A CIL Virtual Machine for Wireless Sensor Network Applications

Yutaka Yanagisawa¹, Yasue Kishino¹, Takayuki Suyama², Tsutomu Terada³
Masahiko Tsukamoto², and Futoshi Naya³
¹NTT Communication Science laboratories, NTT Corporation
²Cognitive Mechanisms Laboratories, Advanced Telecommunications Research Institute International
³Graduate School of Engineering, Kobe University

Abstract—This paper describes CILIX, a compact and powerful implementation of a CIL virtual machine working on resource-poor wireless sensor nodes. CILIX can process CIL programs on a device that has such limited computational resources as an 8-bit/16-bit CPU, 32-KB program memory, and 4-KB RAM. It provides many useful functions for a sensor node, including an I/O manager with UDP, FAT 32, thread control, and dynamic program replacement. For developing software on sensor nodes using CILIX, developers can choose programming languages from C#, C++/CLI, Visual Basic, J++, F#, and the many other languages supported by the .NET Framework.

1. Introduction

A process virtual machine, which enables portability by abstracting each device and operating system, also presents a standard programming interface across a range of target platforms [1]. This mechanism reduces the cost of developing a program that works on various sensor devices with different platforms. Currently, Java Virtual Machine (JVM) and Virtual Execution System (VES) for Common Intermediate Language (CIL) are the two most popular process virtual machines used on personal computers. In this paper, we denote an implementation of VES as CIL-VM. Several JVMs have been implemented on resource-poor sensor devices. For example, SimpleRTJ [2] and Darjeeling [3] provide the execution environment for Java code on small sensor devices that have an 8-bit/16-bit CPU, 2- to 4-KB RAM, and 32- to 128-KB program memory. Due to these existing JVMs, we can develop software for small sensor nodes on CIL-VM, developers can chose programming languages from C#, C++/CLI, Visual Basic, J++, F#, and the many other languages supported by the .NET Framework.

In the design of CILIX architecture, we focus on the following three requirements:

1) Compatibility: CILIX must have high compatibility with the existing implementations of CLR and Mono virtual machines.
2) Functionality: CILIX must provide the necessary functions for wireless sensor devices.
3) Memory-Saving: For porting on small memory devices, CILIX’s program size should be much smaller than existing CIL-VMs. Moreover, we introduce a mechanism to reduce the size of the CIL program stored in the memory.

In general, strong compatibility and functionality increase the program size of the runtime system. Our main challenge is to develop techniques to reduce the required program memory without any deterioration of compatibility and functionality.

The fully compatible CIL-VM described in ECMA 335 [4] has many functions not used in CLR, which is included in the .NET Framework.¹ For example, sleeping functions increase the program size. In our first approach to reducing program memory, we omit them in our designed VM after we investigate the necessity of every function described

¹These functions may be supported in the future or perhaps Microsoft just added as many functions as possible.
in ECMA 355. In the second approach, we identify the
functions that require large program size, but that are used
only for limited purposes. It is impossible for small resource-
poor devices to provide all the functions of the full version
CIL-VM described in ECMA 355 [4] because the full
version of CIL-VM is designed for rich computer devices
that have a 32-bit CPU, large RAM, program memory, and
various I/O devices. To develop a CIL-VM with reason-
able compatibility, we checked all the functions in CIL-
VM presented by ECMA 355 and the number of program
codes generated by both the compiler csc.exe in the
.NET Framework and the gmcs command provided by the
Mono project. We carefully chose functions that are not
implemented on CILIX from the viewpoint that they are
very rarely used for small sensor devices. For example,
functions to execute unmanaged code are available for using
existing native libraries such as Win32API.dll, but small
sensor devices have no such libraries. Therefore, we do not
support any functions that execute unmanaged code on a
sensor device. The details and the reasons for our choices
of functions are described in Section 2.4. Section 3 presents
information to implement a CIL-VM that has substantial
compatibility to the existing CIL-VMs.

We also introduce a mechanism called a Metadata Pre-
Processor (MPP) to reduce the size of the CIL program
to be executed on a sensor derive. The large CIL program
also consumes memory space on a device, because the CIL
program code must be stored on the program memory. The
PE (.exe) file that includes the CIL program code has some
redundant and unused data in the runtime. Our implemented
MPP removes such unused data and compresses the redun-
dant data.

We implemented CILIX on ATmega128L (8-bit CPU,
4-KB RAM, 128-KB program memory), MSP430, (16-bit
CPU, 4-KB RAM, 32-KB program memory), and TWE-001
(32-bit CPU, 128-KB RAM, 128-KB program memory). To
check both the size of the program memory and the com-
pression ratio of the program code by MPP, we developed
several programs for encoding, data compression, numerical
treatment, sorting, and so on. Moreover, we compared the
processing time and the size of the used memory on our
implemented CILIX with existing virtual machines in CLR
and Mono. As a result of these experiments, we show that
CILIX can execute CIL program code for practical usage on
several small sensor devices that have limited computational
resources.

2. Requirements and Design

As mentioned in Section 1, we designed CILIX to meet
three requirements: having high compatibility with existing
CIL-VM; providing available functions to work on a wireless
sensor device; and reducing both the size of CILIX and
the CIL program stored on ROM or flash memory. In this
section, we show the CILIX architecture after explaining our
three requirements.

2.1 Compatibility

The virtual machine brings a standardization of envi-
ronments to develop software by abstracting the diverse
platforms of sensor devices. In other words, compatibility
with existing virtual machines is one of the most important
requirements in porting a virtual machine to a new platform.
Therefore, we designed CILIX as a highly compatible VM
that can execute a CIL program (.exe file) compiled by
existing compilers, such as csc.exe provided by Microsoft
and gmcs provided in the Mono project. Even though CILIX
has high compatibility with existing VMs, it is difficult to
implement a fully compatible VM on a small sensor device
that has only limited computational resources. To achieve
both high compatibility and porting onto a resource-poor
device, we carefully removed the unsuitable functions that
are too expensive to implement on a sensor device.

2.2 Functionality

The CIL-VM defined in ECMA 335 is designed as a
simple virtual 32-bit stack-based processing unit, similar to
JVM. The CIL-VM only has such essential functions as
number calculation, transferring data on the memory, and
controlling the executed program. In general, to support
practical functions, for example, I/O management, threading,
and file systems, developers implement these functions as
a class library. To archive both the reduction of the size
of CILIX and to provide useful functions for developing
program code on a sensor device, we implemented the
following three significant functions as an embedded class
library in CILIX:

1) Dynamic program relocator
2) Interfaces for typical I/O devices (sensors and wireless
communication devices)
3) Multi-threading controller

Fig. 1: CILIX Architecture
These functions are supported by most existing middleware for small sensor devices. We implemented them as a class library that has compatibility with the .NET Framework class library. For example, we implemented the embedded Thread class to support the multi-threading mechanism. Our implemented Thread class also has methods \texttt{run()}, \texttt{stop()}, \texttt{wait()}, and so on.

Each method provides the same function as the method implemented in the Thread class, which is included in the .NET Framework class library.

### 2.3 Memory-Saving

Reducing memory is the most significant technical issue to introduce virtual machines into limited-resource devices. Increasing the size of the logical memory, such as EEPROM, flash memory, and RAM, increases the physical size of the device and its price. In other words, we can use a small, low-price sensor device to reduce the program size.

We determined the minimum hardware requirements for the device, which has 4-KB RAM and 32-KB program memory (EEPROM and/or flash memory). As mentioned in Section 1, our required minimum hardware is smaller than most existing sensor devices. The price of the minimum 8-/16-bit device, which has 4-KB RAM and 32-KB program memory, is lower than $5$. Richer 8-/16-bit devices than our minimum requirements provide little price advantage for 32-bit devices. Therefore, we chose the above minimum hardware requirements.

### 2.4 CILIX Architecture

The CLI specifications are defined in ECMA-335 [4][5]. ECMA-335 has four partitions, I–IV, each of which has independent page numbers. In this paper, the following notation, “ECMA P-X Y P’”, means page $Y$ in partition $X$ of ECMA-335. For example, ECMA P-II 183 P means page 183 in partition II, and ECMA P-X S.Y means section $Y$ in partition $X$.

The CILIX runtime system has the following four runtime modules:

- Executor: loads and executes CIL program data from EEPROM or flash memory.
- Process Manager: controls the start-up and the stopping of the virtual module and also has a function to dynamically replace the program data on the memory.
- Platform-independent I/O Manager: provides a method to access I/O devices and only includes program code that is independent from device architecture, such as processing string data, conversion of data types, and so on.
- Platform-dependent I/O Manager: provides a method to access physical I/O devices and includes device-dependent program code.

Figure 1 shows the runtime system architecture of our designed CILIX. CILIX is composed by runtime modules and a Metadata Pre-Processor (MPP), which compresses the size of the CIL program data (exe\_file) MPP is an independent module from the runtime system that can reduce the total size of the CIL program data by removing unused program code from a exe*

### 3. Implementation

This section describes the implementation of CILIX. We show the information for the implementation of CIL-VM, which has substantial compatibility with existing runtime systems, before we explain the non-CLI modules to provide convenient functions for wireless sensor devices. For the I/O control module, we only describe the essential ideas to implement it.

#### 3.1 Substantial Subset of CLI

As mentioned in the previous section, ECMA defines the CLI specifications, but existing compilers \texttt{csc.exe} and \texttt{gmcs} do not generate all the CIL operations, the metadata tables, and the signatures described in ECMA. To reduce the program size of the runtime system, we implemented CILIX as a substantially compatible CIL virtual machine without functions for supporting such unused data. We extracted the minimum indispensable information from ECMA to implement CILIX, which can execute any CIL program code generated by \texttt{csc.exe} and \texttt{gmcs} without unsupported functions, as explained in the previous section.

To obtain information, we investigated a number of .exe files generated by existing compilers for C#, C++/CLI, and Visual Basic.

In the rest of this subsection, we describe the information to implement CILIX. This information is available for developers who want to implement another CIL virtual machine.

#### 3.2 Process Management Module

This module, which has several important functions for controlling a process on a sensor device like a small embedded operating system, provides these functions: process initialization, multi-thread control, dynamic program relocation, and restoring from an exception. For the initializing process, the module allocates memory for the program and loads the data used in the program onto the runtime memory from the program memory. Next we describe the other functions.

#### 3.2.1 Dynamic Program Relocator

CILIX provides a function to change a CIL program to execute by relocating the program code on the program...
memory. The Program Relocator can read program code from a MicroSD card or a remote server by wireless communication to put the read data into the program memory (flash memory).

The relocation process is very simple. When we use the wireless communication method for relocation, we must send a special packet to inform the next packet including the new program code. If the I/O manager of the wireless device finds the spatial packet, the manager informs the Process Manager who stops the program’s execution before the Relocator starts to work. The program code is transferred as a set of UDP packets next to the special packet. We describe UDP-based communication in the next subsection. After the Relocator retrieves the program code from the buffer in the I/O manager, the Relocator puts the program code into the program memory. Finally, the Process Manager initializes and starts the new program.

For MicroSD cards, we put them into a MicroSD card slot. When the I/O Manager of the SD Card (SPI) finds a new MicroSD card, the Manager checks a file named /program.hex based on the FAT32 format. If the Manager finds the program, it informs the Process Manager who performs the same processes as for using a wireless device.

3.2.2 Thread Controller

A typical process on a sensor node is a combination of a program to read a value from a sensor in an interval and a program to send a set of read values to the server. In this case, we want to concurrently execute two programs on a sensor device. CILIX supports a multi-thread control mechanism, which is available for such uses. The Thread Controller of CILIX provides simple concurrent processing in a sensor device. Each thread has an independent heap area and a buffer to back up the data in the managed area. CILIX has a memory space for the managed area of the current thread. This memory space stores the global variables used in the runtime.

To exchange the current thread with a suspended thread, CILIX moves all the data in the managed area into the current thread’s buffer after CILIX stops to execute the current thread. Next, it moves all the data in the buffer in the suspended thread and switches to the heap area. Finally, it executes a new current thread with the heap area and manages all the thread’s data.

The context switching interval can be given by the developer. As a default setting parameter, CILIX switches the thread every 160 opcodes. In our implementation, CILIX can manage any number of threads as long as the device has memory space.


3.2.3 Restoring from Exception

When an exception occurs and no try/catch block catches it, CILIX must process the restoring from the exception. If the device has a process management system such as an operating system, the system recovers the uncaught exception. On small devices without such a recovery system, CILIX recovers the exception. CILIX provides two alternative methods; the runtime system restarts the program in the first method, and the runtime system halts the process and waits for the program code sent from the server.

3.3 I/O Control Module

A small sensor device has various types of I/O devices, for example, thermometers, acceleration sensors, UART, SD cards, and radio frequency devices for wireless communication, LEDs, and LCDs. In general, developers must write specific program code for each individual device. To abstract I/O devices, we introduce a UDP-based interface in the design of an I/O control module. CILIX allows us to access each I/O device with UDP packets.

Our design offers the following three benefits:
1) Selecting a device with a port number: to change the device to the access mode, a developer only changes the port number related to the device.
2) Emulation of a device as a UDP program on a PC: we can build an I/O device as a program with a UDP port on a PC for debugging.
3) Concentration of device-dependent code in send and recv: any CIL program can only access an I/O device through the send and recv methods. In other words, all the program code, which depends on each I/O device, is gathered into these methods.

CILIX supports reading the /program.hex file from FAT32 sectors on SPI devices. We only suppose the usage of SPI devices to transfer a CIL program file from a PC to a small device. CILIX does not currently support reading other files or writing data onto an SPI device because we must add too much code to support those functions. For reading a program file, CILIX does not use UDP packets to improve the time to load program code in the memory. The runtime system automatically checks the SPI device to determine whether it has a program file to load during the runtime system’s initial process.

4. ICT Application for Agriculture

In this section, we show an application systems using our designed virtual machine.

To improve flower yields, we built a remote environmental monitoring system using our 17 sensor nodes in two greenhouses from July to December in 2012. This application system consists of the following three sub-systems (Fig. 2):

\[\text{Note that we only give this value through our experiments to execute a number of practical programs on sensor devices.}\]
Wireless sensor network: gathers sensor data in greenhouses.

Translator: receives sensor data from the sensor network and sends them to the server by a 3G/4G network.

Sensor data server: stores and retrieves sensor data.
The key system is the wireless sensor network that consists of TWE-001-based wireless sensor nodes. Each sensor node, which has the CIL program of our implemented virtual machine, obtains data from each sensor by the CIL program. The size of the compiled CIL program (exe file) is 16 KB, but our MPP compresses it to 4 KB. At the beginning of this experience, the measurement period was one minute on all the sensor nodes, but later we often changed the period (1-10 minutes) to optimize the balance between energy consumption and the value of the data on each sensor. Several times we also rewrote the program to change the routing protocol and the data compression algorithm to fit the sensor network. In the modification process, we used a function for a dynamic program relocation with wireless communication to reduce the cost of rewriting the program on each sensor node deployed in a large greenhouse. Our CIL virtual machine reduces the cost of program development because many existing Windows programmers have mastered how to develop applications for sensors network for a short period.

5. Related Work

In this section, we describe such previous proposed middleware as operating systems and virtual machines and explain the position of our implemented CILIX in them. Table 1 and 2 show the existing middleware systems and virtual machines.

The middleware for small sensor nodes can be classified into two types of software. The first is operating systems, such as TinyOS[6], Smart-Its[7], PAVENET[8], Ubiquitous Chip[9], BTnodes[10], and Nano-RK[11]. The second is virtual machines, such as Darjeeling[3], SimpleRTJ[2], Sun Spot[12], Maté[13], ASVM[14], and VM*[15]. Several systems present both functions; for example, the .NET Micro Framework (NMF) [16] includes both a virtual machine and an operating system. We focus on the cost of developing programs on a sensor node for each proposed system.

TinyOS is one of the most common operating systems for developing a program on a wireless sensor device. It provides necC and TinyScript as the programming languages, even though they are not so popular for developing programs for resource-rich computer devices. The programs developed in the Ubiquitous Chip are described as a small set of simple ECA rules. However, the developers must be familiar with the techniques to write a program with event-driven rule-based languages. Smart-Its presents libraries to develop a program on a PIC device with C language. Several operating systems only allow C language for program development. For instance, BTnode provides an environment to develop a program on a sensor device with Bluetooth, PAVENET supports a function for real-time hardware processing on a small device, and Nano-RK can be used for developing a program to process data on a sensor node in real time. To develop a program on a sensor device, we must use a specific programming language for each operating system.

Virtual machines can be classified into two types of systems: a subset system of Java Virtual Machine (JVM) and one with an original language (not Java). Maté[13] has a non-Java virtual machine, which can execute a program on TinyOS. The internal language of this virtual machine has a set of 24 convenient operation codes for wireless sensor devices. For example, it has a function to retrieve data from a sensor device with only one-byte operation code, which enables us to reduce the size of the program code. ASVM, another non-Java virtual machine for TinyOS, allows us to customize the set of operation codes. The original virtual machine achieves both powerful functions and a method to reduce the size of the program code; however, we do not choose a programming language to develop software on a sensor device.

On the other hand, there are several subsets of JVM, such as SimpleRTJ, VM*, Sun Spot, and Darjeeling. To execute a Java program on a limited-resource device, SimpleRTJ provides a mechanism to pack and compress both the runtime system and the Java program code into a small program module. Darjeeling presents a program converter from original Java byte code to compressed byte code. Since Darjeeling also provides a virtual machine to execute compressed Java byte code, it enables us to execute a Java program on a resource-poor device. VM*, which is a virtual machine implemented on a MOTE device, reduces the program code by the following two mechanisms. The first removes unused code from the .class file, and the second selects a minimum set of modules to execute Java byte code. Sun Spot presents a virtual machine to execute Java byte code without any conversion; however, it requires a rich device to execute a program. Every virtual machine has several implementations for 8-bit, 16-bit, and/or 32-bit CPUs. Each virtual machine allows us to develop a program only with Java programming language.

As mentioned, most existing middlewares limit the environment to develop a program on a sensor device. On the other hand, CIL-VM can execute programs written in several programming languages, including C#, C++/CLI, J++, Visual Basic, and F#. The .NET Micro Framework (NMF) provides a runtime system for a 32-bit CPU, but it cannot support 16-/8-bit CPUs. Because small sensor devices usually have an 8-/16-bit CPU, a developer cannot use it to develop programs on them. Since our CILIX can execute a CIL program on such a limited-resource device with an 8-/16-bit CPU, we can develop a program with many different programming languages.

6. Conclusions

This paper presented the design and implementation of CILIX, which can work on an 8-/16-bit CPU, 4-KB RAM, and 32-KB program memory with reasonable compatibility.
to existing CIL-VMs. We implemented CILIX on three devices, ATmega128L, MSP430, and TWE-001, and experimentally evaluated the performance of our implemented CILIX with them. We showed that CILIX can execute CIL program code with practical processing times on each device with limited resources. We hope our research leads to the development of another CIL-VM.

References