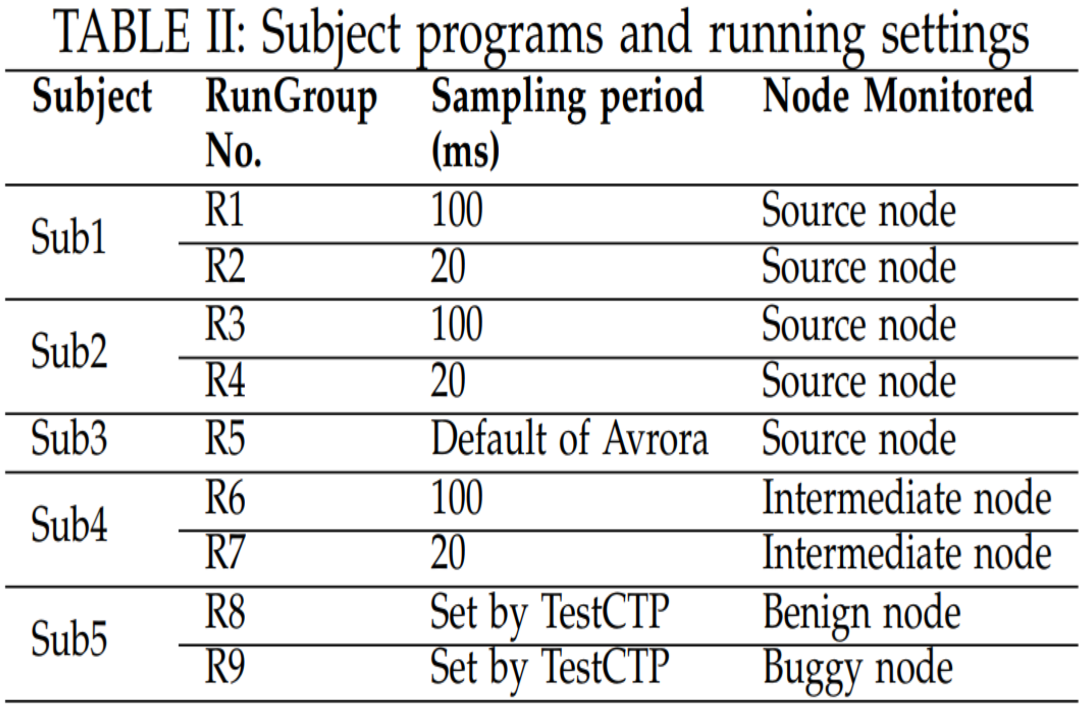
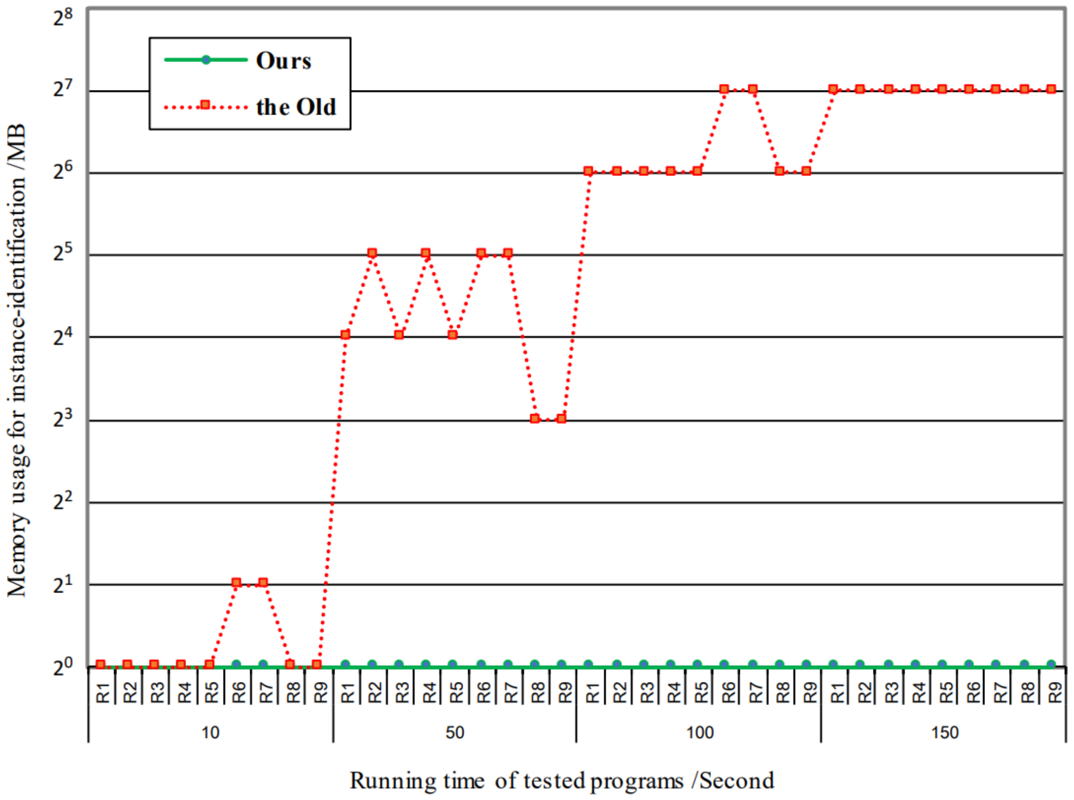


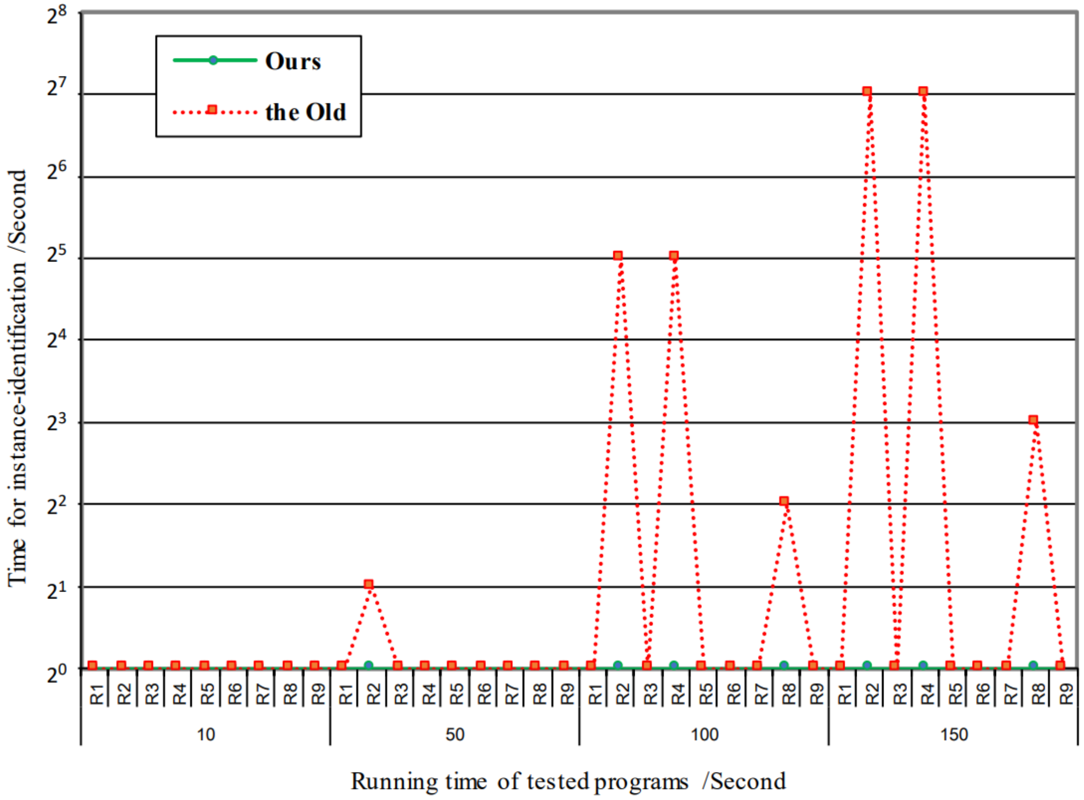
TABLE II: Subject programs and running settings

Subject	**RunGroup No.**	**Sampling period (ms)**	**Node Monitored**
Sub1	R1	100	Source node
	R2	20	Source node
Sub2	R3	100	Source node
	R4	20	Source node
Sub3	R5	Default of Avrora	Source node
Sub4	R6	100	Intermediate node
	R7	20	Intermediate node
Sub5	R8	Set by TestCTP	Benign node
	R9	Set by TestCTP	Buggy node

Experiment results



(a) Space overhead



(b) Time overhead

Improvement reasoning

- Old tool cannot identify all the execution points at real-time, so it has to utilize a list data structure to keep the information. When running time increased, list size will keep increasing and thus RAM cost increased. For time overhead, list-searching operation is time-consuming.
- Proposed algorithm is real-time, which avoids the list data structure and list-searching operation. Also the experiment shows that the theoretical analyses of proposed algorithm on time and space complexity are consistent with the results.