Simulation Modelling Practice and Theory 000 (2016) 1–11

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

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

# Area efficient remote code execution platform with on-demand instruction manager for cloud-connected code executable IoT devices

Daejin Park*, Minwoo Jung, Jeonghun Cho

*School of Electronics Engineering, Kyungpook National University, Daegu, Korea*

ARTICLE INFO

*Article history:*
Available online xxx

*Keywords:*
Remote-code execution
Internet-of-Thing
Virtual memory map
Instruction memory
Scratch pad

ABSTRACT

An energy-area efficient cloud-connected software execution architecture in IoT sensor processor is proposed. A remotely installed sensor device such as an environmental activity monitor is commonly implemented using the conventional embedded processor only providing the fixed services, which includes statically compiled embedded software in on-chip flash memory. Instead of conventional on-chip flash memory for an instruction code area, we adopt an virtually mapped internal memory concept to realize cloud-connected software execution, in where the remote storage area via the IoT platform is indirectly mapped onto the physical address space of the instruction memory using a dynamic address translation technique. The proposed cloud-connected architecture of the system enables on-demand code execution for the instructions, which are fetched from the cloud-side remote storage area in the runtime, instead of using a directly-connected on-chip instruction bus. The proposed storage-less approach may be adopted to reduce the high access current and large chip area overhead by eliminating the 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 and early filled with pre-estimated instruction sector. 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 the reduced chip area by realizing a storage-less sensor architecture.



## 1. Introduction

In recent years, the internet-of-things (IoT) has become one of the most well-known technologies that enables the hyper-connection of embedded hardware, software, cloud infrastructure, and various application services [1]. 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, which serves as a physical interface is wrapped by a virtual software sensor 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 [2]. The IoT devices are implemented using a tiny embedded system based on a microcontroller that includes an on-chip flash memory for the embedded software area [3]. Previous

* Corresponding author.
*E-mail address:* boltanut@knu.ac.kr (D. Park).

http://dx.doi.org/10.1016/j.simpat.2016.08.010
1569-190X/

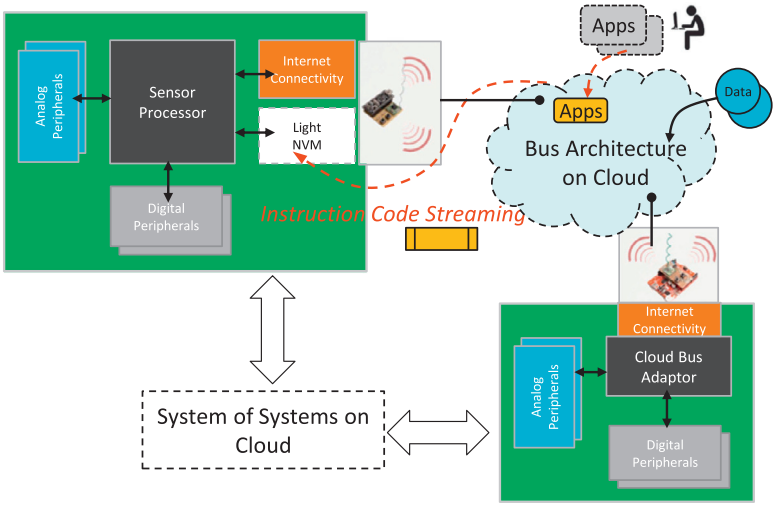



**Fig. 1.** Proposed virtual bus architecture for system-on-cloud.

studies have been conducted on the use of these conventional on-chip flash embedded microcontrollers [4] and have focused on the application layer.

The tiny embedded systems based on the low-cost microcontrollers are operated by executing the user software code, which is downloaded and programmed into the on-chip flash memory [5]. The standalone system, which includes on-chip instruction storage, has advantages when it comes to performing various reprogrammable functions, but this causes several drawbacks in the IoT applications:

1. On-chip flash memory size is most most dominant of the entire chip cost, so only a small area is allocated, which limits various services.
2. The static user codes, which are compiled, downloaded, and programmed code in the on-chip flash memory, have difficulty updating, in the case of remotely installed devices.

The research topic of this paper is newly designed code execution architecture that is optimized to the IoT-driven applications to overcome these issues. Instead of using the code embedded architecture, we adopt memoryless architecture based on a cloud-connected code execution.

Internet-connected IoT devices [6,7] share data from the remote cloud server. However, the embedded software code, which is statically compiled and downloaded, has difficulty updating the predefined 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.

As shown in Fig. 1, the proposed physical sensor downloads executable code blocks for each function unit and stores these blocks in the internal static random-access memory (SRAM) or small size non-volatile memory (NVM), thereby eliminating the need for a large on-chip flash memory. The wireless connection can be considered as virtual bus architecture between IoT devices. The code memory data, which is being streamed via internet connectivity, is virtually mapped from the internet-connected data stream.

User codes are stored in the cloud-side common space, and they are indirectly loaded into the on-chip memory space when the current program counter points to a newer address. Using the proposed memory space translation method, conventional system-on-chip (SoC) architecture is extended by the system-on-cloud concept as system-of-systems topology, in which an IoT device is a building block in SoC, and the system bus is replaced by the wireless interconnection across the IoT devices.

Fig. 2 shows the data-path implementation of the custom-designed MCU core for the proposed cloud-connected code execution. The virtual instruction memory does not need a physical flash storage with a fixed code space size. The code space view in terms of the CPU code access is the entire space of the cloud area. Actually, a physical space with a small memory size is used to hold the instructions of a recently accessed code sector. During CPU access of the virtual memory, the expected areas of code memory are accessed and filled into secondary areas in zigzag style to reduce the latency of fetching the currently accessed instructions.

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 [8]. Fig. 3 shows existing remote software updates and proposed architecture. The remainder of this paper is organized as follows. Section 2 discusses the motivation behind this study, as well as works related to it. Section 3 describes the details of the proposed technique and the system architecture of the proposed IoT device

Please cite this article as: D. Park et al., Area efficient remote code execution platform with on-demand instruction manager for cloud-connected code executable IoT devices, Simulation Modelling Practice and Theory (2016), http://dx.doi.org/10.1016/j.simpat.2016.08.010

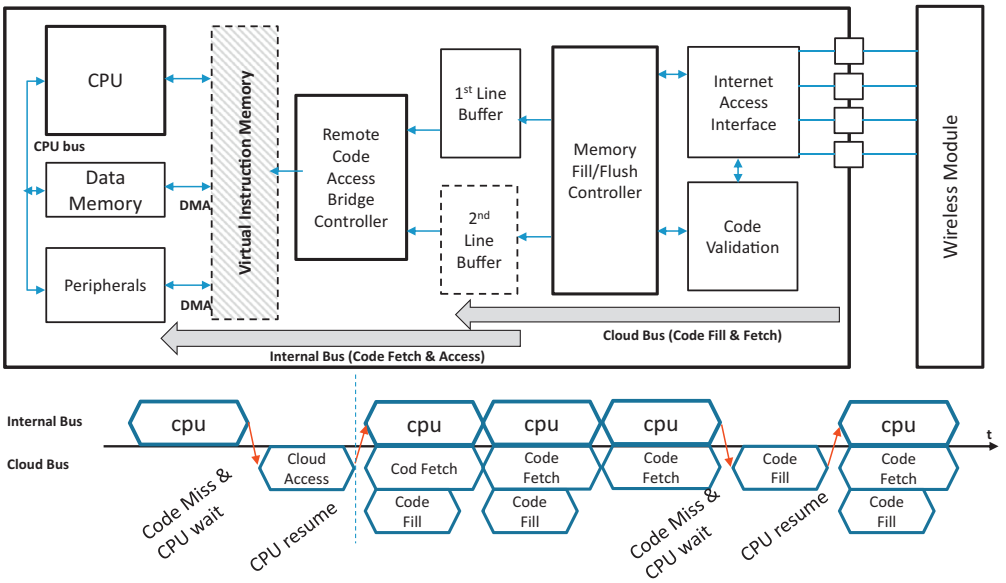



**Fig. 2.** On-chip architecture for virtually mapped code memory space.

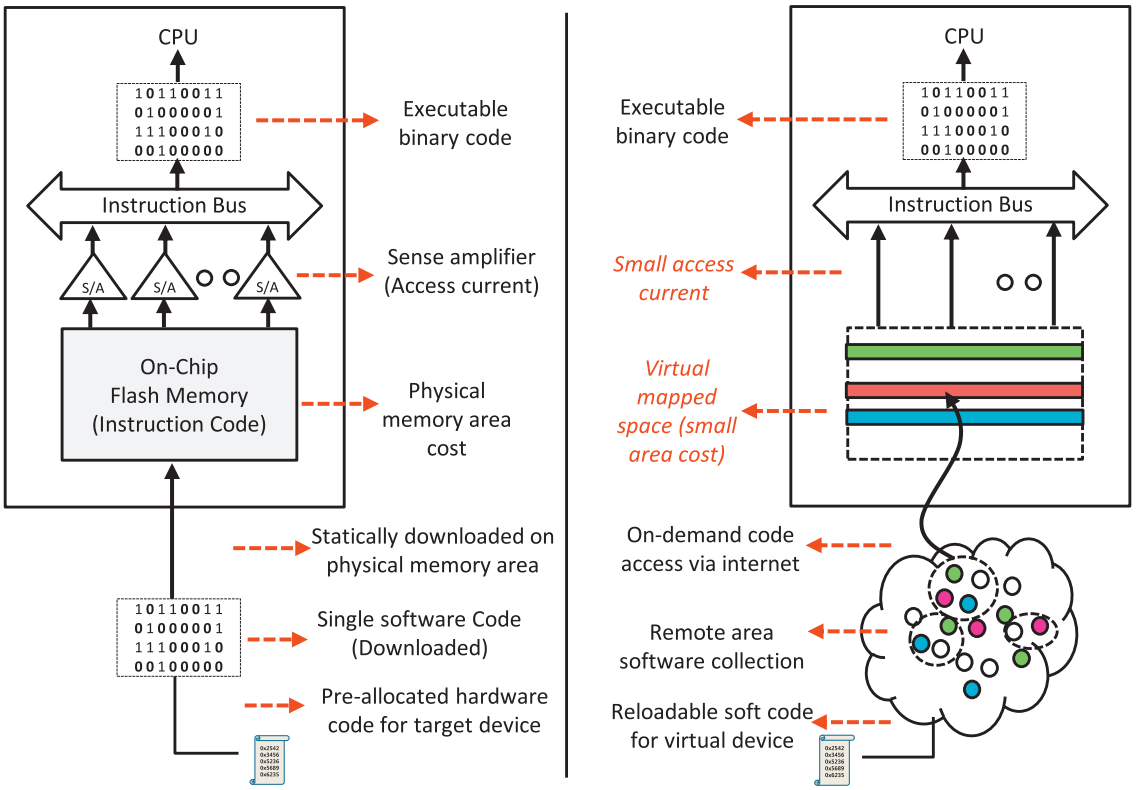



**Fig. 3.** On-demand software execution framework architecture for efficient remote software management.

for software code execution. Section 4 and 5 present the implementation and measurement results. Finally, Section 6 details the conclusion of this study.

Please cite this article as: D. Park et al., Area efficient remote code execution platform with on-demand instruction manager for cloud-connected code executable IoT devices, Simulation Modelling Practice and Theory (2016), http://dx.doi.org/10.1016/j.simpat.2016.08.010

## 2. Motivation and related work

There are many reasons why physical sensors need remote code updates in IoT. In deployment scenarios, physically reaching all physical sensors is impractical [9]. However, it is critical to add or update the software on those physical sensors after deployment. For this reason, the developer must be able to create a program for remotely updating the nodes and/or adding new functionality.

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 its memory due to data confidentiality and encryption [10]. 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.

The size and encoding format of binary codes, which are loaded on demand from the server, affect the necessary communication bandwidth and on-chip buffer space, such as a scratch pad. The machine-independent bytecode is preferred due to its compact code size and manipulability.

In our initial approach, we assumed some constraints of the application-specific MCU core. For instance, in our proposed method, the ARM Cortex-M0 should not be larger due to additional interpretation overhead, like bytecode decoder [11]. We also assumed that the modified CPU should remain backward compatible with its ecosystem (e.g., a debugger or previously compiled library). In this manner, our approach directly parses the fractions of the target-dependent static binary code and reallocates them into the limited area of the on-chip buffer [12]. Additional tasks on the server side are reduced by the utilization of the existing target-dependent compiler collections.

A drawback of the existing techniques in this regard is that they need to maintain a code image in the memory that 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 operating lifetime of an IoT device depends on the static current in sleep mode and the dynamic operating current that is responsible for executing the embedded software in active mode [13,14].

The dynamic operating current of the IoT device consists of the logic current and 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 generating 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 reduce both the area overhead and the access current caused by the on-chip flash memory [15,16].

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 for ultra long-term activation [3,17]. 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 connectivity with the internal code scratch pad, which is implemented with a small-sized SRAM but requires frequent internet access to fetch newer instructions. The trade-off between larger SRAMs and the network overhead will be considered in terms of hardware size and the operating energy consumption. In this study, we research a method of reducing the energy consumption and memory overhead of physical sensors through the reduction of the compiled user code.

## 3. 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 updates 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. Generates a request message for the next function code
2. Checks the response message
3. Loads the received code block in the RAM after it has been checked
4. Translates 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, including minimal energy consumption and minimal delay. We can use locality of reference, which is a term that describes the same value or related storage locations, being accessed frequently,

Please cite this article as: D. Park et al., Area efficient remote code execution platform with on-demand instruction manager for cloud-connected code executable IoT devices, Simulation Modelling Practice and Theory (2016), http://dx.doi.org/10.1016/j.simpat.2016.08.010

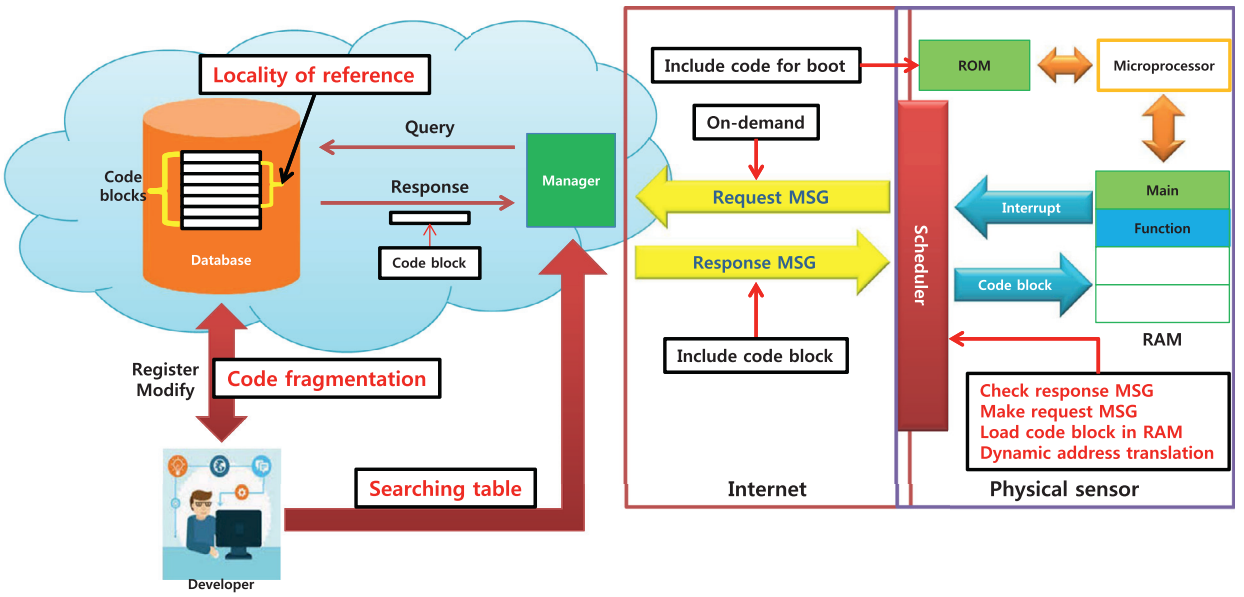



**Fig. 4.** Overall proposed architecture.

for a single code to propagate in the code block unit. The locality of reference facilitates the implementation of a memoryless sensor.

ROCE can reduce 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 would result in the generation of a larger overhead than that caused by ROCE, due to the fact that redundancy codes are also included in the ELF. Energy consumption can be reduced by eliminating the flash memory and decreasing the code size. Fig. 4 shows the architecture of ROCE.

### 3.1. 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 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.

### 3.2. 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, thereby ensuring that the physical sensor can be implemented without memory to achieve remote code update and reduce the packet delay.

### 3.3. 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 flash memory by halting the previously installed IoT device. Although 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. Fig. 5 shows dynamic address translation.

Please cite this article as: D. Park et al., Area efficient remote code execution platform with on-demand instruction manager for cloud-connected code executable IoT devices, Simulation Modelling Practice and Theory (2016), http://dx.doi.org/10.1016/j.simpat.2016.08.010

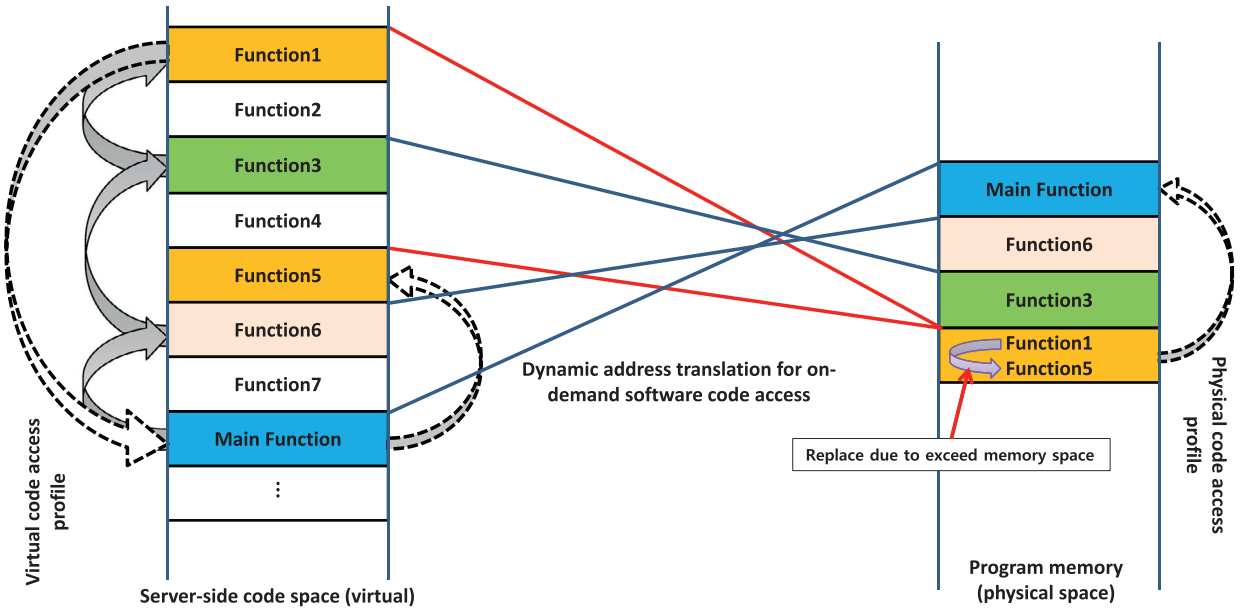



**Fig. 5.** On-demand instruction code management for storage-less IoT device.

## 4. 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 servers and physical sensors.

### *4.1. Physical sensor design for ROCE*

The sensor adopts a particular procedure to carry out ROCE. Once the sensor is turned on, the hardware resources are initialized. The sensor first requests a global variable, stack size, and interrupt vector table. After the sensor is initialized, it requests for the main function block, which is entered into the scheduler once received. The scheduler performs various functions, including 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 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.

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 the received function block to the RAM after checking the received frame. The sensor includes a temporary buffer that checks the received frame. Algorithm 1 shows the internal process of the physical sensor.

### *4.2. Server design for ROCE*

The server required for ROCE is designed by a developer. The developer creates programs that are unique to given sensors. 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 that contains 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

Please cite this article as: D. Park et al., Area efficient remote code execution platform with on-demand instruction manager for cloud-connected code executable IoT devices, Simulation Modelling Practice and Theory (2016), http://dx.doi.org/10.1016/j.simpat.2016.08.010

**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
```

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.

## 5. 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.

### 5.1. Energy consumption of ROCE

We compared the energy consumption of MCU with that of the existing method. We implemented MCU-based IoT devices on the TelosB platform. TelosB includes MSP430 as microprocessor and CC2420 as Zigbee transceiver. We referenced the specification of TelosB. We defined energy consumption as the sum of current during a specific time. We calculated the energy consumption of a physical sensor receiving a file for execution.

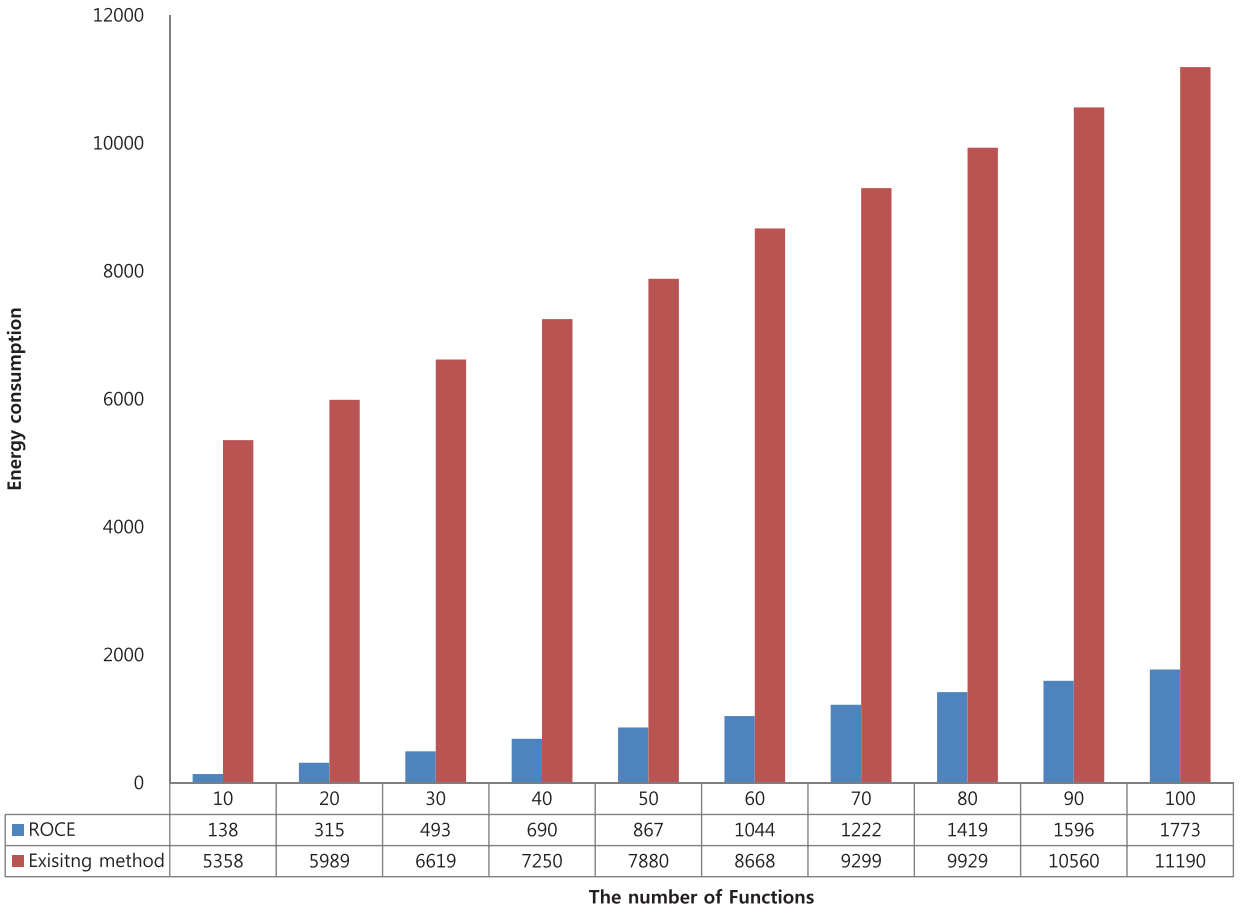
$$E_{exe} = (N_f * E_{RX}) + (S_L * E_p) \tag{1}$$

$N_f$ is the number of frames needed, which are 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 the size of the ELF file and functions for execution.

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

$S_E$ is the size of the ELF file and Sp is the size of payload. We considered various sizes of functions needed for execution. Existing methods assume that the physical sensor receives the entire ELF file. However, MCU only needs fractions of the ELF

Please cite this article as: D. Park et al., Area efficient remote code execution platform with on-demand instruction manager for cloud-connected code executable IoT devices, Simulation Modelling Practice and Theory (2016), http://dx.doi.org/10.1016/j.simpat.2016.08.010




| | 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 |

**Fig. 6.** Comparison of energy consumption between existing method and ROCE.

file for execution. We observed that energy consumption can be drastically reduced. Fig. 6 shows the difference in energy consumption between the existing method and ROCE.

### 5.2. Packet delay of ROCE

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

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

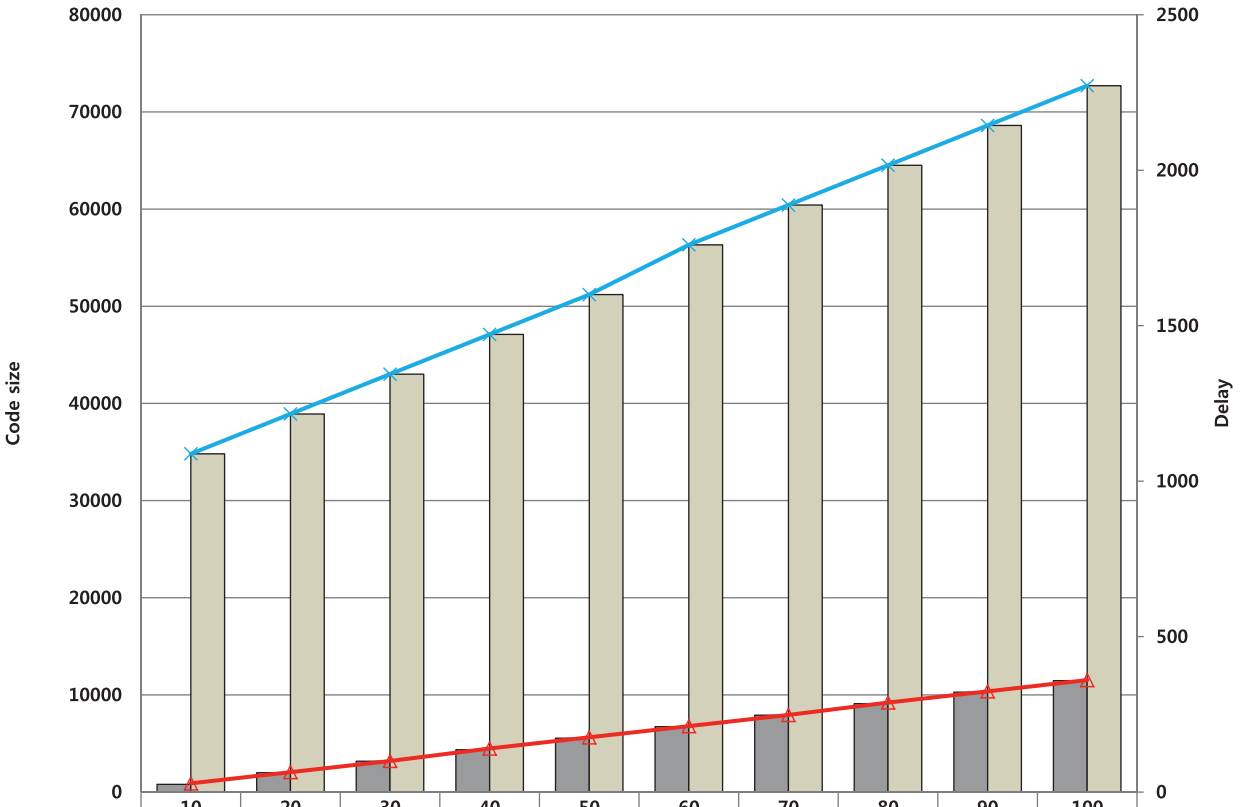
$D_p$ is the packet delay, $N_f$ is the number of frames transmitted from server to physical sensor, and T is the time in which a frame is transmitted. We considered various numbers of functions needed for execution. Existing methods assume that the physical sensor receives an entire ELF file. However, MCU only needs fractions of ELF files for execution. We observed that packet delay can be drastically reduced. Fig. 7 shows reduction of packet delay and code size between the existing method and ROCE.

### 5.3. Data communication overhead

The proposed IoT device architecture consumes operating current in communicating and processing data. There are two types of communication overhead in transferring code and exchanging data. The code access to the server happens in loading the code and reallocating the internal buffer to be executed.

This is explicit additional overhead in the proposed approach, but most time spent executing code involves hotspot code, such as loops. The repeated access to the hotspot does not require repeated communication in the code update. This means that the overall energy consumed when fetching code from the server becomes minor according to the time increase of the program execution.

Please cite this article as: D. Park et al., Area efficient remote code execution platform with on-demand instruction manager for cloud-connected code executable IoT devices, Simulation Modelling Practice and Theory (2016), http://dx.doi.org/10.1016/j.simpat.2016.08.010




| | 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 |

**Fig. 7.** Reduction of packet delay and code size.

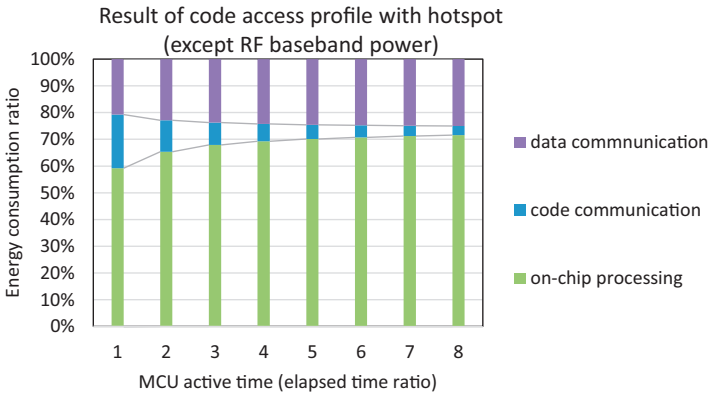



**Fig. 8.** Comparison of energy consumption ratio in updating code and exchanging data.

After the executed code fraction is reallocated into the line buffer, the CPU directly accesses the instructions to the on-chip buffer. The CPU interprets the instructions and then most of the code tries to exchange the required data with adjacent IoT devices. This means that data communication is more dominant in the entire code execution's lifetime.

The communication overhead is dependent on the code patterns and protocols used in exchanging data. We evaluated the communication overhead of the proposed MCU-based IoT device in terms of the time advance of code, shown in Fig. 8.

Please cite this article as: D. Park et al., Area efficient remote code execution platform with on-demand instruction manager for cloud-connected code executable IoT devices, Simulation Modelling Practice and Theory (2016), http://dx.doi.org/10.1016/j.simpat.2016.08.010

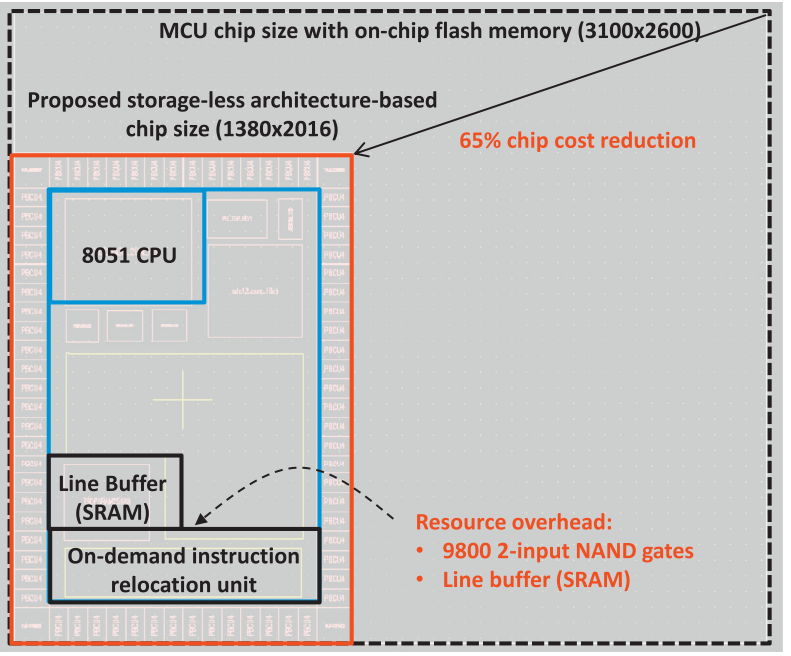



**Fig. 9.** Area cost reduction comparison of the proposed storage-less MCU architecture.

### *5.4. Chip area reduction*

Fig. 9 compares the area reduction of the fabricated MCU chip based on the proposed storage-less architecture including the additional on-demand instruction reallocation unit and internal line SRAM buffer. As the conventional on-chip flash size takes up most of chip, the proposed approach comes to reduce the chip cost by eliminating on-chip flash with an small additional hardware cost. The additional cost is needed for the bride controller to facilitate communication between the cloud and MCU internal bus. In the end, the proposed approach reduces chip area cost, and the trade-off of energy consumption overhead can be considered.

There are various case-by-case approaches [18,19] to targeting CPU-dependent optimization when implementing the runtime code replacement, which means the reference is not easily determined. The evaluation results only show a feasible approach to implementing a seamless, on-demand code streaming method. This method involves minimum modification of software due to using runtime binary address translation, reasonably small on-chip hardware support of about 10,000 NAND logic gates, and the elimination of on-chip flash in a cost-sensitive MCU application.

## 6. Conclusion

In this paper, we introduced our storage-less sensor processor concept and its implementation in the energy-efficient, cloud-connected software execution architecture. In traditional sensor processor MCU only provides a fixed service by the statically compiled and downloaded software, but the proposed sensor processor architecture tries to extend the software executable area to the cloud space. We proposed the revised instruction bus architecture and implemented memory management units based on the instruction access address translation algorithm.

The experimental results show a trade-off in benefits between eliminating the internal instruction memory and access current overhead through indirect code access to the cloud space. Though the proposed approach requires several latencies in the program execution and accesses current overhead to the external memory space, the reduced hardware cost of the fabricated prototype chip with no need for internal flash memory has a chance of boosting cloud-connected software execution in IoT applications.

In future work, further improvement in terms of code size and communication bandwidth reduction is needed for using the machine-independent byte code format and code compaction, such as bytecode, commonly used in virtual machines.

## 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).

Please cite this article as: D. Park et al., Area efficient remote code execution platform with on-demand instruction manager for cloud-connected code executable IoT devices, Simulation Modelling Practice and Theory (2016), http://dx.doi.org/10.1016/j.simpat.2016.08.010

## References

[1] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami, Internet of Things (IoT): a vision, architectural elements, and future directions, Future Generation Computer Syst. 29 (7) (2013) 1645–1660.
[2] D. Vande Ginste, H. Rogier, D. De Zutter, H. Pues, Efficient analysis and design strategies for radio frequency boards dedicated to integrity monitoring of integrated circuits using an electromagnetic/circuit co-design technique, Sci. Measure. Technol. IET 4 (5) (2010) 268–277, doi:10.1049/iet-smt.2010.0032.
[3] K. Leuenberger, R. Gassert, Low-power sensor module for long-term activity monitoring, in: Engineering in Medicine and Biology Society,EMBC, 2011 Annual International Conference of the IEEE, 2011, pp. 2237–2241, doi:10.1109/IEMBS.2011.6090424.
[4] N. Schemm, S. Balkir, M. Hoffman, An ultra low-power single chip intelligent sensing platform, in: Sensors, 2010 IEEE, 2010, pp. 1427–1430, doi:10.1109/ICSENS.2010.5690497.
[5] C. Cifuentes, V. Malhotra, Binary translation: static, dynamic, retargetable? in: Software Maintenance 1996, Proceedings., International Conference on, 1996, pp. 340–349, doi:10.1109/ICSM.1996.565037.
[6] J. Baliga, R. Ayre, K. Hinton, R. Tucker, Green cloud computing: balancing energy in processing, storage, and transport, Proc. IEEE 99 (1) (2011) 149–167, doi:10.1109/JPROC.2010.2060451.
[7] P. Ross, Top 11 technologies of the decade, Spectr. IEEE 48 (1) (2011) 27–63, doi:10.1109/MSPEC.2011.5676379.
[8] Y. Xiao, W. Li, M. Siekkinen, P. Savolainen, A. Yla-Jaaski, P. Hui, Power management for wireless data transmission using complex event processing, Comput. IEEE Trans. 61 (12) (2012) 1765–1777, doi:10.1109/TC.2012.113.
[9] M. Lazarescu, Design of a wsn platform for long-term environmental monitoring for iot applications, Emerg. Selected Top. Circ. Syst. IEEE J. 3 (1) (2013) 45–54, doi:10.1109/JETCAS.2013.2243032.
[10] D. Park, T.G. Kim, Safe microcontrollers with error protection encoder-decoder using bit-inversion techniques for on-chip flash integrity verification, in: Consumer Electronics (GCCE), 2013 IEEE 2nd Global Conference on, 2013, pp. 299–300, doi:10.1109/GCCE.2013.6664833.
[11] J. Ellul, K. Martinez, Run-time compilation of bytecode in sensor networks, in: Sensor Technologies and Applications (SENSORCOMM), 2010 Fourth International Conference on, 2010, pp. 133–138, doi:10.1109/SENSORCOMM.2010.28.
[12] C.-C. Lin, C.-L. Chen, C.-H. Tseng, Source code arrangement of embedded java virtual machine for nand flash memory, in: Communications and Information Technologies, 2007. ISCIT '07. International Symposium on, 2007, pp. 152–157, doi:10.1109/ISCIT.2007.4392003.
[13] M. Seok, S. Hanson, Y.-S. Lin, Z. Foo, D. Kim, Y. Lee, N. Liu, D. Sylvester, D. Blaauw, The phoenix processor: a 30pw platform for sensor applications, in: VLSI Circuits, 2008 IEEE Symposium on, 2008, pp. 188–189, doi:10.1109/VLSIC.2008.4586001.
[14] V. Ekanayake, C. Kelly IV, R. Manohar, An ultra low-power processor for sensor networks, in: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems, in: ASPLOS-XI, ACM, New York, NY, USA, 2004, pp. 27–36, doi:10.1145/1024393.1024397.
[15] D. Park, T.G. Kim, Built-in binary code inversion technique for on-chip flash memory sense amplifier with reduced read current consumption, Very Large Scale Integrat. (VLSI) Syst. IEEE Trans. 22 (5) (2014) 1187–1191, doi:10.1109/TVLSI.2013.2265894.
[16] R. Micheloni, L. Crippa, M. Sangalli, G. Campardo, The flash memory read path: building blocks and critical aspects, Proc. IEEE 91 (4) (2003) 537–553, doi:10.1109/JPROC.2003.811704.
[17] K. Leuenberger, R. Gassert, Low-power sensor module for long-term activity monitoring, in: Engineering in Medicine and Biology Society,EMBC, 2011 Annual International Conference of the IEEE, 2011, pp. 2237–2241, doi:10.1109/IEMBS.2011.6090424.
[18] J. Koshy, R. Pandey, Remote incremental linking for energy-efficient reprogramming of sensor networks, in: Wireless Sensor Networks, 2005. Proceedings of the Second European Workshop on, 2005, pp. 354–365, doi:10.1109/EWSN.2005.1462027.
[19] M.-L. Chiang, T.-L. Lu, Two-stage diff: an efficient dynamic software update mechanism for wireless sensor networks, in: Embedded and Ubiquitous Computing (EUC), 2011 IFIP 9th International Conference on, 2011, pp. 294–299, doi:10.1109/EUC.2011.74.

Please cite this article as: D. Park et al., Area efficient remote code execution platform with on-demand instruction manager for cloud-connected code executable IoT devices, Simulation Modelling Practice and Theory (2016), http://dx.doi.org/10.1016/j.simpat.2016.08.010