一〇〇年度「新一代網際網路協定互通認證計畫」
研究報告附件

uFlow: Dynamic Software Updating in Wireless Sensor Networks

應用服務分項計畫子計畫三
主持人：趙涵捷 校長
執行單位：國立清華大學
中華民國一〇〇年十月

uFlow: Dynamic Software Updating in Wireless Sensor Networks

Ting-Yun Chi , Wei-Cheng Wang, Sy-Yen Kuo

Electrical Engineering, National Taiwan University, Taiepi 10617 Taiwan(R.O.C)
Louk.chi@gmail.com cole945@gmail.com sykuo@cc.ee.ntu.edu.tw

Abstract. A wireless sensor network (WSN) consists of spatially distributed autonomous sensors to monitor physical or environmental conditions, such as temperature, sound, vibration, pressure. Due to the maintenance reason, we may update the software to fix bugs. Because there are more and more sensor nodes using in the vehicle, smart environment, Software updating for wireless sensor networks has become an important issue. In previous related works, Update the node usually is required to reboot. However, reboot the nodes is costly since the previous runtime status may be lost. To recover the runtime status for routing, it will take time and bandwidth to synchronize with other nodes. We present uFlow: a programming paradigm and a prototype implementation for wireless sensor networks. uFlow allows application to update the nodes without rebooting. So we can avoid to lost precious runtime status.

Keywords: Software updating, wireless sensor networks, runtime status

1 Introduction

A wireless sensor network (WSN) consists of spatially distributed autonomous sensors to monitor physical or environmental conditions, such as temperature, sound, vibration, pressure. Because there are more and more sensor nodes using in the vehicle, smart environment, Software updating for wireless sensor networks has become an important issue. By updating the software, we can fix bugs to maintain the senor network. However it is a challenge to update the software with limited hardware resources and environment. Previous works try to save the problems by only distributing the changed parts, but this is not adequate. First, even we change a small part of the code; the binary code will be many differences with the old one. It causes from address relocations of functions and variables. Second, direct function calls makes it hard to safely and dynamically replace the functionalities of the program at runtime. And the worse is that rebooting the entire node after updating is costly – it loses precious status data and waste energy to rebuild the status. A node may take minutes or hours to fully restore the running status.

Our work presents a programming model. The basic functionalities are implemented as tasks and the control flow which is managed by a lightweight engine. By this way, we make dynamic replacement become possible. We not only keep the runtime status data during updating but also avoid unnecessary changes to the binary image. We only transmit the limited data over the network. In section 1, we explain why the software updating is important to sensor network. The rest of the paper is organized as follows. Section 2 presents the related works around software updating in wireless sensor network field. Section 3 proposes the idea of uFlow and explains system architecture. Section 4 shows the result of prototype implementation engine, example program. We make the conclusion in section 5.

2 Related works

The Researches for software updating in wireless sensor networks can be divided by three perspectives – dissemination, patch generating, and execution environment. The big picture of related works is illustrated in Figure 1. These perspectives are not fully mutual exclusive but sometimes cooperative to provide a better and integrative approach for software updating.

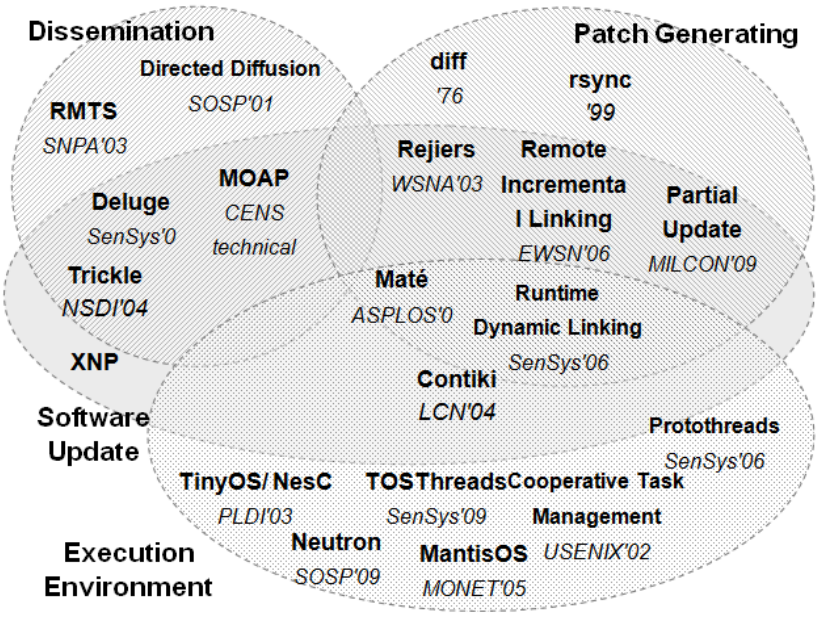

Fig. 1. The big picture - Software Updating for Wireless Sensor Network

2.1 Dissemination

Dissemination concerns about how to spread the message across the network. In wireless sensor network, sensor nodes often cooperate in a distributed manner lacking pre-established infrastructure for routing and forwarding information. In order to exchange information, these nodes have to autonomously build an ad-hoc network to communicate with nodes multiple hops away. Moreover, conventional dissemination approaches for data collection are not adequate when applying to software update. Software program images are usually much larger than sensed data, and hence they will be fragmented into several small chunks before transmitting and assembled, correctly, when receiving. Directed Diffusion [4] is one of the most fundamental approach widely used, and MOAP [6] discussed and analyzed the design goals, constraints, and choices between difference dissemination strategies for software update in wireless sensor network.

2.2 Traffic Reduction

During software update, transmitting the entire program image is not economical, because, in most case, only partial of the program need to be modified. Transmitting unnecessary parts wastes bandwidth, reduces battery life, increase update latency, and more likely to fail.

diff
The rule of thumb is that, only distributing the changed parts. One of the most representative and widely used works is diff [2], a UNIX utility, which was developed in the early 1970s on the UNIX operating system. It is typically used to compare two difference versions of the same text file, and display the changes made per line. The output is called a patch, or an edit script, which is usually very small compared to the original files. Later, one can use the patch file to update the previous version to the newer version. Similarly, one can use the patch file to downgrade the newer version to the previous version. Therefore only the small patch file, instead of the entire newer version file, is needed to update a file.

Revised Diff
Based on the UNIX diff system, Rejiers [7] introduces a revised approach diff-like approach which uses a different operation.

Slop Space
Beyond traffic reduction, another issue [7] emerged from rebuilding a program image. When the size of subroutine is changed to shrink or grow, the addresses of unrelated subroutines may shift backward or forward. Address relocations cause every instruction which referring to the relocated address needs to be patched. Unnecessary patches waste network bandwidth and power. Worse, even a single byte in a flash memory page need to be altered, the entire flash memory page is erased and rewritten with new data. These erase and rewrite operations are slow and power consumed.

2.3 Execution Environment

Instead of writing an application from scratch, reinventing wheel and directly access underlying hardware platform, applications are usually running on top of a hardware abstraction layer [12, 13, 14, 15, 16, and 17]. A hardware abstraction layer is usually provided by an operating system, a middle-ware, or other execution environment. The purpose of hardware abstraction is hide the difference in hardware, hence the application be ported to other hardware more easily. The execution environment may also provide off-the-shelf and try-and-true facilities to programmers that reduce the level of expertise and effort requirement by the programmers.

Task Management
Task management [18-20] is one of the most important functionalities provided by underlying execution environment to coordinate tasks. When multiple tasks are running concurrently, race condition or dead lock may occur if the programmer did not properly use the synchronization tools. To ease the pain of concurrency issue, the execution environment often provides the well-designed concurrency model and programming model to simplify the concurrency issues [18].

Virtual Machine
Maté [13] is a virtual machine designed for sensor network. Compared to native machine code, the bytecode, executed by the interpreter, is very concise. At first glance, running virtual machine on sensor nodes seem impractical due to the cost of interpreting bytecode is quite high. However Maté is based on the assumption that the energy cost of interpreting bytecode is outweighed by the energy saved by transmitting bytecode instead native machine. Although Maté is impressive in terms of code size and the ability of reprogramming, but the results show the interpreting overhead is rather big. According to the conclusion of Maté, in the gdi-comm case, the energy can be saved only when running five days of less.

Table 1. Cost of Maté operations [13]

Operations	Maté Clock Cycles	Native Clock Cycles	Cost
Simple: and	469	14	33.5:1
Downcall: rand	435	45	9.5:1
Quick Splite: sense	1342	396	3.4:1
Long Split: sendr	685+≈20,000	≈20,000	1.03:1

Table 2. Maté application costs [13]

Application	Binary Size	Install Time	Maté Capsules	Instructions
sens_to_rfm	5394B	79s	1	6
gdi-comm	7130B	104s	1	19
bless_test	7381B	108s	7	108

Cost of Reboots
Neutron [17] is a specialized version TinyOS that efficiently recovers from memory safety bugs by only rebooting the faulting units instead of the entire node. The work is based on the assumption – rebooting is costly. When a node is rebooting, the precious runtime state is lost. For example, lost data of weather condition cannot be collected again. Besides a rebooting node takes time to re-synchronize with other nodes, and control packets for routing protocol waste precious network bandwidth and energy. To preserve these status by rebooting only the faulting units, Neutron can reduce 94% of the cost for time-synchronization for Flooding Time Synchronization Protocol and 99.5% of the cost for a routing protocol for Collection Tree Protocol.

Data-driven, or Dynamic, Programming
In computer programming language [21], static means things are determined during and before the compile-time, whereas dynamic means things are determined when the programming is being executed. Data-driven programming [24], sometimes called dynamic programming, is one of the most classic approaches to dynamically modify a program at run-time. Instead of hardcoding the logic of a program, the logic of the program is separating from the code. Therefore, the logic of a programming can be modified by editing the data structures, even at runtime, instead of the code.

3 System Design

A program is decomposed into tasks and transitions. A task is an application-independent reusable functionality, and a transition describes how execution flow transits from tasks to tasks. A task can be constructed from tasks and transitions to represent a more complex task. For example, three tasks have to be done and the second task invoking 3 steps. They can be illustrated as the following structure.

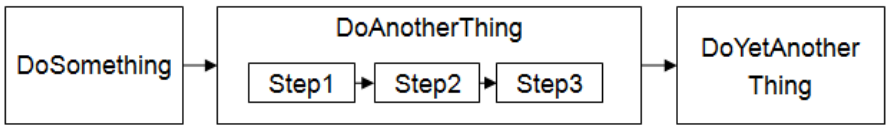

Fig. 2. Structuring of tasks and transitions

Transitions provide the mechanism of conventional control flow statements. As Figure 3 5 shows, the transitions, represented as solid arrow lines, describe the order in which the individual task are executed. That is, DoSomething, Step1 of DoAnotherThing, Step2 of DoAnotherThing, Step3 of DoAnotherThing, and finally DoYetAntoherThing.

Tasks in uFlow

A task is a fundamental building block of a program. It defines what to do, or how to transit the execution flow to next task. There are two types of tasks, simple and composite. A simple task represents a single basic task to be done. It is associated with a subroutine, called execution handler, which is invoked when the task is

executed. For example, tasks for blinking LED or transmitting a packet. A composite task contains other child tasks. It plays the role of control flow to coordinate a collection of child tasks to accomplish a more sophisticated task. For example, instead of building a monolithic task for U-turns from scratch, it can be built from reusing two turn-right tasks, and a forward task, off the shelf. Therefore, a composite task is composed of a collection of child tasks, and a rule of how to execute the child tasks. A rule can be sequence execution, conditional branch, while-loop, or even a programmer defined one. For example, the pseudo code of conditional branch in would be:

```
If condition is true then
   Execute the first child
Else If
   Execute the second child
End If
```

Fig. 3. Pseudo code of conditional branch

The rule of a composite task is implemented by the execution handler and a continuation handler. Similar to simple task, the execution handler is invoked when the task is executed, and the composite task can be notified when a child task complete its execution thought the continuation handler. To elaborate, see EXECUTE UFLOW PROGRAMS. In order to simplify the program structure, localize a modification to prevent from affecting the rest of the program and provide reusability, the composite pattern is adopted [22]. Hereby, a composite task is treated in the same way as a simple task. That is, a task can be composed of tasks, and eventually the hierarchy of task forms a tree which representing the entire program itself. The program starts its execution from the root, then the child tasks, and so forth. The UML diagram of tasks is illustrated in Figure 3 7. Both a simple task and a composite task inherit from the task which provides execution handler to perform a specific work. While a composite task has a list of potential transitions and a continuation handler used to determine which transition to take.

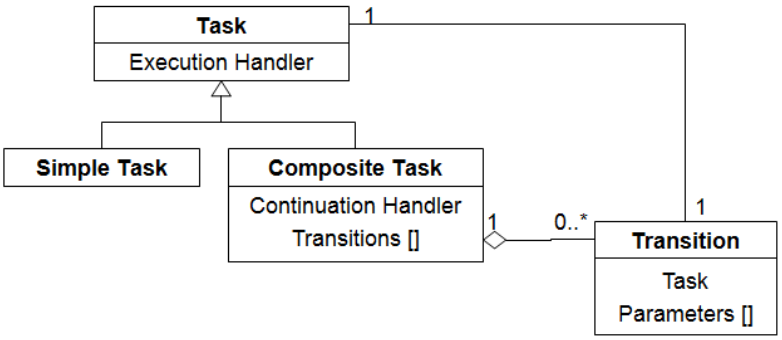

Fig. 4. Task composition

Control Flow in uFlow

As stated previously, a program is a collection of tasks to be done, and transitions are used to define their execution orders. This section describes how uFlow organizes tasks and transitions to into a running program. A task contains other tasks is call a composite task. A composite task manages the execution of is child tasks, and it completes its execution depends on its transition rule. uFlow doesn't use explicit transitions to describe the control flow. Instead, composite tasks are used to structure the control flow. Examples of composite tasks used for control flow are depicted in the following section.

Sequence Execution

Sequence Task is used to run a list of tasks in an ordered manner. The children are executed one by one at a time. When the last child completes, the sequence task per se completes.

Conditional Branch

Conditional branch task is used to execute its child task only if a specified condition is met. It acts like the if-else statement in programming language. A conditional branch task can have one or two children, and a conditional function. If the conditional function is evaluated to true, then the first child is chosen to be run, otherwise the second child, if exists.

While Loop

Loops structure is used to repeatedly execute a task based on a specified condition. For example, while loop, do-while loop, and for-loop.

Hierarchized Structure

These control flow tasks can be combined to form a more sophisticated task. Hereby, although only one or few tasks can be put in a task, but it doesn't mean only a few tasks can be performed.

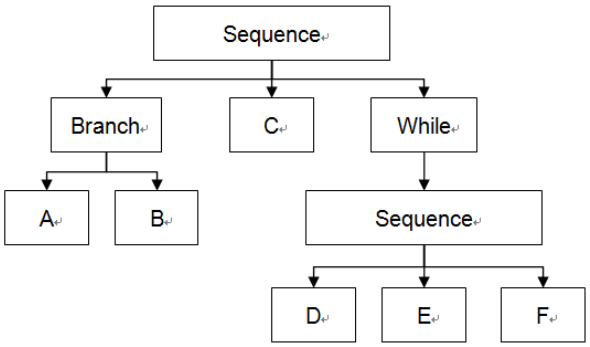

Fig.5. Hierarchized structure example

Before explaining the example in Figure 5, the tree-style diagrams may look confusing and not straightforward about what it intends to do, because it is used to illustrate the parent-child relationship of a composite task, not the execution flow. A flowchart-style equivalent counterpart of the diagram can be used to represent the execution flow of the tasks.

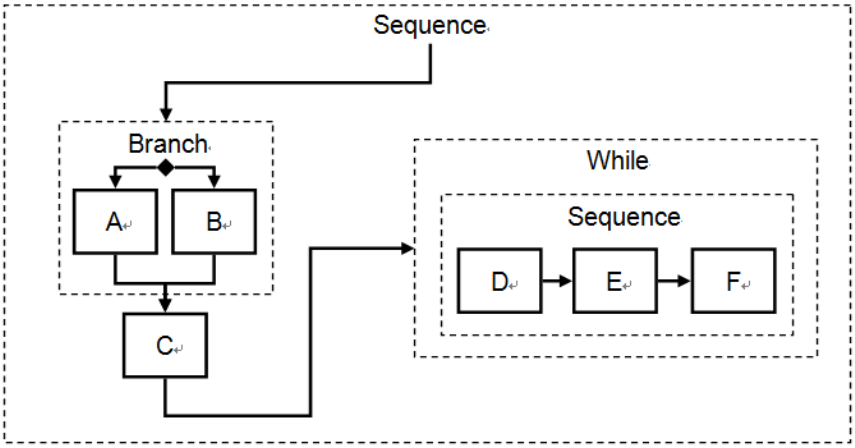

Fig.6. Flowchart-style representation

The diagram in Figure 6 is the equivalent counterpart of the diagram in Figure 5 in different style and emphasizing on the execution flow the tasks. The solid block represents a task to be done, while the solid arrow denotes the execution order.
It is now straightforward that either A or B will be executed, then C, while D, E, and F as a whole will be executed repeatedly based on a given condition.

Context Stack

A task acts like a blueprint to describe the functionality and the behavior, and it may be used again to repeat a same operation or to cooperate with other task to form a new functionality. However, this leads to two issues. First, each executed task may stay in a different execution state. How to keep track their execution status? Second, no

matter how the execution flow traverses down the task tree, it eventually needs to backtrack to the parent task to start next child task. The solution in uFlow is the context along with the context stack. There are two cases of task execution, simple or composite, when the execution flow traverses down the tree. Simple tasks are always leaf nodes of the tree; hence no more than one simple task is executed at the same time. However, execution of tasks may be nested, since a child task of a composite task can also be a composite task and so forth. Furthermore, a child composite task can be the same type of task as its parent, that is, recursion.

Parameter Passing in uFlow

Task acts like a subroutine, which can accept input parameters and return multiple values in the form of output parameters.

Execute uFlow Programs

uFlow engine executes the application in a manner similar to tree traversal. Instead of in a strict order (e.g., preorder, inorder or preoder), it traverses in task-defined order.

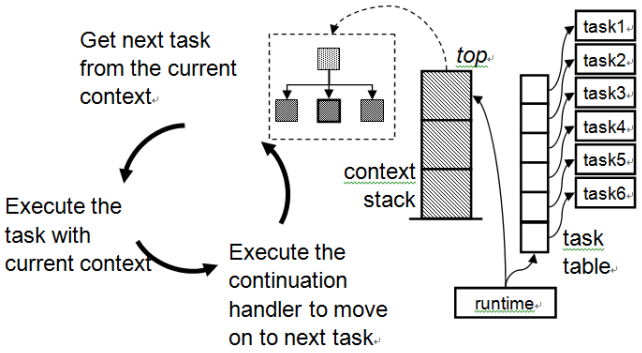

Fig.7. uFlow Task Loop

The uFlow task loop, as illustrated above, manages to execute the entire program, namely tasks. Without loss of generality, the task loop involves three major parts:

1. Looks up the current context to get the next task.
2. Executes the task by invoking the execution handler.
3. Move on the next task by invoking the continuation handler.

Dynamic Replacement

uFlow provides the ability to dynamically update the application behavior by changing the task structures. Structural changing includes adding new tasks, removing unwanted tasks, and even altering the task implantation on the fly without neither stopping the execution before updating nor restarting the program after update to preserve the precious runtime status. The update scenarios are classified into three cases depending on what task structures are modified. In either case, since all the

uFlow programs and engine are executed in a single thread, one can easily define inter-task invariants to guarantee the modification will not break the uFlow execution. The following sections discuss the updating procedure in terms of each case.

4 Prototype Implementation and Result

4.1 Engine Task Loop

The engine task loop is illustrated, in Figure 4 1, to explain its implementation code.

4. The context records the execution status of the current executed composite task. If the context is valid, there is a task waiting for execution. Otherwise, the program is terminated.
5. The engine looks up the context to get the next task and executes it by invoking its execution handler.
6. After execution, the execution result shows whether the newly executed task is a composite task. If not, the continuation handler is called to designate the next task.
7. If the newly executed task is a composite task. It is responsible for pushing a new context of it and designating the first child for execution. By this means, the first child of the newly composite task will be executed at the next iteration.
8. If a composite task is finished, the engine will pop out its context, and invokes the continuation handler of its parent to schedule next task.

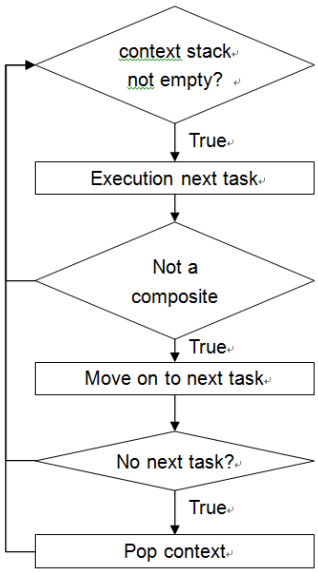

Fig.8. Change the transition rule

4.2 Engine Task Loop

The example is illustrated below. It repeatedly read to input from user, and sum them up. If the summation is an odd number, it prints "Number %d is Odd" ; otherwise, it prints "Number %d Even" where “%d” will be replaced by the summation. The example demonstrates how to write an application using uFlow and parameter binding in uFlow.

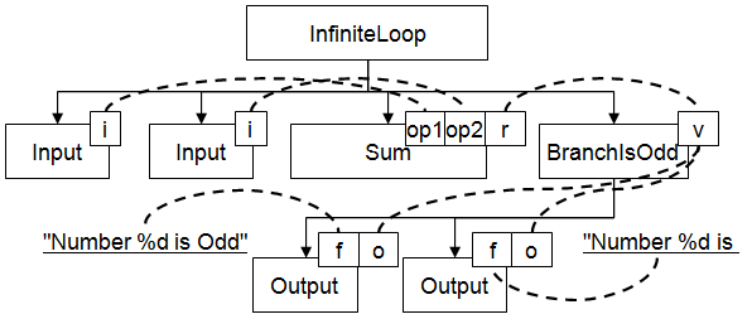

Fig.9.. uFlow example application

4.3 Results

In the following evaluation, the code is compiled with IAR [23] C/C++ Compiler for MSP430 4.21.2 with Debug configuration without optimization. MSP430 is a microcontroller from Texas Instruments. It is designed for low power consumption embedded applications, and widely used in various wireless sensor network environments. For example, the famous Telos from UC and its compatible motes, Tmote Sky from Sentilla and TelosB from Crossbow, are all equipped with MSP430F1611 microcontroller. The compiler is developed by IAR system, a Swedish company, one of the leadership companies providing compilers, debuggers and other tools for various microcontrollers. The compiler is chosen because it is officially suggested by Texas Instruments.

Engine Code Size

The code sizes for required and optional functions are 420 bytes and 146 bytes respectively. The size of each function in detail is listed in the tables below.

Table3. Code size for required functions

Function	Size (in bytes)
Engine task loop	108
uFlowCompositeTaskExec	24
uFlowGetParameters	38
uFlowGetValue	96
uFlowSetValue	50
uFlowPushContext	68
Total	420

Table 4. Code size for optional helper api

Function	Size (in bytes)
uFlowSetTransition	28
uFlowAllocSimpleTask	20
uFlowAllocCompositeTask	66
uFlowAllocTransitions	32
Total	146

uFlow Program Size

Program size is contributed from the code for execution and related data structures for uFlow. The code may be a simple task which implements a specific work, or a composite task implementation for control flow. No matter using uFlow or not, the code for simple task is usually required, because it is used to actually complete specific task. For example, reading sensor, transmitting packet. However, the code for composite task is often an overhead, because it is used for control flow. In Table5 show in order to organize the task, a collection of data structures are needed to describe to task and their transitions. Hence they are all overhead to the application. The Table 6 shows the size for a specific type of task is fixed no matter how many times it is used. Instead, a transition structure is needed to refer the task.

Table 5. The code size of the example application

Function	Task Type	Size (in bytes)
OutputTask	Simple	34
SumTask	Simple	52
InputTask	Simple	52
BranchIsOddExec	Composite	94
BranchIsOddCont	Composite	12
MainLoopCont	Composite	34
Total		278

Table 6. Required data structure size for the application

	Entry	Task	Transition	Parameter	Total
Output	2	2	0	4	8
Sum	2	2	0	6	10
While	2	10	24	0	36
Branch	2	10	26	2	40
Total	10	26	50	14	100

Execution Overheads

When executing uFlow program, the engine task loop traverses the tree to find next task for execution. The task loop *per se* does not help to accomplish the work, therefore is an overhead which slows down the execution and wastes extra energy. We compare the overhead with TinyOS and Maté, because TinyOS is one of the most widely used execution environment in sensor networks while Maté show the impressive ability of software update. The overhead of TinyOS comes from the task scheduler. Instead of directly invoke a subroutine, a subroutine is wrapped as a task, and posted in the task queue. At later time, the scheduler pops the tasks for execution one by one. The overheads of Maté are caused by wrapping each instruction to a TinyOS task.The clock cycles for the uFlow task loop is measured and compared to the scheduler in TinyOS-1.1.15.

Table 7. Execution overhead of uFlow, TinyOS and Maté

System	Task Loop	Clock Cycles
uFlow	uFlow engine task loop	58
	Continuation Function	≈30
TinyOS	TOS_post	26
	TOSH_run_next_task	30
Maté	TOS_post	N × 26
	TOSH_run_next_task	N × 30

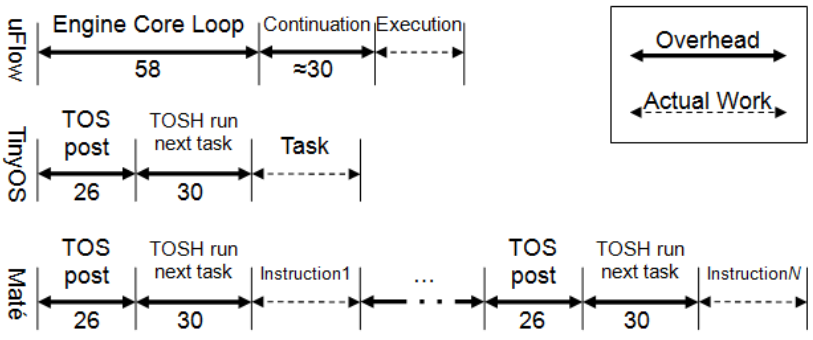

Fig.10. Execution overhead of uFlow, TinyOS and Maté

5 Conclusion

We present uFlow, a programming paradigm and a prototype implementation for wireless sensor networks. By uFlow, we allow the sensor network nodes to update without rebooting. Therefore the precious runtime status can be keep .we achieve the goal by decomposing the application into reusable task that organized as a tree structure. The engine executes the uFlow tree by traversing the task node and executing the task handlers. All the handlers are run-to-complete and never interrupted by other handlers. It is safe to replace or update tasks but does not interrupt microcontroller execution. Besides, all of the tasks are indirectly accessed via the task table. The Tasks are organized as a tree structure; these two properties keep update away from affecting the entire program and address relocation problems.

The prototype implementation result shows our idea works. The execution overhead in terms of clock cycle which costs around 58 for the engine task loop and around 30 for the extended handler. That is around 1.6 times for TinyOS task scheduler, i.e., around 88 clock cycles. Although the overhead is larger than TinyOS, but the overhead is only introduced at transition between tasks. The engine task loop only costs 420 bytes, helper API for dynamically creating uFlow application costs 146 bytes, the code size for the sample application costs 318 bytes, and the data structure used to describe the application costs 100 bytes. By our proposed idea, we can provide a software updating for sensor network and avoid the reboot.

Acknowledgments.
NSC 99-2221-E-002-106-MY3
MOTC-DPT-100-03 ◦

References

1. Chih-Chieh Han, Ram Kumar, Roy Shea, and Mani Srivastava: Sensor Network Software Update Management. Intl. Journal of Network Management, no. 15. (2005)
2. James W. Hunt and M. Douglas McIlroy: An Algorithm for Differential File Comparison. Computing Science Technical Report, Bell Laboratories 41. (1976)
3. Tridgell, A. and Mackerras, P. : The rsync algorithm. Tech. Rep. TR-CS-96-05, Canberra 0200 ACT, Australia. http://samba.anu.edu.au/rsync/.(1996)
4. C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In Proc. of the Sixth Annual International Conference on Mobile Computing and Networking (MobiCOM 2000), Boston, Massachussetts.(2000)
5. Crossbow Technology Inc. (XNP) Mote In-Network Programming User Reference. http://www.xbow.com.(2003)
6. T. Stathopoulos, J. Heidemann, and D. Estrin. (MOAP) A remote code update mechanism for wireless sensor networks. Technical report, UCLA, Los Angeles, CA, USA, (2003)
7. N. Reijers, and K. Langendoen. Efficient Code Distribution in Wireless Sensor Networks. In Proc. 2nd ACM international conference on Wireless sensor networks and applications (WSNA 2003), pages 60-67, (2003).

8. J. Jeong and D. Culler. Incremental Network Programming for Wireless Sensors. In Proc. of Sensor and Ad-Hoc Communications and Networks (SECON 2004), (2004).
9. J. Koshy, and R. Pandey. Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks. In Proc. of the second European Workshop on Wireless Sensor Networks, pages 354–365, (2005).
10. A. Dunkels , N. Finne , J. Eriksson , T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In Proc. of the 4th international conference on Embedded networked sensor systems (SenSys 2006), (2006).
11. M. Mukhtar, B.W. Kim, B.S. Kim, S.S. Joo. An Efficient Remote Code Update Mechanism for Wireless Sensor. In Proc. of Military Communications Conference (MILCON 2009), (2009).
12. D. Gay, P. Levis, R.V. Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In Proc. of Programming Language Design and Implementation (PLDI 2003), (2003).
13. P. Levis, D. Culler. Maté a tiny virtual machine for sensor networks. In Proc. of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X), (2002).
14. H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MANTIS: system support for MultimodAl NeTworks of In-Situ sensors. In Proc. Mobile Networks and Applications (WSNA 2003), (2003).
15. A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked. In Proc. of the 29th Annual IEEE International Conference on Local Computer Networks (LCN 2004), Pages: 455 - 462, (2004).
16. O. Gnawali , K.Y. Jang , J. Paek , M. Vieira , R. Govindan , B. Greenstein , A. Joki , D. Estrin , E. Kohler. The Tenet architecture for tiered sensor networks. In Proc. of the 4th international conference on Embedded networked sensor systems (SenSys 2006), (2006).
17. Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr. (Neutron) Surviving sensor network software faults. In Proc. of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP 2009), (2009).
18. A. Adya, J. Howell, M. Theimer, W.J. Bolosky, and J.R. Douceur. Cooperative Task Management Without Manual Stack Management. In Proc. of the General Track of the annual conference on USENIX Annual Technical Conference (USENIX 2002), (2002).
19. Dunkels , O. Schmidt , T. Voigt , M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In Proc. of the 4th international conference on Embedded networked sensor systems, October 31-November 03,(2006).
20. K. Klues, C.J. M. Liang, J. Paek, Musaloiu-E, P. Levis, A. Terzis, R. Govindan. TOSThreads: Thread-Safe and Non-Invasive Preemption in TinyOS. In Proc. of the 7th international conference on Embedded networked sensor systems (SenSys 2009), (2009).
21. Robert W. Sebesta: Concepts of Programming Languages, Sixth Edition. Addison Wesley. (2003).
22. Gamma, Erich; Richard Helm, Ralph Johnson, John M. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. pp. 395. ISBN 0-201-63361-2. (1995)
23. IAR Systems. http://www.iar.com/
24. Eric Steven Raymond. Art of UNIX Programming, The. Addison-Welsey. pp. 560. ISBN 0131429019.(1999)