J. Parallel Distrib. Comput. 102 (2017) 132–148

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

ReLog: A systematic approach for supporting efficient reprogramming in wireless sensor networks

Xiaorui Zhu[a,*], Xianping Tao[a], Tao Gu[b], Jian Lu[a]

[a] State Key Laboratory for Novel Software Technology, Nanjing University, China
[b] School of Computer Science and Information Technology, RMIT University, Australia

HIGHLIGHTS

- We propose a systematic approach to support efficient reprogramming of WSN platform.
- The programming language naturally supports loosely coupled programs.
- The compiler significantly reduces the size of data for reprogramming.
- The VM reduces the additional energy consumption incurred by interpretive execution.
- We have evaluated the approach through real reprogramming cases.

ARTICLE INFO

Article history:
Received 29 September 2015
Received in revised form
13 July 2016
Accepted 5 December 2016
Available online 13 December 2016

Keywords:
Wireless sensor network
Reprogramming efficiency
Logical programming language
Compiler
Virtual machine

ABSTRACT

Wireless sensor networks are shifting to application platforms that poses several challenges on reprogramming efficiency. To better support the efficient reprogramming, this paper proposes a systematic approach named ReLog which consists of a programming language, a compiler, and a virtual machine. To make application programs concise and easy to modify, the ReLog language extends from a traditional logical programming language and makes the extension part have the similar coding style. To reduce the size of data for reprogramming, the compiler first produces extremely compact executable code by compiling application programs into high-level representations. It also implements efficient incremental reprogramming to diminish differences between the current and new executable code. To mitigate the energy consumption incurred by interpretive execution, the virtual machine optimizes the executable code as well as the execution process to improve the runtime efficiency. We have implemented ReLog and evaluated it with respect to real reprogramming cases. Our experimental results show that it is easy to modify ReLog programs to satisfy new application requirements. Meanwhile, the compiler reduces the size of executable code by 61.4%–83.2% compared to the existing work. In addition, the lifetime of sensors running the ReLog virtual machine is close (97.04%–98.31%) to that running the native code.

1. Introduction

Wireless sensor networks (WSNs) have been successful applied in a variety of applications [27]. In recent years, requirements of WSN applications become more and more complex. Sensors in the applications often have different functions or play different roles [2,1,5,6]. Meanwhile, WSNs of the applications are shifting from application-specific networks to platforms according to the following observations. First, multiple applications could be deployed in a same physical place.[1] Second, applications of the same physical place often have some common requirements on WSN such as sensing ability, network size, and even topology. Therefore, we can extend an existing WSN to support new applications rather than deploying completely new WSNs.

A WSN platform may support multiple applications during its lifetime. Therefore, it may change application code on sensors (i.e., reprogramming) more frequently due to requirements of both

[1] Examples include vehicle classification [2] and vehicle weight estimation [1] in a highway, canopy closure estimation [24], navigation [21], and fire detection [38] in a forest, energy consumption identification [11] and energy management [10] in a building, etc.

* Corresponding author.
E-mail addresses: zxr@smail.nju.edu.cn (X. Zhu), txp@nju.edu.cn (X. Tao), tao.gu@rmit.edu.au (T. Gu), lj@nju.edu.cn (J. Lu).

http://dx.doi.org/10.1016/j.jpdc.2016.12.010
0743-7315/© 2016 Elsevier Inc. All rights reserved.

deploying new applications and updating existing applications. Additionally, the size of data for reprogramming[2] could be large in the cases of deploying new applications. Meanwhile, a WSN platform usually consists of heterogeneous sensors to support complex application requirements. However, the heterogeneity of sensors necessitates multiple scripts even for one reprogramming task.[3] These features give rise to challenges of the reprogramming efficiency. The first challenge rises from *energy efficiency*. Sensors of WSN platform are usually powered by batteries. Excessive energy consumption of sensors due to reprogramming with large scripts will shorten lifetime of the platform. The second challenge lies in *latency*. Sensors of WSN platform need to stop the executing application completely to dedicate their resources for reprogramming. Reprogramming with large scripts will increase the latency of reprogramming process and consequently reduce the availability of WSN applications, while high availability is expected by many applications [5,6,21,38]. These two challenges require efforts to increase the reprogramming efficiency of WSN platform.

There are many efforts to address these challenges by reducing the size of script. Part of them leverage on *incremental reprogramming*. The basic idea of incremental reprogramming is to transfer the *delta* to the (binary) code so that energy consumption and latency can be minimized. For example, incremental linking [16], Hermes [28], and R3 [8] use the difference between the old and new binary code to generate the delta, while dynamic linking [9] and RemoWare [33] leverage on loadable modules which contain modifications in the application program. These efforts work well when reprogramming aims to update applications on homogeneous sensors. However, reprogramming of WSN platform may need to deploy new applications on heterogeneous sensors. In this case, the existing efforts may lose their efficiency due to large deltas to be transferred.

A promising idea to address this problem is to introduce the *virtual machine* (VM). The executable of VM is much smaller than binary code. In addition, VMs can shield the heterogeneity of sensors and make the executable same for all sensors. These features help to reduce the size of script significantly. Efforts such as Darjeeling [4] and Java Card [32] provide generic VMs for resource-constrained sensors and have potential to support the efficient reprogramming of WSN platform. However, these efforts are not specially designed for this purpose. (1) The imperative programming languages used in these efforts require professional skill and enough experience to produce loosely coupled programs. Users (mainly the domain experts) need to design the programs elaborately to tackle changes of application requirement with less modifications. (2) The executables generated by compilers in these efforts are not compact enough since they are composed of machine-level operation details. In addition, these compilers take no endeavor (such as incremental reprogramming) to further reduce the size of script. (3) Energy consumption of VM is important for the lifetime of WSN platform. The main purpose of these efforts is to provide feature-rich VMs for resource-constrained sensor platforms rather than reducing energy consumption incurred by interpretive execution.

To address aforementioned problems, we propose a systematic approach named ReLog which is specially designed to support the efficient reprogramming of WSN platform. The proposed approach consists of three components including a *programming language*, a *compiler*, and a *VM*. (1) The ReLog language extends from a traditional logical programming language and makes the extension part keep the similar coding style. This choice makes the ReLog language naturally support concise and loosely coupled programs. (2) The ReLog compiler aims to reduce the size of script in both cases of deploying new applications and updating existing applications. It can produce extremely compact executables which are composed of high-level representations without containing tedious machine-level instructions. To further reduce the script size, it implements efficient incremental reprogramming with methods of SGN (system-generated name) based compiling and executable rearrangement which diminish the difference between executables. (3) The ReLog VM focuses on reducing the energy consumption through various optimizations. It provides a generic optimization on the executable to improve the execution efficiency. It also adopts methods of program-directed reasoning and adaptive payload to optimize the execution process for any specific application.

In summary, this paper makes the following contributions.

- We propose a systematic approach named ReLog which includes a programming language, a compiler, and a VM to better support the efficient reprogramming of WSN platform. We have implemented ReLog and evaluated it with respect to real reprogramming cases.
- The ReLog language naturally supports concise and loosely coupled programs. Experimental results show that the ReLog programs are almost line-by-line translations of application requirements. Meanwhile, it is easy to modify a ReLog program to satisfy new application requirements without incurring modifications of irrelevant parts of the program.
- The ReLog compiler is specially designed for reducing the size of script. Experimental results show that it reduces the size of executable by 61.4%–83.2% compared to Darjeeling [4]. In addition, the optimized delta generation of the compiler further reduces the size of script by 22.7%–49.6%.
- The ReLog VM takes various optimizations to improve the runtime efficiency. Experimental results show that it only increases the CPU energy consumption by 1.74%–4.08% compared to TinyOS [20]. Additionally, the lifetime of sensors running the ReLog VM is 97.04%–98.31% of that running the native code.

The rest of the paper is organized as follows. Section 2 surveys the related work. Section 3 gives an overview of the proposed approach. Sections 4–6 describe the details of the programming language, the compiler, and the VM, respectively. Section 7 presents our evaluation results, and Section 8 concludes the paper and discusses our future work.

2. Related work

We discuss the related work from two perspectives of programming language and reprogramming approach to justify our key design choices.

2.1. Programming language

Many existing efforts aim at generating concise programs of WSN applications. The basic idea in these efforts is to provide high-level programming abstractions to shield system-level implementation details.

Regiment [26] and Abstract Regions [36] use tuple-space-like abstractions to facilitate data sharing and aggregation within a group of sensors. TinyDB [22] and AQL [35] provide database-like abstractions to facilitate data filtering and collection. μSETL [13] introduces the set abstraction to simplify data operations among sensors. However, these efforts pay less attention to make programs easy to modify. Changes to application requirement may

2 We call the data for reprogramming script in this work.

3 This is because the executable code on sensors usually consists of entangled OS and application code. The OS code for different types of sensors is different.

incur overhead of modifying irrelevant parts of the program. Different from these efforts, the ReLog language inherits the logical programming paradigm and naturally supports loosely coupled programs. Snlog [7] also extends from a traditional logical programming language. However, it allows users to implement a predicate with an external module which is written with an imperative language. Different from snlog, the ReLog language chooses to make the coding style of the extension part be similar with that of the logical part. This choice facilitates easy modifications of ReLog programs. To better support the efficient reprogramming, we make some important improvements on the previous version of the ReLog language [40]. Particularly, we allow users to specify attributes of a predicate when declaring (rather than using) the predicate. We also make the coding style of the extension part uniform and similar to that of the logical part. These improvements make the ReLog programs more concise and easy to modify.

2.2. Reprogramming approach

Efficient reprogramming of WSN applications has been an important research theme in the community for many years. Many existing efforts aim to improve the reprogramming efficiency by reducing the size of script. These efforts can be divided into two categories. The first category adopts the idea of incremental reprogramming, while the second category leverages on VMs. We give the detailed discussions of each category as follows.

2.2.1. Incremental reprogramming

Many efforts improve the efficiency of updating binary code on sensors through incremental reprogramming. These efforts can be divided into two classes.

Efforts in the first class implement the incremental reprogramming by comparing binary code directly. Incremental network reprogramming [15] uses block-level comparison to generate the delta between the old and new binary code. However, it did not consider the problem of code (functions and global variables) shift which may significantly increase the size of delta. Incremental linking [16] tries to solve the function shift by providing a slop after each function. A function can use its slop to grow without changing its location. However, the efficiency of this method is subject to the room of slop. Hermes [28] adopts the indirection table to mitigate the effects of function shift. Additionally, it also solves the global variable shift by pinning down global variables to their existing locations. Different from Hermes, R3 [8] solves the code shift problem by replacing physical addresses in binary code with symbolic indexes. Efforts in the second class implement the incremental reprogramming through loadable modules. Dynamic linking [9] and FlexCup [23] use binary code of the changed module as the delta. The binary code contains a symbol table and a relocation table to support dynamic linking of the module on sensors. However, the size of module is usually large. To address this problem, RemoWare [33] optimizes the binary code of a module by removing useless sections (e.g., debugging section) as well as headers of reserved sections.

The efficiency of these efforts is subject to the difference between the old and new binary code. Therefore, in the case of deploying new applications, the large size of script makes these efforts inefficient. Different from these efforts, ReLog leverages on the compact executable of VM to efficiently address this problem.

2.2.2. Virtual machine

Virtual machines can abstract underlying sensor platforms and provide much smaller executables. Therefore, they have potential to enable the efficient reprogramming of WSN platform.

Mate [18] and ASVM [19] provide application-specific VMs which are optimized for a specific problem domain. These VMs provide instructions abstracting from common operations of the problem domain. VM* [17] proposes synthesis of the VMs tailored for specific applications. Application-specific VMs can support highly compact executables. However, the VMs themselves often need to update when deploying applications of other problem domains on the WSN platform. Since VMs are implemented by binary code which often entangles with the OS code on sensors, updating the VMs usually requires large scripts and significantly reduces the reprogramming efficiency. Different from these efforts, ReLog provides a generic VM to support a wide range of WSN applications. This choice allows ReLog to improve the reprogramming efficiency by only focusing on the application layer.

Java Card [32] and Darjeeling [4] are also generic VMs. Java Card is specially designed for resource-constrained devices. It proposes a modified instruction set to execute Java programs on these devices. Darjeeling proposes a more powerful VM by adding features such as multi-threading and efficient garbage collection. However, these efforts are not specially designed for the efficient reprogramming of WSN platform. First, it is not easy for novice programmers to write loosely coupled programs with these efforts. Second, executables of these efforts are not compact enough since they are composed of machine-level operation details. In addition, these efforts take no endeavor to further reduce the size of script. Third, these efforts are not specially optimized for reducing additional energy consumption incurred by interpretive execution of VM. Different from these efforts, ReLog leverages on the logical programming paradigm to naturally support concise and loosely coupled programs. Meanwhile, it focuses on reducing the script size by generating extremely compact executables and implementing efficient incremental reprogramming. In addition, ReLog takes several optimizations on the executable as well as the execution process to reduce the energy consumption of VM at runtime.

3. Overview

To provide the efficient reprogramming of WSN platform, we propose a systematic approach named ReLog which combines a programming language, a compiler, and a VM. Fig. 1 gives an overview of ReLog and the main concerns of each component.

The ReLog language mainly focuses on the following two aspects. (1) As a language for WSN platform, it provides suitable constructs to program a wide range of WSN applications. The ReLog language considers a WSN application as the combination of data processing and system operations. It uses constructs of *fact* and *rule* from the traditional logical programming language to deal with data processing, and extends with constructs of *event* and *action* to handle system operations. These constructs are suitable for programming WSN applications. (2) As a language for reprogramming, it also facilitates to modify application programs. On one hand, the constructs of the ReLog language shield system-level implementation details. On the other hand, the organization of these constructs inherits the loose coupling nature of the logical programming language. These features make application programs easy to modify.

The compiler in ReLog endeavors to reduce the size of script by focusing on the executable design and the delta generation. (1) To make the executable compact, we take the compactness as the main principle of the executable design. In particular, the compiler chooses to compile application programs into extremely high-level intermediate representations which contain no detail about machine-level operations. This design choice significantly reduces the size of executable. (2) To reduce the size of delta,

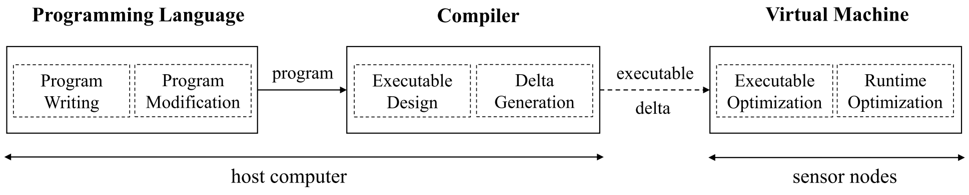


Fig. 1. Overview of the ReLog approach.

we aim to reduce the difference between the old and new executables. Since the difference may be increased due to changes of system-generated names and code shift, the compiler introduces *SGN-based compiling* to make the same predicate have the same system-generated name in different executables. Meanwhile, it adopts *executable rearrangement* to make the unchanged part of different executables be in the same location.

The VM in ReLog improves runtime efficiency from the following two aspects. (1) It first optimizes the executable to make it support efficient execution. Since the compiler focuses on the compactness of executable, it encodes some information required by the execution process implicitly in the executable. Therefore, the VM optimizes the executable by extracting the information and storing it explicitly with additional fields and lists. This optimization significantly improves the execution efficiency. (2) The VM also provides application-specific optimizations at runtime. Since the VM needs to support different applications, it has to adopt a generic reasoner and predefined payloads which may be not efficient enough for a specific application. To address these problems, the VM first takes *program-directed reasoning* which allows users to guide the reasoning process for an application. It also introduces *adaptive payload* to provide application-specific payloads according to the requirement of the executing application.

4. Programming language

The ReLog language aims to support programming of popular WSN applications and make modifications of these programs easily. According to surveys on WSN applications [27,25], we found that most of the applications consist of uncomplicated data processing and limited system operations. The ReLog language is designed to target these applications.

Logical programming paradigm promotes clean and concise specifications of application requirements and leads to code that is significantly easier to specify and adapt. Consequently, logical programming languages have been successfully applied for developing WSN applications [7,29,37,12]. Datalog is a well-known logical programming language. Meanwhile, it has shown its potential to deal well with data processing of WSN applications [7,29]. Therefore, the ReLog language chooses to extend from Datalog. However, designing the extension part of the language bears the following challenges. (1) The first challenge rises from the suitable way to describe the system operations. To address this challenge, we analyze WSN application requirements and provide the constructs of *event* and *action* for users to program the system operations in a familiar *event-driven* way. (2) The second challenge lies in the management of application data (i.e., facts in logical programs). Since data in WSN applications is often highly dynamic, it may place a burden for users to implement the fact management manually in the program. Therefore, we provide attributes which help to manage facts automatically according to predefined policies. (3) The third challenge is the consistency of coding style. The extension code needs to keep the features of conciseness and ease-of-modification. Therefore, we make the constructs in the extension part contain no system-level implementation detail and organize these constructs in a loosely coupled structure.

Listing 1 An example of the ReLog program

```
# sys_dutyCycle = 10

Predicate:
    address@unique.
    seqNum@unique.
    message@volatile:t1.
    reading@volatile:t1.

Clause:
    address(sys_nodeID).
    seqNum(0).
    seqNum(N+1) :- seqNum(N)@passive, reading(Value).
    message(Src, N, Value):- reading(Value)@passive, address(Src)@passive,
seqNum(N).

Shell:
    boot() → setTimerMilli(t1, 5000, sys_infinity).
    t1() → insert(reading(sense(sys_thermometer))).
    generate(message(Src, N, Temperature)) → send(1, <Src, N,
Temperature>).
    receive(2, <NewAddr>) → insert(address(NewAddr)).
```

We illustrate the ReLog language [30] by going through an example of the ReLog program shown in Listing 1. This program describes an application which collects temperature values of the environment. Each sensor in this application samples and sends back a temperature value along with the sensor's ID and a sequence number every 5 s.

4.1. Introduction to the relog language

A typical ReLog program consists of three parts including predicate, clause, and shell. The constructs in these parts are feasible to describe common application requirements. However, they cannot provide customized configurations such as communication channel and duty cycle. To increase the flexibility, the ReLog language allows users to specify some configurations through optional annotations.

Annotations used for specifying configurations should be at the beginning of the program. They all start with the symbol '#' and have parameters with the same prefix of '*sys_*'. For example, the annotation in the first line of the example indicates that the sensor sets duty cycle to 10%.

4.1.1. Predicate

Unlike traditional logical programming languages, the ReLog language requires users to declare predicates explicitly. This is because these explicit declarations can help users to manage facts of the predicates automatically. Each declaration consists of two parts: a predicate name which is syntactically an identifier beginning with a lower-case letter, and an optional *attribute* which specifies the policy of the fact management.

Due to memory limitation in sensors, the out-of-date facts should be cleared from time to time. It is difficult for users to manage facts of each predicate manually. In WSN applications, we observe that most of the application data has predicable lifetime. For example, a temperature value sensed from an environment is only valid in its sampling period. This observation inspires us to create some predefined data management policies through attributes. Particularly, (1) the *@unique* attribute (lines 4 and 5) indicates that a fact will be deleted if a new fact of the same predicate comes. (2) The *@volatile* : ID_{timer} attribute (lines 6 and 7) indicates that all facts of a predicate will be deleted if the associated timer fires.

4.1.2. Fact and rule

Before introducing fact and rule, we first give definitions of constant, variable, term, and atom. A *constant* is either an integer, a real number, or a system-defined string with the prefix of '*sys_*'. A *variable* is an identifier which begins with an up-case letter. A *term* is either a variable, a constant, or a compound term of a function or an arithmetic expression which uses terms as parameters, while an *atom* is a predicate with parameters of terms.

The ReLog language uses facts and rules to deal with data processing in WSN applications. Facts are used to represent application data. A fact is an atom with all parameters of constants. For example, the fact *address*(*sys_nodeID*) (line 10) represents that the address of a sensor in the application is the sensor's ID. Rules are used to implement data calculations. A rule consists of a deduction symbol (:-), an atom on the left side of the symbol called the head, and one or more atoms on the right side called the body. The body defines some preconditions, which if true, instantiates the head to a fact. For example, the rule in line 13 creates a fact of *message* if there exist the facts of *reading*, *address*, and *seqNum*. A predicate in the body of a rule may contain the *@passive* attribute (lines 12 and 13), which indicates that the arrival of new facts of this predicate will not trigger the evaluation of the rule.

4.1.3. Event and action

The ReLog language uses events and actions to handle system operations. Each statement in the shell part consists of an event and one or more actions that appear on the left side and right side of the statement's triggering symbol ('→'), respectively. The action(s) will be triggered to execute if the associated event occurs. For example, the statement in line 18 indicates that if a new fact of *message* arrives, the content of the fact will be sent to the base station. Each variable in the action will be initialized with the value of the variable with the same name in the event.

There are four types of events in the ReLog language. (1) The *boot* event (line 16) occurs when the execution starts. This event facilitates the initialization of the execution. (2) The *timer* event (line 17) occurs when the associated timer fires. The timer's ID is used to differentiate different timer events. (3) The *fact generation* event (line 18) occurs when a new fact of the predicate in the event is generated. The variables of the atom in the event will be initialized with the constants in the newly generated fact. (4) The *message receiving* event (line 19) occurs when an associated message is received. This event consists of two parts. The first part is the message type which is used to differentiate different types of messages, while the second part indicates the payload in the received message.

Various actions are provided for users to program WSN applications. These actions can be roughly divided into two classes. (1) Actions in the first class manipulate various devices in a sensor such as timer, radio, sensing devices, and serial ports (e.g., ADC and GPIO). The ReLog language provides multiple actions for a device to facilitate the programming. For example, two *send* actions are provided to send messages through the radio. The first one (line 18) only requires the message's payload. It will send the message back to the base station through the built-in routing protocol. The second one has one more parameter of an address. It will send the message directly to a neighboring sensor with the address. (2) Actions in the second class manipulate facts of a program. For example, the *insert* action (lines 17 and 19) inserts a fact to the fact repository.

To maintain the consistency of coding style, events and actions in ReLog programs contain no system-level implementation detail. In addition, the statements in the shell part are almost uncoupled with each other and independent of their orders. Meanwhile, the coupling between these statements and clauses (facts and rules) is the same as that among clauses. These features ensure that the code in the extension part keeps concise and easy to modify.

4.2. Illustration of the program

We now walk through the program in Listing 1 to explain how the program works. When the execution starts, the *boot* event occurs and triggers a timer setting action. This action sets a timer *t*1 which will fire every 5000 ms continuously (line 16). When *t*1 fires, the *insert* action uses the temperature sensing value to generate a *reading* fact and inserts it to the fact repository (line 17). The *reading* fact triggers the evaluation of the first rule to generate an *seqNum* fact which will then trigger the evaluation of the second rule to generate a *message* fact (lines 12 and 13). When the *message* fact is generated, the *fact generation* event occurs and triggers the *send* action to send the content of the fact to the base station (line 18). If a sensor receives a message of type 2, the *message receiving* event occurs and triggers the *insert* action to generate a new *address* fact and replace the current one in the fact repository (line 19).

5. Compiler

To increase the reprogramming efficiency, the ReLog compiler aims to reduce the size of both executable and delta. The main challenge to achieve compact executables is the universality of the executable design. Since a WSN platform is supposed to support different WSN applications, the design needs to keep executables compact for all these applications. To address this challenge, the ReLog compiler generates the executable by translating the constructs of a program (i.e., predicates, variables, constants, etc.) into internal representations including their names and special attributes. Therefore, the size of the executable is only determined by the scale of the application program. Since ReLog programs are usually concise, this design works well for WSN applications.

There are two challenges to reduce the size of delta. (1) The first challenge rises from changes of system-generated names. Predicates have system-generated names in the executable. For the same predicate in two programs, the change of the predicate's system-generated name often incurs a large range of differences between the two executables. To reduce these differences, the compiler uses *SGN-based compiling* to ensure the same predicate has the same system-generated name in different executables. (2) The second challenge lies in code shift. Updating a program may incur severe code shift in the executable. To address this challenge, the ReLog compiler first divides the executable into several lists. Changes in one list will not incur code shift in other lists. In addition, it also introduces *executable rearrangement* to pin down the unchanged part of the executable in the original location.

We now give an overview of the ReLog compiler as shown in Fig. 2. The compiler contains three components including a lexical analyzer and a syntax parser, an executable generator, and a delta generator. If reprogramming aims to deploy a new application, the compiler will compile the new program into the executable directly. Otherwise, the compiler will generate a delta according to executables of the current and updated programs. We discuss these two tasks in detail as follows.

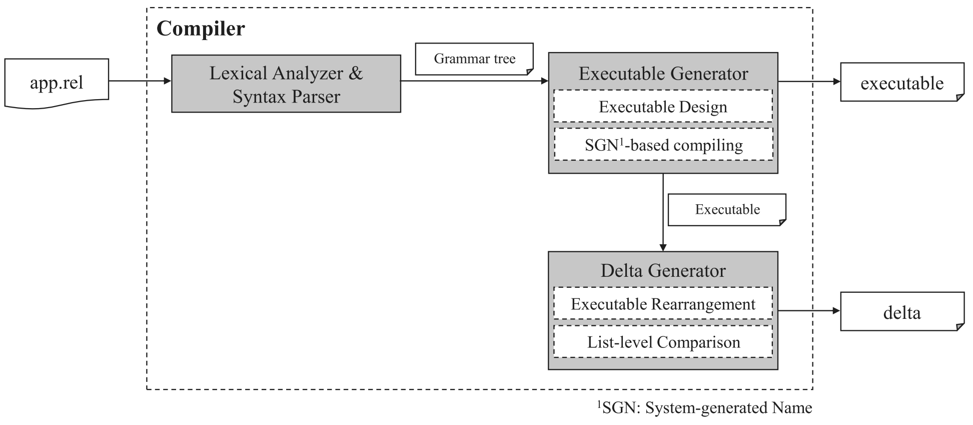
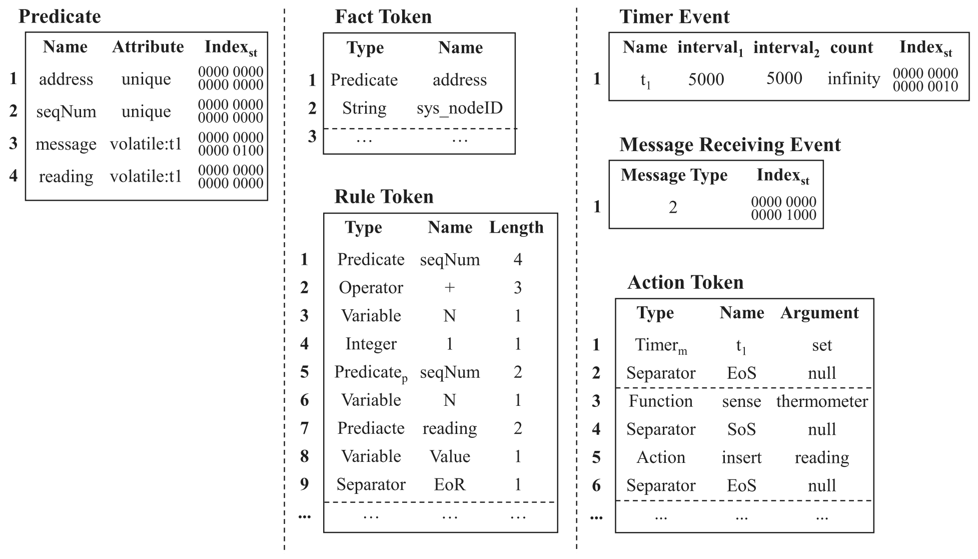


Fig. 2. Overview of the ReLog compiler.



Predicate

	Name	Attribute	$Index_{st}$
1	address	unique	0000 0000 0000 0000
2	seqNum	unique	0000 0000 0000 0000
3	message	volatile:t1	0000 0000 0000 0100
4	reading	volatile:t1	0000 0000 0000 0000

Fact Token

	Type	Name
1	Predicate	address
2	String	sys_nodeID
3	…	…

Rule Token

	Type	Name	Length
1	Predicate	seqNum	4
2	Operator	+	3
3	Variable	N	1
4	Integer	1	1
5	$Predicate_p$	seqNum	2
6	Variable	N	1
7	Prediacte	reading	2
8	Variable	Value	1
9	Separator	EoR	1
…	…	…	…

Timer Event

	Name	$interval_1$	$interval_2$	count	$Index_{st}$
1	t_1	5000	5000	infinity	0000 0000 0000 0010

Message Receiving Event

	Message Type	$Index_{st}$
1	2	0000 0000 0000 1000

Action Token

	Type	Name	Argument
1	$Timer_m$	t_1	set
2	Separator	EoS	null
3	Function	sense	thermometer
4	Separator	SoS	null
5	Action	insert	reading
6	Separator	EoS	null
…	…	…	…

(a) Predicate. (b) Clause. (c) Shell.

Fig. 3. Illustration of the executable.

5.1. *Executable design*

Fig. 3 illustrates fragments of the executable for the program in Listing 1.[4] Similar to the program, the executable mainly contains three parts including predicate, clause, and shell.

5.1.1. *Predicate*

The predicate part has only one list which is shown in Fig. 3(a). In this list, each predicate has an item which records its name, attribute, and statement index ($index_{st}$). The attribute is used to specify the fact management policy. It is stored with a 8-bit integer. The first 3 bits indicate the type of the attribute, while the last 5 bits indicate the timer's ID if necessary. The $index_{st}$ of a predicate indicates indexes of the shell statements which contain fact generation events of this predicate. We use a 16-bit bitmap to represent the $index_{st}$ of a predicate. Specifically, the i_{th} bit represents the i_{th} shell statement, and value 1 represents the shell statement which contains the fact generation event of this predicate. For example, the $index_{st}$ value of the predicate *message* is 0000 0000 0000 0100, which means the third shell statement (line 18 of the program in Listing 1) contains the fact generation event of *message*. Whenever a new fact of *message* is generated, actions in this shell statement will be executed.

5.1.2. *Clause*

The clause part typically contains a fact token list and a rule token list as shown in Fig. 3(b). The fact token list records types and names of elements of each fact. For example, the fact *address*(*sys_nodeID*) has two elements including a predicate named *address* and a system-defined string of *sys_nodeID*.

The rule token list records types, names, and lengths of elements of each rule. The first two fields are the same as that of the fact token list. An exception is that a predicate in the rule token list may have two types including the *predicate* and the $predicate_p$. The latter one indicates the type of a predicate with the *@passive* attribute. The third field (length) of an element specifies its scope in the syntax. For example, the length of a variable or a constant is 1, while the length of a predicate or a function name should include lengths of all the parameters. Different from the fact token list, the rule token list introduces a *separator* named *EoR* (end of rule) at the end of each rule to separate rules. With length attributes and EoR separators, the VM can restore the syntax structure of each rule efficiently.

[4] For ease of understanding, we do not use internal representations (pure integers) in Fig. 3.

5.1.3. Shell

The shell part includes an action token list and at most two event lists as shown in Fig. 3(c). In the timer event list, each timer has an item which records its name, parameters, and statement index ($index_{st}$). The first two parameters ($interval_1$ and $interval_2$) specify the interval value of a timer. If values of these two parameters are equal, they represent a fixed interval. Otherwise, they represent the lower and upper bounds of a random interval, respectively. The third parameter (*count*) specifies the number of times a timer will fire. The *sys_infinity* value indicates that a timer will fire continuously. The $index_{st}$ of a timer indicates indexes of the shell statements which contain actions to be executed if the timer fires. The message receiving event list records message types and statement indexes ($index_{st}$) of messages. Each item of this list indicates indexes of the shell statements which contain actions to be executed when a certain type of message is received. Note that statement indexes in these two lists are represented by 16-bit integers just as which in the predicate list.

The action token list records types, names, and optional arguments of elements of each action. Similar to the rule token list, the action token list uses a *separator* named *EoS* (end of statement) to separate actions of different statements. Another special *separator* named *SoS* (suspension of statement) is introduced after each *sense* function. It will suspend execution of the statement until the sensing value is returned.

We propose several optimizations in the ReLog compiler to reduce the code size of the shell part. (1) It deletes the boot event list by making actions triggered by the boot event be at the beginning of the action token list. In addition, it deletes the fact generation event list and merges its content (i.e., statement indexes of predicates) to the predicate list. (2) It removes the *length* field in the action token list. This is because actions in the shell part only have parameters of variable, constant, and built-in function. The VM does not need the length of element to restore the syntax structure. (3) It moves parameters of timers from the action token list to the timer event list. This choice can reduce the space for storing these parameters[5] and share parameters of each timer to all actions in the action token list. (4) It compresses some tokens in the action token list by adding the field of *argument*. For example, we can use one token '*Function sense thermometer*' to represent the *sense* function and its parameter. Although some elements (e.g., variables, constants) have no argument, the compression can often reduce the size of the action token list since compressible actions and functions occur frequently in ReLog programs.

5.2. Delta generation

The ReLog compiler will generate deltas if reprogramming aims to update existing applications. The delta generation contains three important steps including *SGN-based compiling*, *executable rearrangement*, and *list-level comparison*.

5.2.1. SGN-based compiling

In ReLog programs, a predicate may appear in all of the predicate part, the clause part, and the shell part. This implies that any change in the predicate's system-generated name may incur a large range of modifications in the executable.

The traditional compiling cannot guarantee to assign the same system-generated name to a predicate which appears in both the current and the updated programs. We use Fig. 4(a) to illustrate this issue. Suppose the current program has three predicates of *predicate1*, *predicate2*, and *predicate3*. The system-generated names of these predicates are *p1*, *p2*, and *p3*, respectively. If the updated program deletes *predicate1* and adds a new predicate of *predicate4*, the traditional compiling will assign the names of *p1*, *p2*, *p3* to the predicates of *predicate2*, *predicate3*, and *predicate4*, respectively. As a result, the same predicates (*predicate2* and *predicate3*) have different system-generated names in executables of the two programs.

To address this problem, we introduce the SGN-based compiling to compile different versions of an application program. In particular, when compiling a new application program, the compiler creates a global SGN list which contains mappings of predicates in the program and theirs system-generated names in the executable. When compiling an updated version of the program, the compiler searches the global SGN list before assigning a name to a predicate. If the list has recorded the predicate, the compiler assigns the system-generated name in the list to the predicate. Otherwise, the compiler first generates a new name different from all the names in the list and assigns this name to the predicate. After that, the compiler updates the global SGN list by adding a mapping of this predicate and its system-generated name. As a result, the compiler can guarantee that the same predicate has the same system-generated name in executables of different versions of an application program. Fig. 4(b) gives an example of the SGN-based compiling. In this example, the compiler searches the global SGN list and assigns *p2* and *p3* to predicates *predicate2* and *predicate3*, respectively. Since there is no record of predicate4 in the list, the compiler generates a new name *p4* (different from all the system-generated names in the list) and assigns it to *predicate4*. After that, the compiler updates the global SGN list by adding *predicate4* and its system-generated name *p4* to the list.

5.2.2. Executable rearrangement

Modifying a program may incur severe code shift in the rule token list and the action token list. The code shift often leads to a large range of differences between executables. We introduce *executable rearrangement* to address this issue and use the rule token list as an example to illustrate this method.

For rule token lists ($L_{current}$ and $L_{updated}$) of the two executables, executable rearrangement operates in the following steps. (1) It marks the unchanged rules in $L_{updated}$ by comparing $L_{current}$ and $L_{updated}$. (2) For these unchanged rules, it rearranges them in $L_{updated}$ according to their locations in $L_{current}$. This rearrangement often causes discrete blocks of free space in $L_{updated}$. (3) For other (changed) rules, it rearranges them with these free blocks by following the *best-fit* principle. That means it places each rule in the smallest block in which the rule will fit. If there exists any rule failing in the allocation, the compiler gives up the executable rearrangement.

An illustrated example is shown in Fig. 5. The current program in the example has three rules of *rule1*, *rule2*, and *rule3*, while the updated program modifies *rule1*, deletes *rule2*, and adds a new rule of *rule4*. In the initial arrangement of the rules in $L_{updated}$, *rule3* shifts due to the expansion of *rule1* and the deletion of *rule2* as shown in Fig. 5(a). In the optimized arrangement, the executable rearrangement keeps the location of *rule3* in $L_{updated}$ be the same as that in $L_{current}$. As a result, it solves the code shift problem as shown in Fig. 5(b).

5.2.3. List-level comparison

The ReLog compiler generates a delta by comparing lists in the old and new executables.[6] The delta usually consists of a

[5] This is because the timer event list does not need to record types and names of these parameters.

[6] We do not compare fact token lists since the fact token list is quite dynamic during the execution. Instead, we just replace the current fact token lists on sensors with the new one.

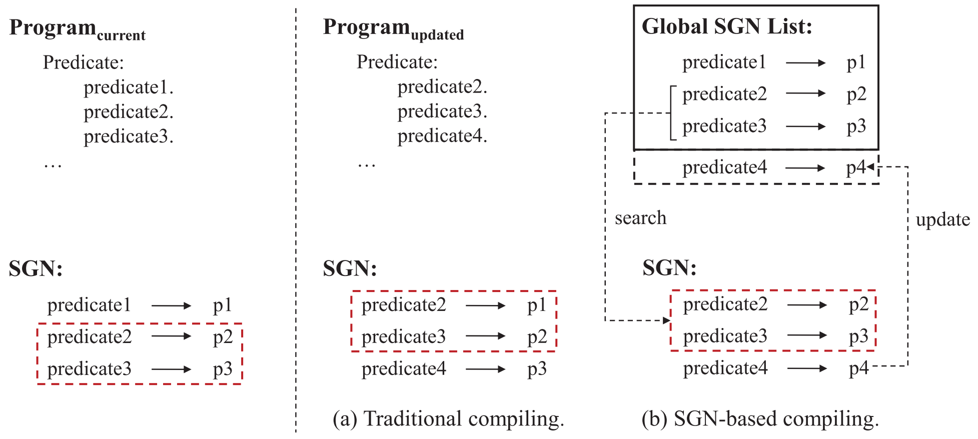


Fig. 4. Illustration of traditional compiling and SGN-based compiling.

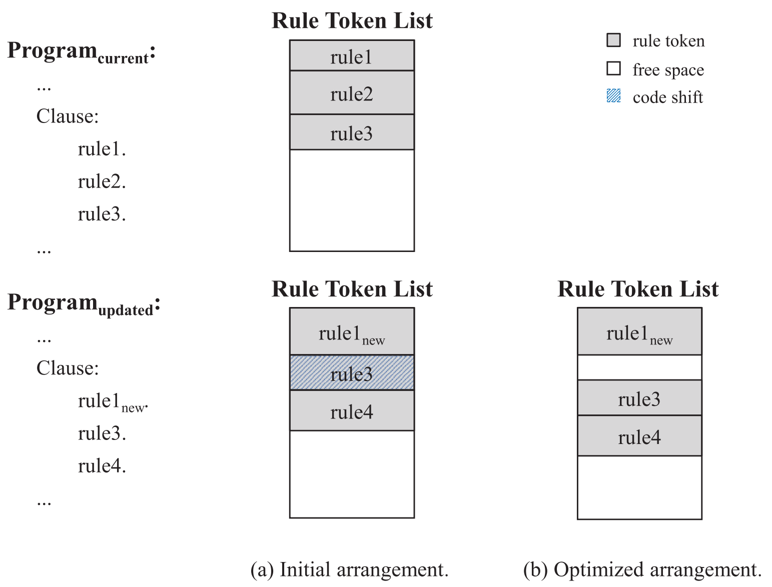


Fig. 5. Illustration of executable rearrangement.

header and several segments. The header contains information of the program's version and the length of the delta. Each segment consists of a preamble as well as reconstructing instructions (and their data). The preamble indicates the objective list, while the reconstructing instructions are used to update the list.

The compiler uses the *diff* algorithm to generate reconstructing instructions of each list. These instructions have the uniform form of $\langle type, index_{start}, index_{end}\rangle$. There are three types of instructions including *add*, *change*, and *remove*. The first two are used for adding new items to a list and changing the current items of a list, respectively. Following these instructions in a segment are their required data. The last type of instructions is used for removing current items from a list. Reconstructing instructions generated by *diff* are arranged reversely in a segment. This arrangement guarantees that execution of the current instruction will not affect indexes in unexecuted instructions.

6. Virtual machine

The ReLog VM aims to improve the runtime efficiency. There are three challenges we are facing in our design. (1) The first challenge rises from the inefficient data locating process. To reduce the size, the executable generated by the compiler contains only the most necessary information. This feature makes the data locating process full of list traversing operations, which incurs significant overhead. To address this challenge, we choose to optimize the executable by adding additional fields and lists to build links before execution. These links can help the execution process to locate data much more efficiently. (2) The second challenge lies in the unnecessary rule evaluations. The VM adopts a generic reasoner which has the complete logic for evaluating rules. This choice can reduce the size of executable since no logic of the rule evaluation is required in the executable. However, it may incur some unnecessary rule evaluations since the reasoning process is not optimized for any specific application. To address this challenge, we introduce *program-directed reasoning* which allows programmers to provide some directions for the reasoner. With these directions, the reasoner can customize the reasoning process to eliminate unnecessary rule evaluations. (3) The third challenge lies in the inefficient predefined payloads. Since payloads of WSN applications are unpredictable, the VM has to provide predefined payloads which are long enough to satisfy different applications. This choice may lead to idle fields in the predefined payloads for some applications. Therefore, the VM introduces adaptive payload which can remove the idle fields at runtime according to the executing application's requirement.

Fig. 6 gives an overview of the ReLog VM. The VM can be divided into two parts according to their functions. (1) The first part contains two components including an executable builder and an executable optimizer. They are used to provide

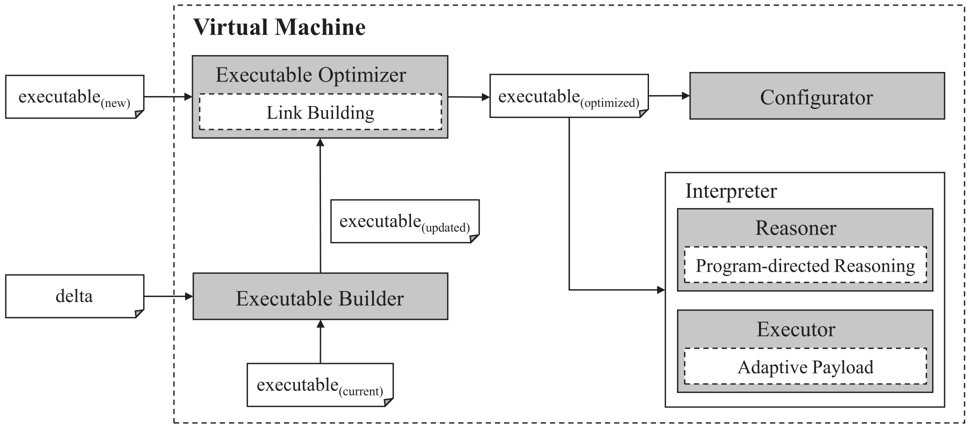


Fig. 6. Overview of the ReLog virtual machine.

optimized executables for the VM. (2) The second part consists of a configurator and an interpreter. They are responsible for running the optimized executables. We give detailed discussions of important components in these two parts as follows.

6.1. Executable optimizer

With the original executable, the VM has to locate data in lists through inefficient *traversing* operations. To address this problem, the executable optimizer builds links in the executable by adding additional fields and lists. These links can help the VM to locate data through efficient *addressing* operations. Fig. 7 illustrates the fragment of optimizations on the executable in Fig. 3. We give detailed explanations of these optimizations as follows.

During the rule evaluation, the reasoner needs to find out the facts of a predicate. With the original executable, the reasoner has to traverse the fact token list to locate these facts. To reduce the overhead, (1) the executable optimizer first adds a *fact* list to the executable. The fact list contains two fields: the *index* field indicates the location of a fact in the fact token list, while the *next* field is the index of the next fact of the same predicate in this list. After that, (2) the executable optimizer adds a new field of $index_f$ to the predicate list. The $index_f$ of a predicate indicates the index of the predicate's first fact in the fact list. With these optimizations, the reasoner can locate all facts of a predicate in the fact token list efficiently. For example, the predicate *seqNum* has a fact starting at the index of 3 in the fact token list. The reasoner can locate the fact using only twice addressing operations.

When a new fact comes, the reasoner needs to find out the rules to evaluate. With the original executable, the reasoner has to traverse the rule token list to locate the rules whose bodies contain the predicate associated with the fact. To reduce the overhead, (1) the executable optimizer first adds a *rule* list to the executable. The i_{th} item of this list indicates the starting index of the i_{th} rule in the rule token list. After that, (2) a new field named $index_r$ is added to the predicate list. The $index_r$ of a predicate (represented by a 16-bit bitmap) indicates indexes of items in the rule list. The i_{th} bit represents the i_{th} item and value 1 represents the target item. With these optimizations, the reasoner can locate the rules to be evaluated in the rule token list efficiently when a fact comes. For example, a new fact of the predicate *seqNum* will trigger evaluation of the rule starting at the index of 10 in the rule token list. With the optimized executable, the reasoner can locate this rule using only twice addressing operations.

When a new fact is generated, the executor needs to find out the shell statements to execute. With the original executable, the executor needs to traverse the action token list to locate these statements. To reduce the overhead, (1) the executable optimizer first adds a *statement* list to the executable. The i_{th} item of this list indicates the starting index of the i_{th} shell statement in the action token list. Meanwhile, (2) the $index_{st}$ of a predicate (represented by a 16-bit bitmap) changes to indicate indexes of items in the statement list. The i_{th} bit represents the i_{th} item and value 1 represents the target item. With these optimizations, the executor can locate shell statements in the action token list efficiently when a new fact is generated. For example, a newly generated fact of the predicate *message* will trigger execution of the shell statement starting at the index of 7 in the action token list. With the optimized executable, the executor can locate the statement using only twice addressing operations.

6.2. Interpreter

The optimized executable will then be sent to the interpreter to execute. As shown in Fig. 8, the interpreter consists of a reasoner and an executor.

The reasoner contains two components including an inference engine and a fact manager. The inference engine evaluates rules using the *forward chaining* method. This method starts the evaluation of a rule from existing facts to extract new facts until no new fact can be derived. Besides managing (i.e., inserting, replacing, and deleting) facts, the fact manager has two additional tasks. (1) It triggers the inference engine to evaluate rules according to newly generated facts and user-provided directions. (2) It calls the event handler to deal with fact generation events according to facts derived by the inference engine.

The executor consists of an event handler and an action executor. The event handler deals with four types of events including application booting, timer firing, message receiving, and fact generation. It triggers the action executor to execute particular actions according to different events. The action executor is responsible for executing various actions including calling the fact manager to process new facts.

We discuss important technical details of the interpreter as follows.

6.2.1. Execution mechanism

The execution process of a WSN application may be often interrupted by various events. It is not trivial for the interpreter to correctly deal with these interruptions. For example, suppose the reasoner needs to evaluate a rule $q(X)$:- $p(X, Y), X == Y$ with a fact $p(3, 7)$. It first needs to assign integers 3 and 7 to the variables X and Y of the atom $p(X, Y)$, respectively. After assigning 3 to the variable X, the reasoner is interrupted by a message receiving event. The event handling result may replace the current fact $p(3, 7)$ with a new fact $p(7, 3)$. Then the reasoner continues the

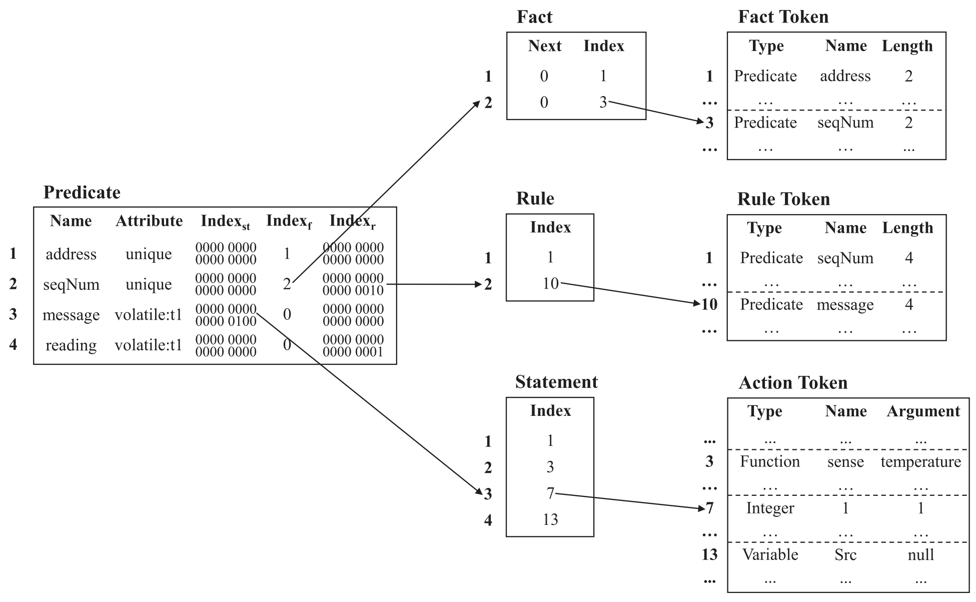


Fig. 7. Illustration of optimizing the executable through building links.

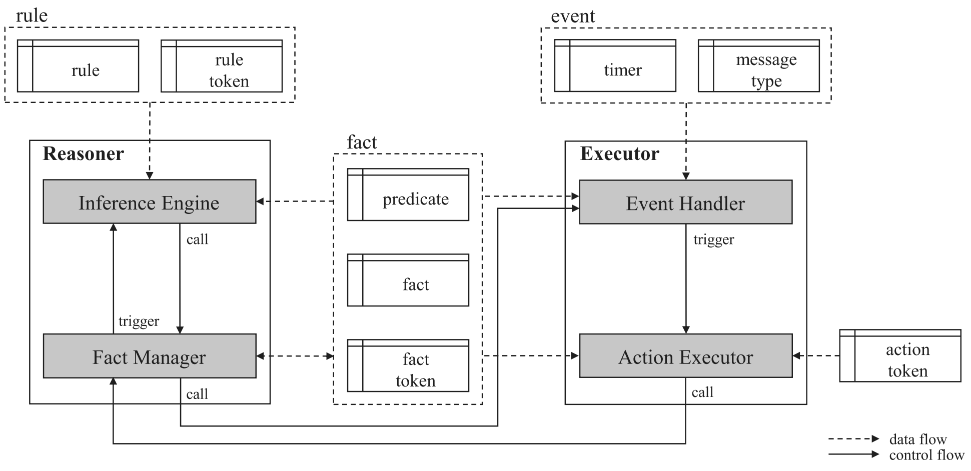


Fig. 8. Architecture of the interpreter.

evaluation with the new fact $p(7, 3)$ and assigns 3 to the variable Y. Since the values of X and Y are equal, an erroneous fact $q(3)$ is derived from this rule evaluation.

To address this problem, the interpreter leverages on the *task* mechanism to organize the execution. This mechanism has two important features. (1) Tasks are scheduled on the FIFO basis. A task is executed to the completion before another runs. (2) Events have higher priority than tasks. They can interrupt the execution of tasks. With this mechanism, the interpreter uses tasks to encapsulate rule evaluations, fact operations, and action executions. Events can be responded instantly since they can interrupt the execution of tasks. However, the interpreter will encapsulate actions triggered by these events as new tasks rather than executing actions immediately.

With the task mechanism, the fact replacing operation in the above example will not be executed during the rule evaluation. Therefore, this mechanism makes the execution process correct.

6.2.2. Program-directed reasoning

The reasoner may incur some unnecessary rule evaluations during the reasoning process. For example, suppose we have a rule $\min(X)$:- $value(X), \min(Y), X < Y$ for finding the minimum value and a unique fact $\min(5)$ indicating the current minimum value. If a fact $value(3)$ comes, the reasoner needs to evaluate the rule and a new fact $\min(3)$ will be derived. This new fact will trigger the reasoner to evaluate the rule again. However, it is obvious that the rule evaluation with the facts of $value(3)$ and $\min(3)$ will not generate a new fact. Therefore, this rule evaluation is unnecessary.

To address this problem, users can provide some directions in the program for the reasoner to optimize the reasoning process. Particularly, they can mark any predicate in the body of a rule with the *@passive* attribute to prevent unnecessary or unwanted evaluations. This attribute will stop the arrival of new facts of a predicate to trigger the evaluation of the rule. For example, we can add the *@passive* attribute to the rule as $\min(X)$:- $value(X), \min(Y)@passive, X < Y$. In this case, the new fact $\min(3)$ will not trigger the rule evaluation.

6.2.3. Adaptive payload

WSN applications usually have different requirements (i.e., formats and lengths) on payloads. Since these requirements are not predictable, the interpreter cannot provide payloads for applications in advance. A possible solution is to provide predefined payloads with common formats and sufficient space (e.g., the maximum length of payload). Therefore, we provide two predefined payloads. The first one contains 12 data fields of 16-bit integer.

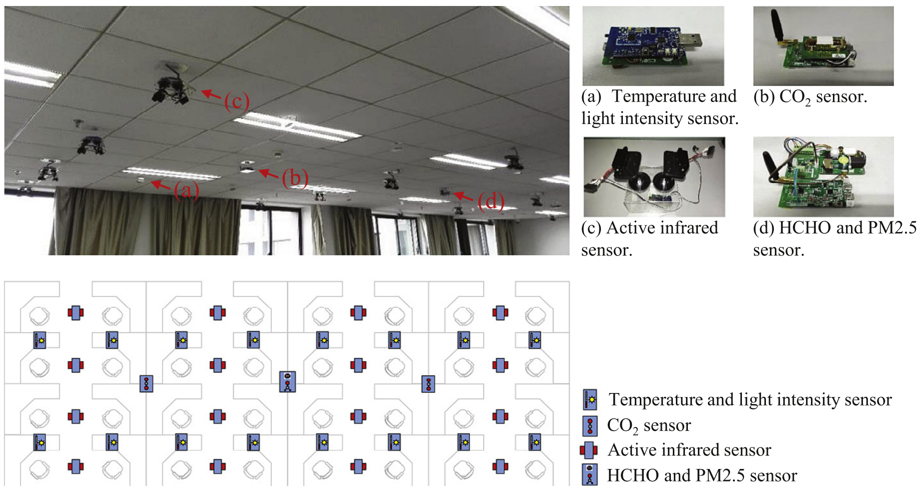

(c)
(b)
(a)
(d)
(a) Temperature and light intensity sensor.
(b) CO$_2$ sensor.
(c) Active infrared sensor.
(d) HCHO and PM2.5 sensor.
Temperature and light intensity sensor
CO$_2$ sensor
Active infrared sensor
HCHO and PM2.5 sensor

Fig. 9. Sensor network of the iLab application.

It can satisfy requirements of most of WSN applications such as transmitting sensing data, sequence numbers and node addresses. The second one contains 6 data fields of 32-bit integer. It can deal with special requirements of some applications. Message types are used to differentiate them. The message types from 0–99 indicate the first payload, while the message types from 100–199 indicate the second one.

Although this solution works, it often leads to many idle fields in the predefined payloads. For example, payload in the *send* action in Listing 1 uses only three fields. That means most of the fields are useless and waste considerable energy for transmission. To address this problem, the interpreter introduces the adaptive payload which can customize predefined payloads for a specific application. Particularly, it can remove idle fields in the predefined payloads at runtime according to the applications' requirements. With this method, the interpreter can provide appropriate payloads for applications.

7. Evaluation

We follows the goal-question-metric (GQM) approach [31] to evaluate the performance of ReLog. The GQM approach conducts experiments from three layers. The conception layer (Goal) specifies goals of the experiments, the operation layer (Question) describes questions to achieve the goals, while the quantification layer (Metric) includes metrics to answer the questions.

The goal of ReLog is to support the efficient reprogramming of WSN platform. To achieve this goal, we first need to answer the following two questions. Q1, how does ReLog affect the reprogramming efficiency when deploying new applications? Q2, how does ReLog affect the reprogramming efficiency when updating existing applications? Meanwhile, since ReLog is a VM-based approach, the interpretive execution may incur side-effects on energy consumption and application results. Therefore, we also need to answer the third question. Q3, how does ReLog affect the execution of WSN applications?

To answer Q1, we first measure the energy consumption (M1) and the latency (M2) of disseminating the complete executable. These two metrics directly show the reprogramming efficiency of deploying new applications. In addition, we also measure the scale of application program (M3) as well as the size of executable (M4) since these metrics strongly affect the reprogramming efficiency of deploying new applications. Note that some existing efforts use *lines of code* to quantify the metric M3. However, a host of factors can influence M3 such as programming paradigm, abstraction level, and even programmers' skill. This simple quantification cannot exhibit these factors sufficiently. Instead, we choose to give complete programs to represent this metric.

To answer Q2, we first measure the energy consumption (M1) and the latency (M2) of disseminating the delta. These two metrics directly show the reprogramming efficiency of updating existing applications. In addition, we also measure the scale of program modification (M5) as well as the size of delta (M6). These two metrics strongly affect the reprogramming efficiency of updating existing applications. We choose to give fragments containing modifications of application programs to represent the metric M5.

To answer Q3, we first measure the CPU energy consumption (M7) of executing application code. We then measure the lifetime of sensors (M8) in practice by running applications on them until they run out of batteries. These two metrics reflect the energy efficiency of code execution. In addition, we also measure the total packet reception ratio of sensors (M9) to demonstrate the results of code execution of a data collection application.

We organize experimental results of these metrics into four parts including program (containing results of M3 and M5), script (containing results of M4 and M6), disseminating process (containing results of M1 and M2), and application execution (containing results of M7, M8, and M9).

7.1. Application scenario and reprogramming cases

To evaluate the performance of ReLog, we have implemented the approach and evaluated it with respect to the natural software evolution of an intelligent lab application, named iLab. iLab aims at automatically creating and maintaining a comfortable and green working environment. It detects indoor environment such as brightness, temperature and air quality, and adjusts the environment to satisfy customized user requirements and promotes healthy and green working style (e.g., pulling off curtains rather than switching lights on). One key component of this application is the sensor network platform which collects indoor environment information. The platform consists of four types of sensors as shown in Fig. 9. 16 active infrared sensors (AI sensors) detect whether seats are being occupied.[7] 16 temperature and light intensity sensors (TL sensors) provide temperature and brightness values around seats. Two CO_2 sensors as well as a HCHO and PM2.5 sensor provide indoor air quality.

We use the programs on AI sensors as an example to demonstrate the reprogramming cases. These cases consider an

[7] Note that each AI sensor has two active infrared sensing devices.

original program and a series of updates that the program had actually gone through during refinement of the application.

Case 1: The original program requires the AI sensors to send sensing values to the base station every 5 s.

Case 2: The original program is inefficient since the application only concerns data about the occupied seats. To improve the efficiency, a new program is created to request each AI sensor to send back its sensing values only if the monitored seat is occupied.

Case 3: In the previous program, an AI sensor may report false states of a seat accidentally (e.g., due to people passing). To address this issue, a new program requires each AI sensor to calculate the state of a seat according to three consecutive sensing values. Only the three equivalent sensing values can derive a stable state of a seat. To further improve the efficiency, an AI sensor is allowed to send back the stable state of a seat only if the state changes.

Case 4: Sensing values of TL sensors are useless if the seats are not being used. To address this problem, a new program requires each AI sensor to send the changed state of a seat to its associated TL sensor. With this state, a TL sensor can then decide when to send back its sensing values.

Case 1 presents the original program in iLab. Cases 2 and 4 represent small and moderate updates, respectively. Case 3 represents a huge update which can be seen as deploying a new program.

7.2. Program

We illustrate conciseness and ease-of-modification of ReLog programs using the four reprogramming cases. Specifically, we use the programs of cases 1 and 3 to demonstrate the scale of complete ReLog programs when dealing with various application requirements. We then use the programs of cases 2 and 4 to demonstrate the scale of program modifications when addressing different types of updates.

7.2.1. Scale of application program

Listing 2 The ReLog program of case 1

```
 1: # sys_dutyCycle = 10
 2:
 3: Predicate:
 4:     msg@unique.   infrared1@volatile:t1.   infrared2@volatile:t1.
 5:
 6: Clause:
 7:     msg(sys_nodeID, 0, 0, 0).
 8:     msg(Addr, X, Y, N+1):- infrared1(X)@passive, infrared2(Y),
                                            msg(Addr, X1, Y1, N)@passive.
 9:
10: Shell:
11:     boot() → setTimerMilli(t1, 5000, sys_infinity).
12:       t1()  →   insert(infrared1(sense(sys_GPIO2))),  insert(infrared2(sense
    (sys_GPIO3))).
13:     generate(msg(Addr, X, Y, N)) → send(1, <Addr, X, Y, N>).
```

Listing 2 gives the ReLog program of case 1 which has simple application logic. In this program, a sensor gets sensing values from GPIO interfaces every 5 s (lines 11 and 12). These sensing values are used to generate a new message with the sensor's ID and an increased sequence number (line 8). The sensor then sends the new message to the base station (line 13).

Listing 3 gives the ReLog program of case 3 which has more complex application logic. For an active infrared sensing device, the sensor gets the sensing value every 5 s (lines 20 and 21). The sensing value is then compared with the current state of the seat. If they indicate the same state, the counter decreases (line 9). Otherwise, the sensor initializes the current state with the sensing value (line 10). If the countering number is equal to 0 and the current state

Listing 3 The ReLog program of case 3

```
 1: # sys_dutyCycle = 10
 2:
 3: Predicate:
 4:        infrared1@volatile:t1.        cntState1@unique.        state1@unique.
       msg1@unique.
 5:        infrared2@volatile:t1.        cntState2@unique.        state2@unique.
       msg2@unique.
 6:
 7: Clause:
 8:     cntState1(3, 0).    state1(0).
 9:     cntState1(N-1, Sc) :- cntState1(N, Sc)@passive, infrared1(Sr), Sc == Sr.
10:     cntState1(3, Sr) :- cntState1(N, Sc)@passive, infrared1(Sr), Sc != Sr.
11:     state1(Sc) :- cntState1(N, Sc), N == 0, state1(S)@passive, S != Sc.
12:     msg1(sys_nodeID, 1, S) :- state1(S).
13:     cntState2(3, 0).    state2(0).
14:     cntState2(N-1, Sc) :- cntState2(N, Sc)@passive, infrared2(Sr), Sc == Sr.
15:     cntState2(3, Sr) :- cntState2(N, Sc)@passive, infrared2(Sr), Sc != Sr.
16:     state2(Sc) :- cntState2(N, Sc), N == 0, state2(S)@passive, S != Sc.
17:     msg2(sys_nodeID, 2, S) :- state2(S).
18:
19: Shell:
20:     boot() → setTimerMilli(t1, 5000, sys_infinity).
21:     t1() → insert(infrared1(sense(sys_GPIO2)), infrared2(sense(sys_GPIO3))).
22:     generate(msg1(Addr, ID, S)) → insert(cntState1(3, S)), send(1, <Addr, ID,
    S>).
23:     generate(msg2(Addr, ID, S)) → insert(cntState2(3, S)), send(1, <Addr, ID,
    S>).
```

is different from the stable state, the stable state changes its value to which of the current state (line 11). The changed state is used to generate a new message with IDs of the sensor and the sensing device (line 12). The sensor then sends the message to the base station and initializes the counter of the current state (line 22). Similar steps (lines 14–17, 23) are applied to the other sensing device.

From these two programs, we find that both of them are almost line-by-line translations of application requirements. They directly map application requirements into high-level descriptions without involving system-level implementation details. The results indicate that the ReLog language is able to help users to get concises programs.

7.2.2. Scale of program modification

Listing 4 Modifications in the ReLog program of case 2

```
1: ...
2: Clause:
3:     ...
4:     msg(Addr, 1, V, N+1):- infrared1(V), msg(Addr, ID, V1, N)@passive, V == 1.
5:     msg(Addr, 2, V, N+1):- infrared2(V), msg(Addr, ID, V2, N)@passive, V == 1.
6: ...
```

Listing 4 shows modifications in the program of case 2. According to the new application requirement in case 2, we replace the original rule (line 8 in Listing 2) with two new rules (lines 4 and 5) in the program. These new rules indicate that sensing values can be used to generate messages only if the monitored seats are being used (i.e., sensing values equal to 1).

Listing 5 gives modifications in the program of case 4. According to the new application requirement in case 4, we modify the boot event statement and add two predicates as well as a rule to calculate the modulus of the sensor's ID (lines 13, 3, 6). This modulus is used to calculate addresses of the sensor's associated TL sensors when generating messages (lines 8 and 10). The sensor then uses these addresses to send the messages to its associated TL sensors (lines 15 and 16).

These two fragments show that it is easy to modify ReLog programs to satisfy new application requirements. Further, changes of application requirements will not incur modifications of irrelevant parts of the programs. The results provide compelling evidence that the ReLog language naturally supports loosely coupled programs which can be modified easily.

Listing 5 Modifications in the ReLog program of case 4

```
...
Predicate:
    address.   modulus.   ...

Clause:
    modulus(NodeID % 2) :- address(NodeID).
    ...
    msg1(sys_nodeID - 1 - Mod, S) :- state1(S), modulus(Mod)@passive.
    ...
    msg2(sys_nodeID + 2 - Mod, S) :- state2(S), modulus(Mod)@passive.

Shell:
    boot() → setTimerMilli(t1, 5000, sys_infinity), insert(address(sys_nodeID)).

    ...
    generate(msg1(Addr, S)) → insert(cntState1(3, S)), send(Addr, 1, <Addr,
S>).
    generate(msg2(Addr, S)) → insert(cntState2(3, S)), send(Addr, 1, <Addr,
S>).
```

7.3. Script

We measure the sizes of scripts of all the reprogramming cases. Particularly, we measure the sizes of executables of all the four cases as well as the sizes of deltas of the last three cases. We choose Darjeeling [4] for comparison since it also provides a generic VM.

7.3.1. Size of executable

Fig. 10 shows the sizes of executables of the four cases. The results show that ReLog reduces the script size by 61.4%–83.2% compared to Darjeeling [4]. This is because the executables of ReLog only contain high-level intermediate representations with no machine-level instruction. The size of the executable is only determined by the scale of its source program.

7.3.2. Size of delta

Fig. 11 shows the sizes of deltas in the last three cases. Compared to the complete executables, the deltas further reduce the script size by 43.4%, 22.7%, and 49.6% of the three cases, respectively. These results show that the adoption of incremental reprogramming can achieve significant reduction on the script size for small and moderate updates (cases 2 and 4). Additionally, it also works well even for the huge update (case 3). The ReLog compiler takes several optimizations during the delta generation. To evaluate effects of these optimizations, Fig. 11 also gives the sizes of deltas without these optimizations. The results show that the optimizations reduce the size of delta by 9.3% and 30.4% of cases 3 and 4, respectively. They did not work for case 2 for the following two reasons. (1) Predicates in the program of case 2 are unchanged, so as their system-generated names. (2) There is no need to rearrange rule tokens in the executable since rules in the program of case 2 are totally new.

7.4. Dissemination process

We measure energy consumption and latencies of disseminating scripts of the four cases via simulations. These simulations consider different data dissemination protocols (T^2C [39] and McTorrent [14]) as well as different sensor networks (i.e., grids of 5×10, 5×20, and 5×30 nodes). We also measure the latencies of disseminating these scripts on the iLab sensor network platform.

The scripts are first divided into a number of pages. Each page includes 10 packets and the size of payload in each packet is 25 bytes. With these settings, the scripts in Darjeeling need 4, 5, 6, and 6 pages for the four cases, respectively, while the scripts in ReLog need 1, 1, 2, and 2 pages, respectively. We use the Castalia [3] simulator since it provides realistic wireless channels and radio models as well as realistic node behaviors such as clock drift.

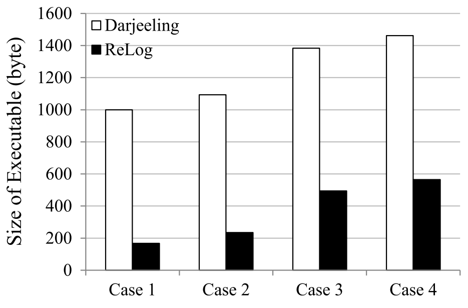


Fig. 10. Sizes of executables in the four reprogramming cases.

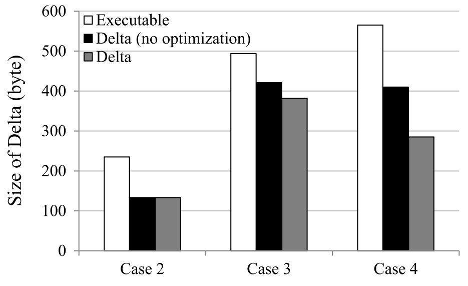


Fig. 11. Sizes of deltas in the last three reprogramming cases.

To better show the experimental results, we provide confidence intervals for each of the average values. All these confidence intervals are shown with 90% confidence level. This confidence level is also used in T^2C [39] and McTorrent [14].

7.4.1. Energy consumption of dissemination process

Fig. 12 shows the total energy consumption of disseminating the scripts in grids of 50, 100, and 150 nodes with T^2C. Compared to Darjeeling, ReLog reduces the energy consumption by 28.66%–36.09%, 23.55%–28.37%, and 17.52%–24.83%, respectively. Fig. 13 shows the total energy consumption of disseminating the scripts in the three grids with McTorrent. Compared to Darjeeling, ReLog also reduces the energy consumption by 38.73%–46.18%, 19.54%–28.51%, and 16.53%–23.85%, respectively. These results show that the smaller scripts of ReLog have better performance on the energy consumption of the dissemination processes.

7.4.2. Latency of dissemination process

Fig. 14 shows the latencies of disseminating the scripts in grids of 50, 100, and 150 nodes with T^2C. Compared to Darjeeling, ReLog reduces the latency by 34.47%–43.98%, 31.65%–38.59%, and 20.66%–32.84%, respectively. Fig. 15 shows the latencies of disseminating the scripts in the three grids with McTorrent. Compared to Darjeeling, ReLog also reduces the latency by 38.69%–47.18%, 19.54%–25.89%, and 16.55%–23.34%, respectively. Fig. 16 shows the latencies of disseminating the scripts to the AI sensors of the iLab sensor network platform. We only use T^2C in this experiment since it supports the target-specified dissemination which can disseminate scripts to the AI sensors only. The results show that ReLog reduces the latency by 46.52%–57.5% compared to Darjeeling. These results suggest that the smaller scripts of ReLog can also reduce the latencies of the dissemination processes.

7.5. Application execution

We measure the CPU energy consumption by running programs of the four cases through simulations. We choose the Avrora [34]

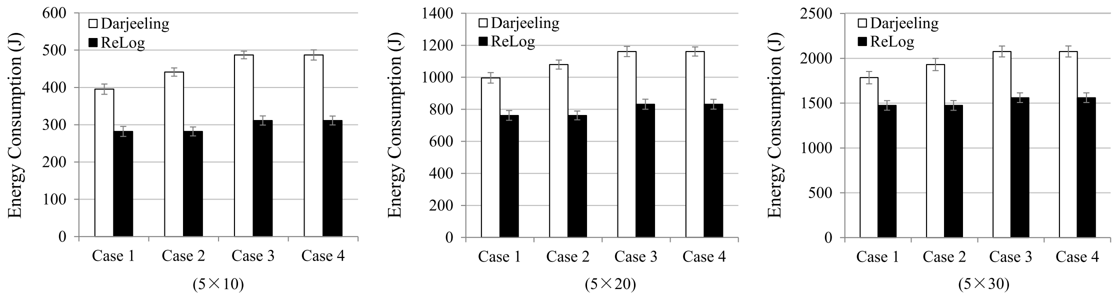


Fig. 12. Total energy consumption of disseminating the scripts in various networks with T^{2}C.

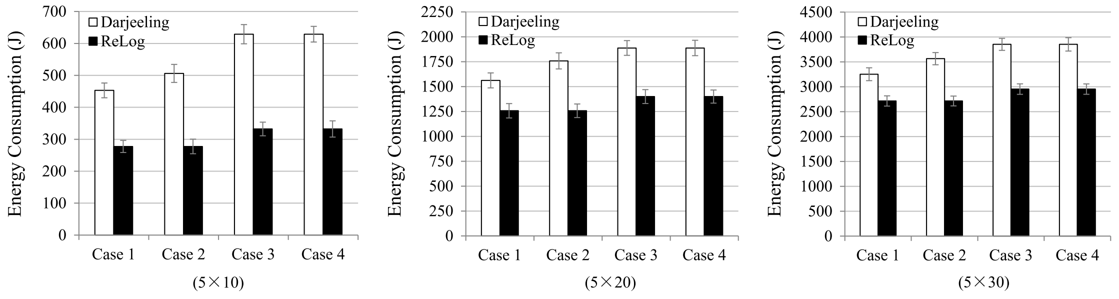


Fig. 13. Total energy consumption of disseminating the scripts in various networks with McTorrent.

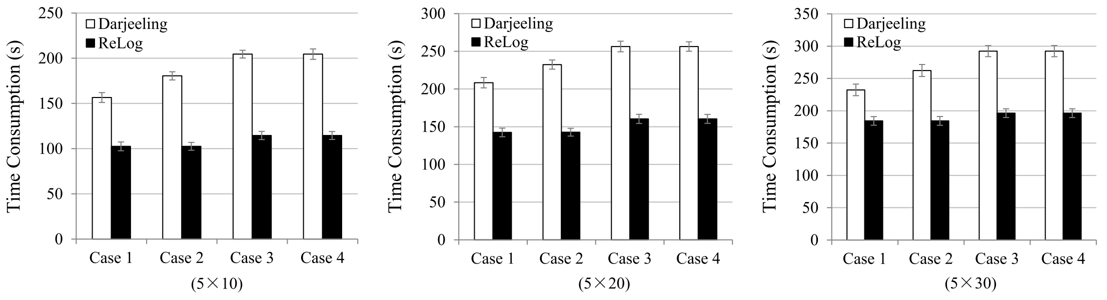


Fig. 14. Latencies of disseminating the scripts in various networks with T^{2}C.

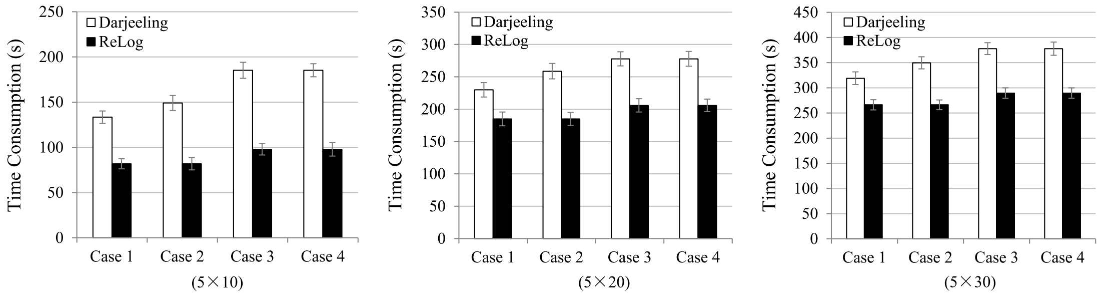


Fig. 15. Latencies of disseminating the scripts in various networks with McTorrent.

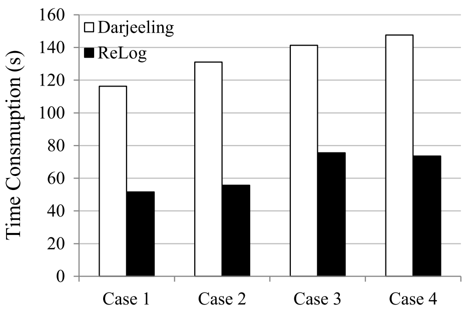


Fig. 16. Latencies of disseminating the scripts to the active infrared sensors.

simulator since it can provide accurate statistic data in detail. We measure the lifetime of sensors by counting the number of their transmitted packets before their batteries are exhausted. We choose the program of case 4 since it has more complex application logic. We also measure the total packet reception ratio of the AI sensors using the program of case 1. We choose TinyOS [20] for comparison in these experiments since it can provide native code which has high execution efficiency.

7.5.1. CPU energy consumption

We make programs of the four cases send messages directly (not through the routing protocol) in each sampling slot to get comparable results. We run each simulation for 1000, 2000, and

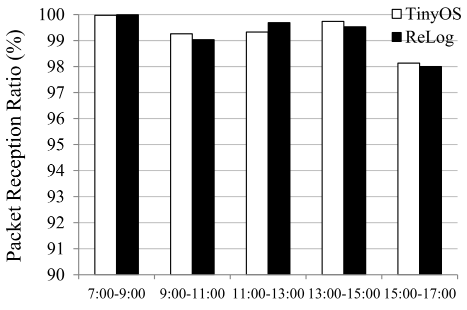


Fig. 17. Total packet reception ratios of different slots.

3000 s. Fig. 18 shows the CPU energy consumption of TinyOS [20] and ReLog of the four cases for 2000 s. Compared to TinyOS, ReLog increases the CPU energy consumption by 1.74%, 2.23%, 3.76%, and 4.08%, respectively. The results show that the additional overhead incurred by interpretive execution is acceptable. We obtain much similar results for the other two experiments running for 1000 and 3000 s.

7.5.2. *Sensor lifetime*

We use two telosB sensors (sensor 1 and sensor 2) and each of them equips a lithium battery with a capacity of 750 mAh. To analyze the impact of battery's lifetime, the battery of sensor 1 is almost new while the battery of sensor 2 has been used for nearly 11 months. For each sensor, we repeat the experiments four times for TinyOS and ReLog, respectively. We alternate the experiments of TinyOS and ReLog on each sensor to achieve fair results. Fig. 19 gives the number of transmitted packets of the two sensors. For sensors 1 and 2, the number of transmitted packets of ReLog is 97.66%–98.22% and 97.04%–98.31% of that of TinyOS, respectively. The results suggest that the additional overhead of VM is acceptable.

7.5.3. *Packet collection ratio*

We run the data collection application in the same period of time of two consecutive working days and count the packet reception ratios of different slots. Fig. 17 shows the packet reception ratios of different slots of TinyOS and ReLog. The results show that TinyOS and ReLog perform nearly identically, with absolute differences of less than 0.5%. An interesting phenomenon is that the packet reception ratios of TinyOS and ReLog have similar variations with time. This is because slots of working hours (i.e., 9:00–11:00, 13:00–15:00, and 15:00–17:00) have more interference (mainly due to Wi-Fi devices) than which of other hours (i.e., 7:00–9:00 and 11:00–13:00) in the lab.

8. Conclusions

This paper proposes ReLog, a systemic approach which aims to better support efficient reprogramming of WSN platform. ReLog consists of a programming language, a compiler, and a VM. By inheriting the logical programming paradigm and providing high-level programming abstractions, the ReLog language can support concises programs with the loosely coupled structure. With the specially designed executable and the optimized delta generation, the compiler can significantly reduce the script size in reprogramming cases of both deploying new applications and updating existing applications. By optimizing the executable as well as the execution process, the VM can efficiently diminish the additional energy consumption incurred by interpretive execution. We have implemented ReLog and evaluated it with respect to real reprogramming cases. Experimental results show that ReLog can significantly improve the reprogramming efficiency of WSN platform.

For our future work, we attempt to improve the flexibility of the ReLog language by providing interfaces for user-defined functions. The logical programming language is not well suited to some tasks such as image processing and signal filtering. User-defined functions facilitate users to implement these tasks with other

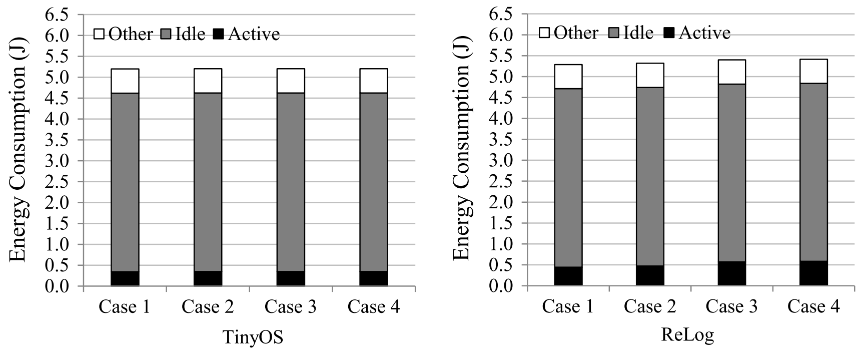


Fig. 18. CPU energy consumption in various states through simulation.

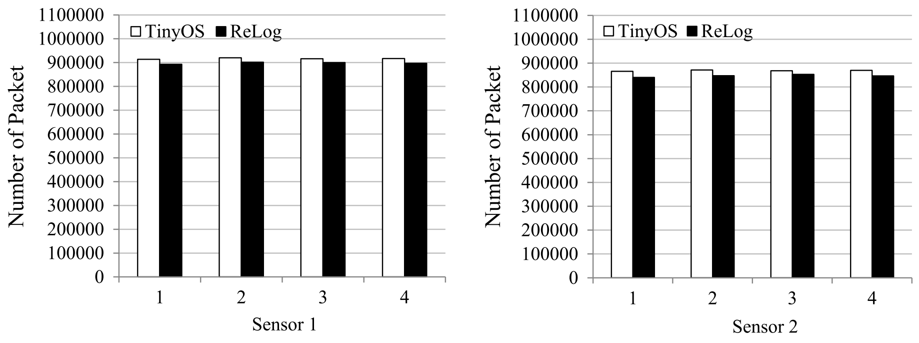


Fig. 19. Number of packets transmitted by two sensors.

programming languages. Different from snlog [26], ReLog adds these user-defined functions to the implementation of VM rather than application programs. Therefore, the programs still keep both conciseness and ease-of-modification features. A consequential problem of the improvement is that the ReLog VM requires the capability to update itself. In the future version of ReLog, we plan to improve the ReLog VM by making it support loading and registering user-defined functions dynamically.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (Nos. 61373011, 91318301, and 61321491).

References

[1] R. Bajwa, R. Rajagopal, E. Coleri, P. Varaiya, C. Flores, In-pavement wireless weigh-in-motion, in: Proceedings of the 12th International Conference on Information Processing in Sensor Networks, IPSN'13, 2013, pp. 103–114.
[2] R. Bajwa, R. Rajagopal, P. Varaiya, R. Kavaler, In-pavement wireless sensor network for vehicle classification, in: Proceedings of the 10th International Conference on Information Processing in Sensor Networks, 2011, pp. 85–96.
[3] A. Boulis, Castalia: revealing pitfalls in designing distributed algorithms in wsn, in: Proceedings of the 5th ACM Conference on Embedded Networked Sensor Systems, ACM, 2007, pp. 407–408.
[4] N. Brouwers, K. Langendoen, P. Corke, Darjeeling, a feature-rich vm for the resource poor, in: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, SenSys'09, 2009, pp. 169–182.
[5] M. Ceriotti, M. Corra, L. D'Orazio, R. Doriguzzi, D. Facchin, S. Guna, G. Jesi, R. Lo Cigno, L. Mottola, A. Murphy, M. Pescalli, G. Picco, D. Pregnolato, C. Torghele, Is there light at the ends of the tunnel? wireless sensor networks for adaptive lighting in road tunnels, in: Proceedings of the 10th International Conference on Information Processing in Sensor Networks, 2011, pp. 187–198.
[6] O. Chipara, C. Lu, T.C. Bailey, G.-C. Roman, Reliable clinical monitoring using wireless sensor networks: experiences in a step-down hospital unit, in: Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems, ACM, 2010, pp. 155–168.
[7] D. Chu, L. Popa, A. Tavakoli, J.M. Hellerstein, P. Levis, S. Shenker, I. Stoica, The design and implementation of a declarative sensor network system, in: Proceedings of the 5th International Conference on Embedded Networked Sensor Systems, ACM, 2007, pp. 175–188.
[8] W. Dong, B. Mo, C. Huang, Y. Liu, C. Chen, R3: Optimizing relocatable code for efficient reprogramming in networked embedded systems, in: Proceedings of the International Conference on Computer Communications, IEEE, 2013, pp. 315–319.
[9] A. Dunkels, N. Finne, J. Eriksson, T. Voigt, Run-time dynamic linking for reprogramming wireless sensor networks, in: Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, SenSys'06, 2006.
[10] V.L. Erickson, S. Achleitner, A.E. Cerpa, Poem: Power-efficient occupancy-based energy management system, in: Proceedings of the 12th International Conference on Information Processing in Sensor Networks, IPSN'13, 2013, pp. 203–216.
[11] R. Fontugne, J. Ortiz, N. Tremblay, P. Borgnat, P. Flandrin, K. Fukuda, D. Culler, H. Esaki, Strip, bind, and search: A method for identifying abnormal energy consumption in buildings, in: Proceedings of the 12th International Conference on Information Processing in Sensor Networks, IPSN'13, 2013, pp. 129–140.
[12] H. Gupta, X. Zhu, X. Xu, Deductive framework for programming sensor networks, in: Proceedings of the 25th International Conference on Data Engineering, IEEE, 2009, pp. 281–292.
[13] M.S. Hossain, A.B.M. Alim, Al Islam, M. Kulkarni, V. Raghunathan, μsetl: A set based programming abstraction for wireless sensor networks, in: Proceedings of the 10th International Conference on Information Processing in Sensor Networks, 2011, pp. 354–365.
[14] L. Huang, S. Setia, R. Simon, Mctorrent: Using multiple communication channels for efficient bulk data dissemination in wireless sensor networks, J. Syst. Softw. 83 (1) (2010) 108–120.
[15] J. Jeong, D. Culler, Incremental network programming for wireless sensors, in: Proceedings of the qst Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, 2004, pp. 25–33.
[16] J. Koshy, R. Pandey, Remote incremental linking for energy-efficient reprogramming of sensor networks, in: Proceeedings of the 2nd European Workshop on Wireless Sensor Networks, 2005, pp. 354–365.
[17] J. Koshy, R. Pandey, Vmstar: Synthesizing scalable runtime environments for sensor networks, in: Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems, SenSys'05, 2005, pp. 243–254.
[18] P. Levis, D. Culler, Mate: A tiny virtual machine for sensor networks, Sigops Oper. Syst. Rev. 37 (10) (2002) 85–95.
[19] P. Levis, D. Gay, D. Culler, Active sensor networks, in: Proceedings of the 2nd USENIX/ACM Symposium on Network Systems Design and Implementation.
[20] P. Levis, S. Madden, J. Polastre, R. Szewczyk, et al., Tinyos: An operating system for sensor networks, in: Ambient Intelligence, Springer, 2005, pp. 115–148.
[21] M. Li, Y. Liu, J. Wang, Z. Yang, Sensor network navigation without locations, in: Proceedings of the International Conference on Computer Communications, 2009, pp. 2419–2427.
[22] S.R. Madden, M.J. Franklin, J.M. Hellerstein, W. Hong, Tinydb: an acquisitional query processing system for sensor networks, ACM Trans. Database Syst. (TODS) 30 (1) (2005) 122–173.
[23] P.J. Marrn, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, K. Rothermel, Flexcup: A flexible and efficient code update mechanism for sensor networks, in: Proceedings of the 3rd European Workshop on Wireless Sensor Networks, 2006, pp. 212–227.
[24] L. Mo, Y. He, Y. Liu, J. Zhao, S.-J. Tang, X.-Y. Li, G. Dai, Canopy closure estimates with greenorbs: Sustainable sensing in the forest, in: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, SenSys'09, 2009, pp. 99–112.
[25] L. Mottola, G.P. Picco, Programming wireless sensor networks: Fundamental concepts and state of the art, ACM Comput. Surv. (CSUR) 43 (3) (2011) 19.
[26] R. Newton, G. Morrisett, M. Welsh, The regiment macroprogramming system, in: Proceedings of the 6th International Symposium on Information Processing in Sensor Networks, 2007, pp. 489–498.
[27] F.J. Oppermann, C.A. Boano, K. Römer, A decade of wireless sensing applications: Survey and taxonomy, in: The Art of Wireless Sensor Networks, Springer, 2014, pp. 11–50.
[28] R. Panta, S. Bagchi, Hermes: Fast and energy efficient incremental code updates for wireless sensor networks, in: Proceedings of the International Conference on Computer Communications, 2009, pp. 639–647.
[29] A. Singh, C.R. Ramakrishnan, I.V. Ramakrishnan, D.S. Warren, J.L. Wong, A methodology for in-network evaluation of integrated logical-statistical models, in: Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems, ACM, 2008, pp. 197–210.
[30] Specification of the relog language, in: http://moon.nju.edu.cn/people/xiaoruizhu/static/Spec.pdf, 2015.
[31] R. Van Solingen, V. Basili, G. Caldiera, H.D. Rombach, Goal question metric (gqm) approach, in: Encyclopedia of Software Engineering, 2002.
[32] Virtual machine specification, java card platform, v2.2.2, 2006.
[33] A. Taherkordi, F. Loiret, R. Rouvoy, F. Eliassen, Optimizing sensor network reprogramming via in situ reconfigurable components, ACM Trans. Sens. Netw. (TOSN) 9 (2) (2013) 14.
[34] B.L. Titzer, D.K. Lee, J. Palsberg, Avrora: Scalable sensor network simulation with precise timing, in: Proceedings of the 4th International Symposium on Information Processing in Sensor Networks, IEEE, 2005, pp. 477–482.
[35] N. Tsiftes, A. Dunkels, A database in every sensor, in: Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems, ACM, 2011, pp. 316–332.
[36] M. Welsh, G. Mainland, Programming sensor networks using abstract regions., in: Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (2004) 29–42.
[37] K. Whitehouse, F. Zhao, J. Liu, Semantic streams: A framework for composable semantic interpretation of sensor data, in: Proceedings of the European Workshop on Wireless Sensor Networks, Springer, 2006, pp. 5–20.
[38] L. Yu, N. Wang, X. Meng, Real-time forest fire detection with wireless sensor networks, in: Proceedingsof the International Conference on Wireless Communications, Vol. 2, IEEE, 2005, pp. 1214–1217.
[39] X. Zhu, X. Tao, T. Gu, J. Lu, Target-aware, transmission power-adaptive, and collision-free data dissemination in wireless sensor networks, IEEE Trans. Wireless Commun. (2005) 6911–6925.
[40] X. Zhu, X. Tao, H. Xie, J. Lu, Reprogramming-oriented logical programming language for wireless sensor network applications (in Chinese), J. Softw. 2 (2014) 326–340.

Xiaorui Zhu received the B.Sc. and M.Sc. degrees in computer science from HoHai University in 2006 and 2009, respectively. He is currently a Ph.D. candidate in the Department of Computer Science at Nanjing University. His research interests include programming languages and reprogramming systems for wireless sensor network applications.

Xianping Tao received his M.Sc. and Ph.D. degrees in computer science from Nanjing University in 1994 and 2001, respectively. He is currently a professor in the Department of Computer Science at Nanjing University. His research interests include software agents, middleware systems, Internetware methodology, and pervasive computing. He is a member of CCF and IEEE.

Tao Gu received his B.Sc. degree from Huazhong University of Science and Technology, and M.Sc. from Nanyang Technological University, Singapore, and Ph.D. in computer science from National University of Singapore. He is currently an Associate Professor in the School of Computer Science and Information Technology at RMIT University. His research interests include mobile and pervasive computing, wireless sensor networks, distributed network systems, sensor data analytics, cyber physical system, Internet of Things, and online social networks. He is a senior member of the IEEE and a member of the ACM.

Jian Lu received his B.Sc., M.Sc. and Ph.D. degrees in Computer Science from Nanjing University, China. He is currently a Professor in the Department of Computer Science and Technology and the Director of the State Key Laboratory for Novel Software Technology at Nanjing University. He serves on the Board of the International Institute for Software Technology of the United Nations University (UNU-IIST). He also serves as the director of the Software Engineering Technical Committee of the China Computer Federation. His research interests include software methodologies, software automation, software agents, and middleware systems. He is a member of ACM and a fellow of CCF.