2015 18th International Conference on Network-Based Information Systems

Efficient Remote Software Execution Architecture based on Dynamic Address Translation for Internet-of-Things Software Execution Platform

Minwoo Jung, Daejin Park*, and Jeonghun Cho+
School of Electronics Engineering, Kyungpook National University
Daehakro, Bukgu, Daegu, 702-701, Republic of Korea
**boltanut@knu.ac.kr,* +*jcho@ee.knu.ac.kr, +82-10-7529-1231*

***Abstract*—In this paper, an efficient remote code execution technique is proposed to implement a storage-less sensor in the internet-of-Things (IoT) paradigm. A statically installed sensor such as an environmental activity monitor includes statically compiled embedded software and only offers its pre-defined functionality. To realize a flexible code update mechanism and to optimize the use of an IoT device in terms of utilizing its various functions, we adopt the concept of remote on-demand code execution (ROCE) to implement a storage-less sensor. Instead of using conventional on-chip flash memory for an instruction code, an instruction memory is used wherein the remote storage area based on the IoT platform is virtually mapped onto the address space of the instruction memory using a dynamic address translation technique. A pervasive Internet-connected sensor enables on-demand code execution from the cloud-side remote storage resource, without the need for a direct instruction bus. The proposed storage-less approach using the remote resource as a virtual code space may be adopted to reduce the high access current and chip area overhead of an on-chip code flash memory. To reduce the access current overhead in order to retrieve the requested instruction, a small-sized RAM scratch pad is adopted for retaining the hot-spot instruction code. The experimental results show that the proposed technique reduces the energy consumption and packet delay of an IoT device for executing the remote embedded software, as well as realizing a storage-less sensor architecture.**

I. Introduction

In recent years, the internet-of-things (IoT) has become one of the most well-known technologies enabling the hyper-connection of embedded hardware, software, cloud infrastructure, and various application services. The IoT platform includes an application layer, a middleware layer, and a physical sensor network layer for basic machine-to-machine (M2M) communication. The sensor network layer serving as a physical interface is wrapped by a virtual software sensor layer.

This architecture ensures that the physical sensor interface is virtually linked with the middleware layer. The middleware layer communicates with virtual sensors and saves the sensed data to the server-side cloud. Further, this layer integrates the heterogeneous virtual sensors on a virtual plug-in interface with the low-level physical sensor layer and provides a pre-defined software interface in the application layer. To connect various sensors in the IoT, application developers indirectly access the virtual interface provided by the middleware layer.

The overall performance issues between devices in the IoT platform have been studied in terms of energy consumption reduction, security, congestion control, and packet delivery ratio. The IoT devices are implemented using the tiny embedded system based on a microcontroller including an on-chip flash memory for the embedded software area. Previous studies have been conducted on the use of these conventional on-chip flash embedded microcontrollers and have focused on the application layer.

Internet-connected IoT devices share data from the remote cloud server; but the embedded software code, which is statically compiled and downloaded, has difficulty updating the pre-defined functions. Our study proposes a newly designed remote software management technique for efficient instruction code management; this technique involves the modification of the on-chip software code bus architecture. The proposed physical sensor downloads executable code blocks for each function unit and stores these blocks in the internal static random-access memory (SRAM), eliminating the requirement of a large on-chip flash memory.

The area cost and large access current attributed to the on-chip flash memory are replaced by a small on-chip SRAM and on-the-fly internet connection overhead. Figure 1 shows existing remote software update and proposed architecture. The remainder of this paper is organized as follows. Section II discusses the motivation behind this study, as well as works related to it. Section III describes the details of the proposed technique and the system architecture of the proposed IoT device for software code execution. Section IV presents the implementation and measurement results. Finally, Section V provides the conclusion of this study.

II. Motivation and Related Work

There are many reasons why physical sensors need remote code updates in IoT. In the case of some deployment scenarios, physically reaching all physical sensors is impractical to the sensing process. However, it is critical to add or update the software on those physical sensors, after deployment. For this purpose, the developer must be able to develop a program for remotely updating the nodes and/or adding new

978-1-4799-9942-2/15 $31.00 © 2015 IEEE
DOI 10.1109/NBiS.2015.57

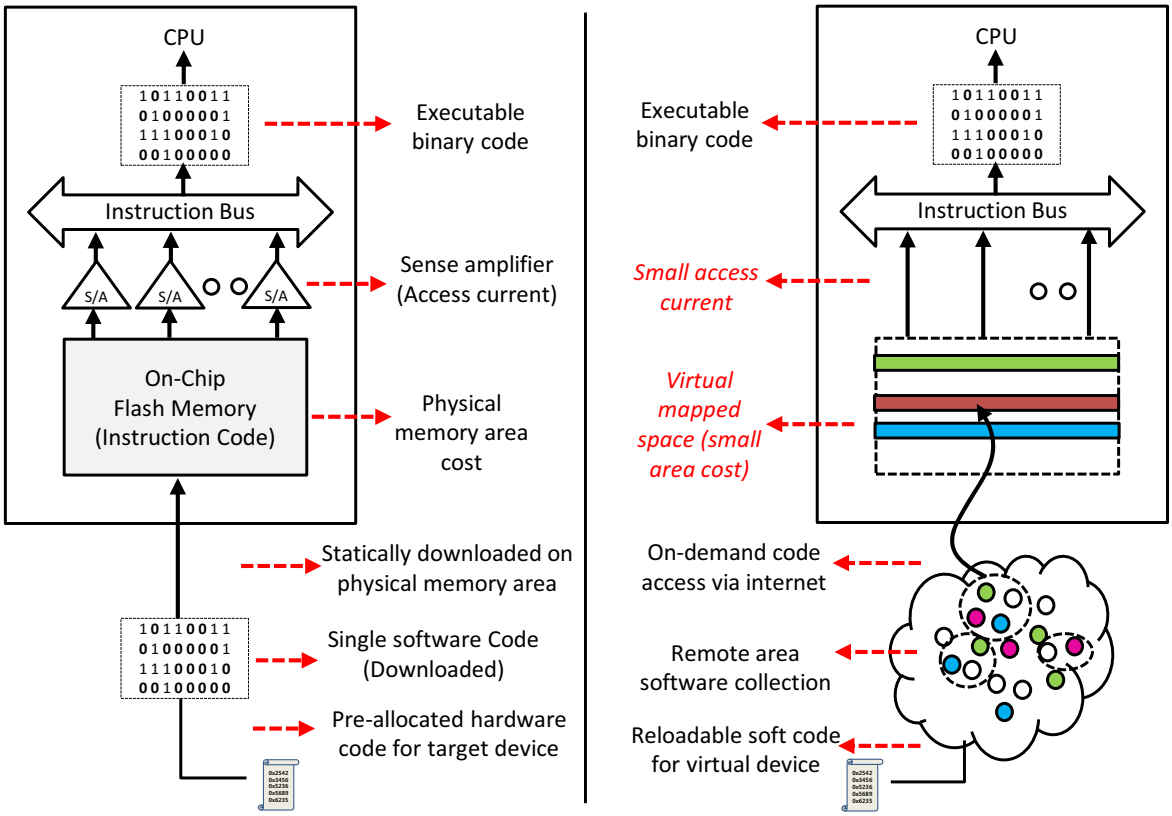


Figure 1. On-demand software execution framework architecture for efficient remote software management

functionalities. This feature is also beneficial for software maintenance and in-site debugging. A drawback of the physical sensors is that they have limited power; because of this, it is likely that an application running on the physical sensor may consume most of the memory of the physical sensors owing to data confidentiality and encryption. In such cases, it may not be feasible to transmit all of the newly updated code images in order to update the code of the physical sensor.

A drawback of the existing techniques in this regard is that they need to maintain the code image in the memory, which imposes a memory overhead on the physical sensor, which has a constrained memory. The sensor network device used in the IoT platform is implemented on a tiny embedded system as a physical layer. The embedded system includes an on-chip flash memory in which the software code is programmed. The long-term lifetime of an IoT device depends on the static current in the sleep mode and the dynamic operating current that is responsible for executing the embedded software in the active mode.

The dynamic operating current of the IoT device consists of the logic current and the code access current to the on-chip flash memory. The on-chip flash memory, in which the embedded software is installed, requires a large area of the entire IoT device chip and is responsible for the generation of a high access current in the sense amplifier in order to identify the programmed binary code. Several studies have been conducted on various approaches that can be used to reduce both the area overhead and the access current caused by the on-chip flash memory.

However, it is more important to realize both a smaller on-chip flash memory and a low access current in low-cost battery-operated IoT devices requiring ultra long-term activation. The proposed technique replaces the area occupied by the on-chip code flash memory with the Internet-connected remote area , which is a virtual code space. The instruction code blocks to be executed are dynamically transferred via the connectivity to the internal code scratch pad, which is implemented with a small-sized SRAM, but requiring frequency internet access to fetch newer instructions. The trade-off between larger SRAM and the network overhead will be considered in terms of hardware size and the operating energy consumption. In this study, we consider a method to reduce the energy consumption and memory overhead of physical sensors by the reduction of the compiled user code.

A. Storage-less remote code update

Software update has been an important area of interest in the IoT paradigm. Existing approaches for software update in wireless sensor networks can be classified into four main categories: full image replacement, differential image replacement, virtual machines, and dynamic operating systems. In the case of full image replacement, new binary images of both an application and the operating system in the network are transmitted. This method offers a very fine-grained control over possible reconfiguration, owing to the fact that the transmitted images are compiled and integrated for each iteration.

Although, these approaches result in bandwidth overhead owing to the fact that the unchanged parts of an application need to be re-transmitted in the network. W. Hui et al. [1] propose Deluge, a reliable data dissemination protocol for transmitting a large image object from one or more source nodes to many other sensors over a multi-hop wireless network. In the case of differential image replacement, the changes between an executable file deployed in the network and a new image are transmitted. While this reduces bandwidth consumption, the fundamental difficulty associated with replacing images exists. Panta et al. [2] present a multi-hop incremental reprogramming protocol.

Further, they propose Zephyr, which involves the transfer of the delta between the old and the new software and enables the sensor nodes to rebuild the new software using the received delta and the old software. In addition, it reduces the delta size by using application level modifications to mitigate the effects of function shifts. Then, it compares the binary images at the byte level with those obtained using a novel method to generate a small delta, which is then transmitted over the wireless network to all the nodes. Zephyr is advantageous over Deluge in that it causes less traffic through the network and it requires less time for reprogramming than the existing incremental reprogramming protocol.

However, this increases the similarity between the two versions and imposes additional memory and performance defects. Jeong et al. [3] present an incremental network-programming mechanism that reprograms wireless sensors quickly by transmitting the incremental changes for the new program version. This mechanism generates the difference between the two program images, enabling the distribution of only the key changes of the program, using the re-sync algorithm.

Further, this mechanism does not assume any prior knowledge of the program code structure, and it can be applied to any hardware platform. Virtual machines reduce the energy consumption associated with transmitting a new functionality in the network given that a virtual machine code is typically more compact than a native code. However, virtual machines generally allow only application updates and interpret the virtual machine codes. Therefore, they result in runtime overhead and decrease the lifetime of sensor nodes. Levis and Culler [4] propose Mate-a tiny communication-centric virtual machine designed for sensor networks. Its high-level interface allows complex programs to be very short (up to 100 bytes), reducing the energy cost of propagating new programs.

In the case of Mate, a code is broken up into small capsules of 24 instructions, which can self-replicate through the network. Packet transmission and reception capsules enable the deployment of ad-hoc routing and data aggregation algorithms. Its concise, high-level program representation simplifies programming and allows large networks to be frequently reprogrammed in an energy-efficient manner; in addition, its safe execution environment suggests the use of virtual machines to provide the user/kernel boundary on motes that have no hardware-protection mechanisms.

Muller [5] presents SwissQM, a virtual machine designed to address all limitations. SwissQM offers a platform-independent programming abstraction that is geared towards data acquisition and in-network data processing.

Brouwers [6] proposes a new virtual machine and tool chain that allow a significant subset of the Java language to execute on a microcontroller of the MSP430 or Atmega 128 class. Dynamic operating systems provide the advantages of both the image replacement approach and the virtual machines, in that they enable fine-grained code updates at low transmit and run-time overhead.

Dunkels [7] presents Contiki, a lightweight operating system that supports dynamic loading and replacement of individual programs and services. Although Contiki is built around an event-driven kernel, it provides optional pre-emptive multi-threading that can be applied to individual processes. Further, he proposes that dynamic loading and unloading are feasible in a resource-constrained environment and ensure that the base system is light in weight and compact. It is important to note that Contiki allows only one-way linking for loaded modules and obligates more energy-intensive, polling-based service routines for interrupts. Further, its architecture restricts possible reconfigurations to application components only.

Han [8] presents SOS, a new operating system for mote-class sensor nodes that takes a more dynamic point on the design spectrum. SOS consists of dynamically loaded modules and a common kernel, which implement messaging, dynamic memory, and module loading and unloading, among other services. However, it is necessary for the SOS to use a position-independent code because of compiler limitations, and it is not completely supported on common WSN platforms.

Mottola et al. [9] present FiGaRo, a programming model supported by an efficient run-time system and distributed protocols, collectively enabling an unprecedented fine-grained control over what is being reconfigured, and

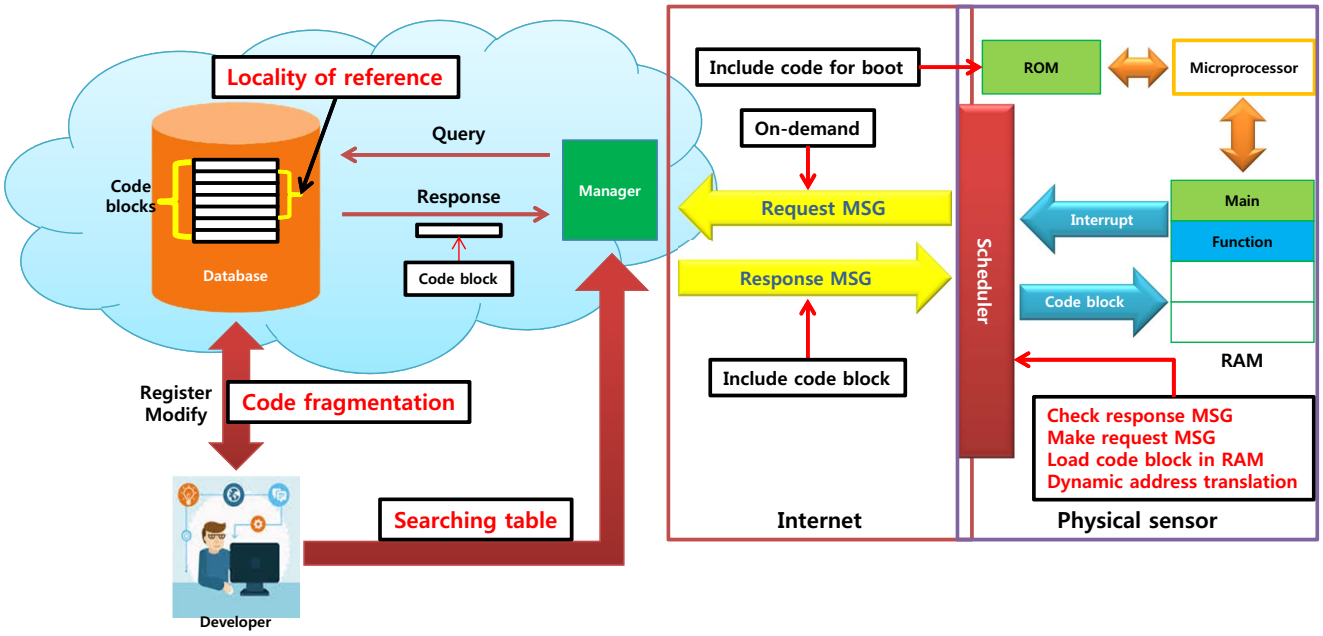


Figure 2. Overall proposed architecture

where. Using FiGaRo, the programmer can deal explicitly with component dependencies and version constraints, as well as precisely select the subset of nodes targeted by reconfiguration, leaving the others unaltered. The software update has been an important area of interest in IoT. Existing approaches for software update in wireless sensor networks can be classified into four main categories: full image replacement, differential image replacement, virtual machines, dynamic operating systems

III. Proposed Architecture and Techniques

We have proposed remote on-demand code execution (ROCE) to reduce the energy consumption and packet delay of physical sensors. Existing approaches for remote code update have assumed that a server provides the entire code file to a physical sensor, which has a large on-chip flash memory and adopts a complex procedure to update the code file. On the basis of locality of reference, ROCE has significantly reduced the number of codes that are transmitted from a server to a physical sensor. Given that a single code is partitioned into function block units , the developer should register and update the code block in the cloud server.

Furthermore, the developer should create an efficient search table to find a suitable code block in the database; this table will facilitate a reduction in the overhead of the physical sensor, because the sensor will wait for the fetch request of the next code block. The physical sensor loads the received code block on the small on-chip SRAM through a scheduler. The scheduler, which is the core component of the physical sensor, manages communication with the server. The functions of the scheduler are outlined as follows:

1) Generate a request message for the next function code
2) Check the response message
3) Load the received code block in the RAM after it has been checked
4) Translate the dynamic address

One of the most promising features of ROCE is that a single code is propagated with a code block unit. Apart from this feature, ROCE offers several other advantages such as minimal energy consumption and minimal delay. We can use locality of reference, which is a term describing the same value or related storage locations, being accessed frequently, for a single code to propagate the code block unit. The locality of reference facilitates the implementation of a memory-less sensor.

ROCE can reduce the network overhead, which can occur because of the transmission of large data blocks. Existing architectures propagate the entire single code in an executable and linkable format (ELF). Therefore, these architectures should use an on-chip flash memory to store the ELF, which results in the generation of a larger overhead than that caused by ROCE, owing to the fact that redundancy codes are also included in the ELF. Energy consumption can be reduced by eliminating the flash memory and by decreasing the code size. Figure 2 shows the architecture of ROCE.

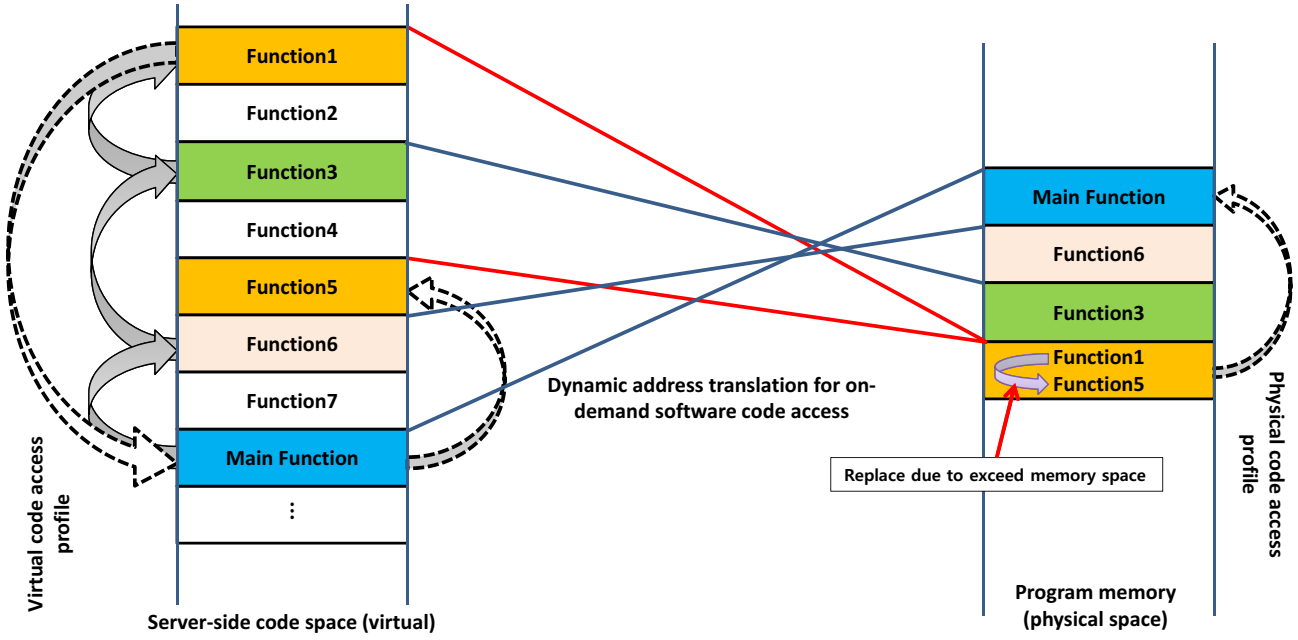


Figure 3. On-demand instruction code management for storage-less IoT device

A. Requirements

ROCE should be designed such that the following requirements are met:

1) The code block, which is transmitted from a server in the network, must reach all the physical sensors over the Internet.
2) An execution file should be stored as the function block unit.
3) The operation of ROCE should not influence the lifetime of the network (minimization of energy consumption).
4) Memory usage should not be very high, because ROCE does not include a memory.

Further, in order to ensure the correct operation of ROCE, the following requirements should be satisfied. First, in order to construct a sensor network for the IoT system, a reliable remote code update is required to ensure that the code block is transmitted to all physical sensors in a stable network environment.

B. Locality of reference

The principle underlying locality of reference implies that an application does not access all of its data at once with equal probability. Instead, it accesses only a small portion of it at any given time. Because of this principle, we can reduce the amount of transmitting data, thanks to which we can be sure that the physical sensor can be implemented without memory to achieve remote code update and the packet delay can be reduced.

C. Dynamic binary code translation

The proposed approach assumes that the dynamically loaded software code bytes are relatively allocated on the physical code area of the target sensor. The conventional static compilation and linking approach for a new target device requires the programming of a flash memory by halting the previously installed IoT device. Although the on-the-fly software access reduces the overall performance of the system code execution, dynamic binary code translation enables the on-demand software code execution transparently without any system architecture modification. Figure 3 shows dynamic address translation.

IV. Implementation

ROCE involves the use of a server and various physical sensors. The ROCE behaves as both server and physical sensor. When a physical sensor requires other functions, it communicates with the server through ROCE. In this section, we deal with each behavior and interaction between a server and the physical sensors.

A. Physical sensor design for ROCE

The sensor adopts a particular procedure so that it can carry out ROCE. Once the sensor is turned on, the hardware resources are initialized. The sensor first requests a global variable, stack size, and an interrupt vector table. After the sensor is initialized, it requests for the main function block, which once received is entered into the scheduler. The scheduler performs various functions such as frame check and binary translation. When the binary code of the main

function block is transferred to the RAM, the main function block begins to execute instructions.

When jump instruction is satisfied, the program counter (PC) shifts to the scheduler function. It calculates the start address of the next function block in the server and checks whether or not the function block exists in the schedule table. If the function block exists in the schedule table, it will be executed. If not, a request for the function block is sent to the server. The sensor carries out this procedure repeatedly. The function first checks the start bit and end bit. Then, these bits are compared with those listed in the schedule table through the extracted address in order to check whether a duplicate function block exists or not.

If a duplicate function block exists, the GetRecInfo function is executed. If it does not exist, the acquired information is added to the schedule table, and then, the GetRecInfo function is executed. The GetRecInfo function carries out binary translation and transfers the received function block to the RAM. Binary translation is carried out when the received frame comprises a jump or branch instruction. The jump or branch instructions are replaced with new binary that can move to the start point of the scheduler. Another function transfers received function block to the RAM after checking the received frame. The sensor includes a temporary buffer in order to check the received frame. Algorithm 1 shows the internal procedure of the physical sensor.

Algorithm 1 Internal procedure of physical sensor

procedure $Start - up(Stacksize, IRQtable)$
if $first\ request\ function$ **then**
  $START\ BIT = Request\ message[0][0X88]$
  $END\ BIT = Request\ message[6][0XFCFC]$
  $ID = Request\ message[1][0X00]$
  $Address = Request\ message[2][0X00000000]$
else
  $START\ BIT = Request\ message[0][0X88]$
  $END\ BIT = Request\ message[6][0XFCFC]$
  $ID = Request\ message[1][function_ID]$
  $Address = Request\ message[2][function_ptr]$
end if
for $scheduler$ **do**
  $check\ response\ message$
  $load\ code\ block\ in\ SRAM$
end for
for $execution$ **do**
  if $meet\ function\ call$ **then**
    $move\ scheduler$
    if $exist\ function\ in\ scheduler\ table$ **then**
      $move\ execution$
    else
      $move\ request\ function$
    end if
  else
    $maintain\ execution$
  end if
end for
return $status$

B. Server design for ROCE

The server required for ROCE is designed by a developer. The developer develops programs that are unique to a given sensor. The program is stored in the server, and users access the files required to execute the program over the Web protocol. The developer should provide an appropriate file for the start-up of the sensor, containing attributes such as the global variable, interrupt vector table , and stack size. The developer also creates a search table for a request message.

This table includes the function name, the address of the function block, and the function size. The developer should reference the ELF file, which is generated as a binary file, in order to create the search table. The ELF file defines the entire execution protocol. The server searches for the requested function block from the sensor in the search table. Subsequently, the server generates a response message after extracting the function block.

V. Evaluation

We consider an 8-MHz TI MSP430 microcontroller with a Chipcon CC2420 IEEE 802.15.4 radio transceiver in order to verify the effect of ROCE. For the sake of evaluation, we consider the communication between a single test sensor node and a sink node. We implement the ELF files such that their sizes vary with the number of functions. Next, we extract the code size for locality of reference. We then compare the proposed ROCE with the existing remote code update system in terms of energy consumption and packet delay. Energy is consumed when the physical sensor receives and processes a code. The packet delay refers to the time spent transmitting the required code.

A. Energy consumption of ROCE

We compared the energy consumption of OBE with that of the existing method. We implemented OBE on TelosB platform. TelosB includes MSP430 as micro-processor and CC2420 as zigbee transceiver. We referenced specification of TelosB. We defined energy consumption as the sum of current during a specific time. We calculated energy consumption when a physical sensor received a file for execution.

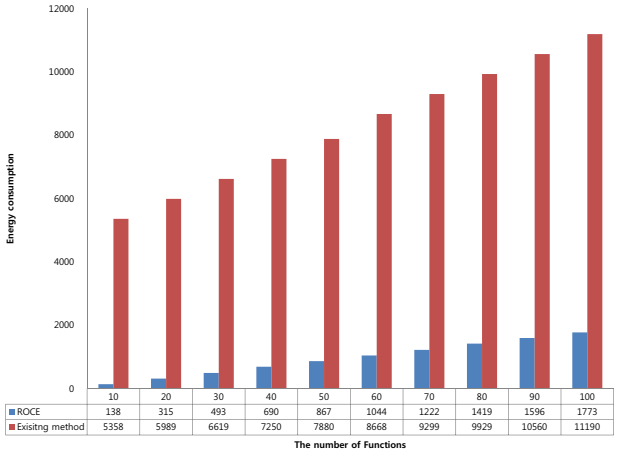
$$E_{exe} = (N_f * E_{RX}) + (S_L * E_p) \quad (1)$$

N_f is the number of frames needed, which is transmitted from server to physical sensor. E_{Rx} is value of energy consumption when the physical sensor receives a frame. S_L is the size of the function needed for execution. E_p is

the value of energy consumption when the physical sensor processes an execution file. Most physical sensors for WSN have a payload, which is the fraction of frame for data transmission. We could calculate the number of frames needed using payload. We extracted size of ELF file and functions for execution.

$$N_f = S_E / S_p \tag{2}$$

S_E is the size of the ELF file, Sp is the size of payload. We considered various sizes of needed functions for execution. Existing methods assume that the physical sensor receives the entire ELF file. However, OBE only needs fractions of the ELF file for execution. We observed that energy consumption can be drastically reduced. Figure 4 shows energy consumption between the existing method and ROCE.



	10	20	30	40	50	60	70	80	90	100
ROCE	138	315	493	690	867	1044	1222	1419	1596	1773
Exisitng method	5358	5989	6619	7250	7880	8668	9299	9929	10560	11190

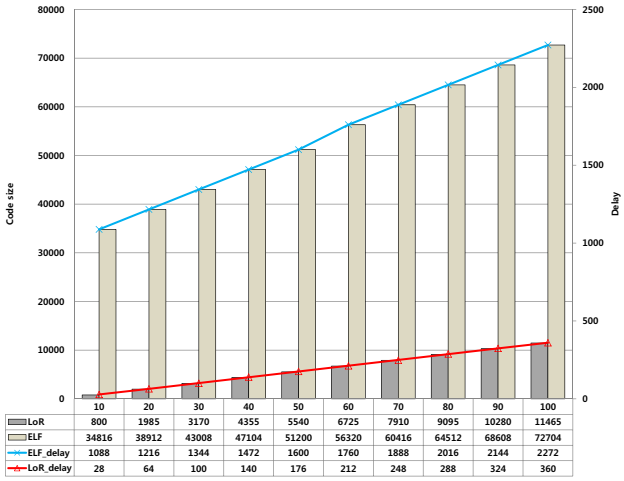
Figure 4. Comparison of energy consumption between existing method and ROCE

B. Packet delay of ROCE

The packet delay means the duration to complete the transmission of an entire file for the execution of a physical sensor. We calculated the packet delay using frame size.

$$D_p = N_f * T \tag{3}$$

D_p is the packet delay, N_f is the number of frames transmitted from server to physical sensor. T is the time for a frame to be transmitted. We considered various numbers of functions needed for execution. Existing methods assume that the physical sensor receives an entire ELF file. However, OBE only needs fractions of ELF files for execution. We observed that packet delay can be drastically reduced. Figure 5 shows reduction of packet delay and code size between the existing method and ROCE.



	10	20	30	40	50	60	70	80	90	100
LoR	800	1985	3170	4355	5540	6725	7910	9095	10280	11465
ELF	34816	38912	43008	47104	51200	56320	60416	64512	68608	72704
ELF_delay	1088	1216	1344	1472	1600	1760	1888	2016	2144	2272
LoR_delay	28	64	100	140	176	212	248	288	324	360

Figure 5. Reduction of packet delay and code size

VI. Conclusion

We have proposed ROCE to efficiently manage physical sensors. The characteristics of ROCE are outlined as follows.

- Storage-less sensor: ROCE facilitates the elimination of flash memory from a physical sensor, implying that the storage-less physical sensor will have lower power consumption and a smaller size than the existing sensor.
- Efficient management of physical sensors: ROCE can enable the storage of all the information pertaining to the physical sensors to the cloud server. Further, it can control and easily monitor the sensor network in the IoT platform. It can also easily update the physical sensors.

ROCE can be used in several technologies in the future, one of them being fragmentation technology. In this technology, a single code of a physical sensor is stored as a function block. A drawback of such a type of storage mechanism is that memory is used inefficiently. If a block is fixed, the reference-paging method can be used. In the future, we intend to conduct studies on a hybrid fragmentation method reducing the internal memory area, still requiring small bandwidth for instruction code access.

Acknowledgment

This research was supported by the Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(2014R1A6A3A04059410).

References

[1] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, ser. SenSys '04. New York, NY, USA: ACM, 2004, pp. 81–94. [Online]. Available: http://doi.acm.org/10.1145/1031495.1031506

[2] R. K. Panta, S. Bagchi, and S. P. Midkiff, “Zephyr: Efficient incremental reprogramming of sensor nodes using function call indirections and difference computation.”

[3] J. Jeong and D. Culler, “Incremental network programming for wireless sensors,” in *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on*, Oct 2004, pp. 25–33.

[4] P. Levis and D. Culler, “Mate: A tiny virtual machine for sensor networks,” 2002.

[5] R. Müller, G. Alonso, and D. Kossmann, “A virtual machine for sensor networks,” *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 145–158, Mar. 2007. [Online]. Available: http://doi.acm.org/10.1145/1272998.1273013

[6] N. Brouwers, P. Corke, and K. Langendoen, “A java compatible virtual machine for wireless sensor nodes,” in *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys ’08. New York, NY, USA: ACM, 2008, pp. 369–370. [Online]. Available: http://doi.acm.org/10.1145/1460412.1460456

[7] A. Dunkels, B. Gronvall, and T. Voigt, “Contiki - a lightweight and flexible operating system for tiny networked sensors,” in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, Nov 2004, pp. 455–462.

[8] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, “A dynamic operating system for sensor nodes,” in *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’05. New York, NY, USA: ACM, 2005, pp. 163–176. [Online]. Available: http://doi.acm.org/10.1145/1067170.1067188

[9] L. Mottola, G. P. Picco, and A. Amjad, “Figaro: Fine-grained software reconfiguration for wireless sensor networks.”