Dynamic Linking and Loading in Networked Embedded Systems

Wei Dong[1], Chun Chen[1], Xue Liu[2], Jiajun Bu[1], and Yunhao Liu[3]
[1]Zhejiang Key Laboratory of Service Robot, College of Computer Science, Zhejiang University
[2]School of Computer Science, McGill University
[3]Department of Computer Science, HKUST
{dongw, chenc, bjj}@zju.edu.cn, xueliu@cs.mcgill.ca, liu@cse.ust.hk

Abstract

We present a holistic dynamic linking and loading mechanism in networked embedded systems. Our design and implementation are guided by four requirements, which are to provide (i) minimal code size (ii) efficient execution and loading speed (iii) portable design (iv) isolated kernel/application development. First, we develop a tool to minimize the standard ELF format via many techniques in order to reduce the code dissemination cost. Second, we employ the techniques of pre-relocating and pre-linking (to kernel functions) to reduce the run-time linking overhead, thus improving the loading speed. Third, based on relocatable ELF and the modular design of the dynamic linker and loader, our approach can be easily ported to different platforms. Fourth, by maintaining a kernel jump table, we provide a clean isolation between kernel and application development. We have implemented the dynamic linking and loading mechanism on SenSpire OS, a micro sensor node operating system. The evaluation results show that our design and implementation meet our design goals: the code size of our SELF format is only 15%–30% of that of standard ELF, 38%–83% of that of CELF, a compact ELF format for the Contiki operating system; the loading speed improvement varies from 40%–50% compared to the standard mechanism; our design is portable to both MicaZ and TelosB motes, and we allow updating both application modules and kernel services in isolation without prior knowledge about the whole system information.

1. Introduction

Wireless sensor networks (WSNs) have been proposed for a wide range of applications such as military surveillance, habitat monitoring, and infrastructure protection, etc. WSN applications need to be changed after deployment for a variety of reasons, such as correcting software bugs, modifying tasks of individual nodes, and patching security holes. Many large-scale WSNs, however, are deployed in environments where physically collecting previously deployed nodes is either very difficult or infeasible. Enabling sensor nodes to be reprogrammable over the air is an important technique to address such challenges [23].

In the reprogramming process, the loading mechanism of a sensor node is responsible to load a new code image (disseminated over the air) onto the program flash, enabling sensor nodes to execute the new code. The loading mechanism impacts the overall reprogramming efficiency: a simple bootloader, e.g., `TOSBoot` [9], requires the replacement of an entire application image, which is not energy-efficient to disseminate; a virtual machine (VM), e.g., Maté [14] and JVM [5], can naturally support loading and executing a very compact code, but is not energy-efficient to execute in the long-term [5].

Contrary to the abovementioned mechanisms, dynamic linking and loading is an efficient way to enable sensor nodes to be reprogrammable: on one hand, it allows disseminating the loadable module, which is much smaller than the entire application image required by the bootloader mechanism; on the other hand, it allows executing the native code, which is much more efficient for execution than the VM instructions required by the virtual machine mechanism.

To this end, we explore the design of a dynamic linking and loading mechanism for micro sensor nodes. Our design and implementation are guided by four requirements:

Minimal Code Size. The radio subsystem is one of the major cost drivers in terms of energy consumption on current mote platforms. Therefore, communication should be limited to a minimum during code dissemination in order not to reduce the lifetime of the network too much. In order to reduce the communication cost, the disseminated code size should be minimized.

Efficient Execution and Loading. In many application scenarios, e.g., correction of software bugs, the disseminated code is expected to be executed for a long time [5]. Therefore, the code execution speed should be very efficient to reduce the duty cycle in order to save energy. In other application scenarios, e.g., software development, the code update frequency is very often [5]. It is hence expected that the loading process should be efficient in terms of loading speed, RAM consumption, and Flash I/Os.

Portable Design. Because of the diversity of sensor node hardware, the dynamic linking and loading mechanism should be designed to be easily portable between different platforms (e.g., MicaZ and TelosB).

Isolated Kernel/Application Development. Modern sensor network systems are very complex, which require different classes of developers. For example, kernel developers are responsible to provide system services while application developers are responsible to construct application logic. It is desired that kernel developers and application developers should be able to update their own code in isolation without prior knowledge about the whole system information.

To the best of our knowledge, no existing dynamic linking and loading mechanisms satisfy all of these requirements. FlexCup [15] allows dynamic loading of TinyOS components, but the loading overhead is large because it makes extensive use of Flash and requires a hardware reboot for executing the new code image. Contiki [5, 6] uses the CELF format for dynamic linking and loading. As the design philosophy of Contiki is somewhat towards using standard mechanisms and file formats, the code size is not yet minimized. SOS [8] uses the Mini ELF (MELF) format for dynamically loading and unloading modules. As described in [5, 8, 19], the MELF format uses position independent code (PIC) and due to architectural limitations on common embedded platforms, the relative jumps can be only within a certain offset (such as 4K for the AVR platform). RETOS [2] and LiteOS [1] also support loadable modules by using specific file formats. However, no specific techniques for improving the loading speed are employed.

This paper presents a holistic dynamic linking and loading mechanism in networked embedded systems. First, we develop a tool (i.e., elftoself) to minimize the standard ELF format via many techniques, to reduce the code dissemination cost. Second, we employ the techniques of pre-relocating and pre-linking (to kernel functions) to reduce the run-time linking overhead, thus improving the loading speed. Third, based on relocatable ELF and the modular design of the dynamic linker and loader, our approach can be easily ported to different platforms. Fourth, by maintaining a kernel jump table, we provide a clean isolation between kernel and application development.

We have implemented the dynamic linking and loading mechanism on *SenSpire OS*, a recent micro sensor node operating system. It is worth noting that, our mechanism can also be ported to other sensor node operating systems. We chose *SenSpire OS* mostly because of its kernel code accessibility and multi-platform support. The evaluation results show that our design and implementation meet our design goals: the code size of our SELF format is only 15%–30% of that of standard ELF, 38%–83% of that of CELF, a compact ELF format for the Contiki operating system; the loading speed improvement varies from 40%–50% compared to the standard mechanism; our design is portable to both MicaZ and TelosB motes, and we allow updating both application modules and kernel services in isolation without prior knowledge about the whole system information.

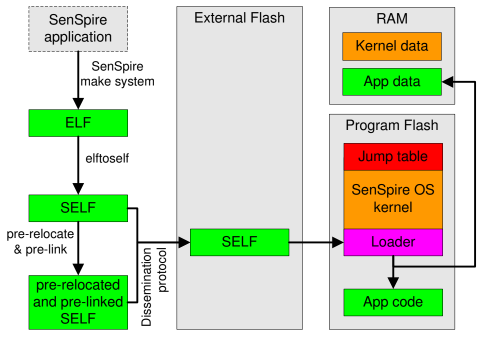

Figure 1. Architectural overview of the dynamic linking and loading mechanism based on *SenSpire OS*

The rest of this paper is structured as follows. Section 2 gives an overview of our dynamic linking and loading mechanism. Section 3 presents our design and implementation. Section 4 shows the evaluation results. Section 5 describes related work. Section 6 concludes this paper.

2. Overview

Figure 1 depicts the architectural overview of the dynamic linking and loading mechanism based on *SenSpire OS*. Programmers write *SenSpire* applications which will be compiled (and linked) by our *SenSpire* make system. Our *SenSpire* make system generates a standard ELF file using relocatable code, which is more portable than position independent code (PIC). As the standard ELF file is initially designed for traditional PCs, it contains extra overheads and is not energy-efficient to disseminate. Thus, we have developed a tool, i.e., elftoself, to transform the standard ELF into the SELF (i.e., "Slim" ELF) file, in order to *minimize the code size*.

At this time, it still contains internal symbols and external symbols to be resolved at run-time. For the clarity of presentation, we call the process of resolving internal symbols as relocating and the process of resolving external symbols as linking [5]. We try to avoid run-time relocating and linking by employing the techniques of pre-relocating and pre-linking (to kernel functions) at compile-time, thus *improving the loading efficiency*. Traditional pre-relocating and pre-linking techniques improve loading efficiency at the cost of limited flexibility, as the layout of program flash (e.g., addresses of variables and functions) must be all the same for all nodes. We address this problem by two mechanisms. First, we always remain necessary relocation information for code and data references to cope with the situation when the actual allocated base addresses do not match the "pre-allocated" based addresses. Second, we keep a kernel jump table to remain the flexibility of kernel code layout and application code layout. The existence of the kernel jump table allows *updating kernel/application code in isolation*. After this step, we obtain a pre-relocated and pre-linked SELF file.

The SELF file (either pre-relocated and pre-linked or not) is then transmitted to all sensor nodes via a code dissemination protocol, and is saved onto the external flash. Next, the dynamic loader starts loading the file: it resolves symbols if necessary; it writes the code section onto the program flash and the data section onto the data RAM; finally, the loader executes the initialization routine provided by the application module. It is worth noting that based on the modular design, our dynamic loader is *easily portable* to different sensor node platforms.

3. Design and Implementation

This section describes our design and implementation to meet the requirements listed in Section 1, including techniques to minimize the code size (Section 3.1), techniques to improve loading efficiency (Section 3.2), portability considerations (Section 3.3), and the kernel/application boundary for isolated development (Section 3.4).

3.1. Code Size

As we have already mentioned, it is very important to minimize the code size in order to reduce the code dissemination cost. To achieve this goal, we have developed a tool, i.e., elftoself, to transform standard ELF to SELF. The basic techniques we employed are described as follows.

Redefining Basic Data Types. The standard ELF format is originally designed to work on 32-bit and 64-bit architectures. This causes all ELF data structures to be defined with 32-bit data types. For 8-bit or 16-bit targets, such as current sensor nodes, the high 16 bits of these fields are unused. For this purpose, we have redefined the basic data types in standard ELF to the minimized data types in SELF (as shown in Table 1).

Tailoring the ELF Content. From the viewpoint of linking, the ELF content includes (i) the ELF header (ii) section headers (iii) various sections. We tailor the ELF content

Table 1. Data types redefinitions

Name	Def. in ELF/bits	Def. in SELF/bits
`elf_addr`	`unsigned int`/32	`uint16_t`/16
`elf_half`	`unsigned short`/16	`uint8_t`/8
`elf_off`	`unsigned int`/32	`uint16_t`/16
`elf_sword`	`int`/32	`int16_t`/16
`elf_word`	`unsigned int`/32	`uint16_t`/16

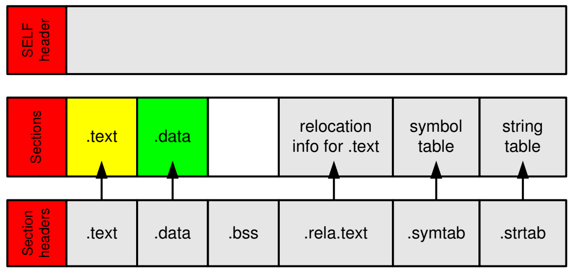

Figure 2. The SELF format

by two methods. First, the definitions of the ELF header, the section header, the symbol, and the relocation entry are tailored as many fields in the standard ELF will not be used in our limited environment. Second, we exclude unrelated sections by including only the text section (for the code), the data section (for the initialized data), the relocation table, the symbol table, and the string table. Figure 2 shows the graphical layout of the SELF format, which includes: (i) the SELF header, which is used to provide basic file information, and to locate section headers. (ii) section headers, which are used to locate corresponding sections. (iii) sections, which contain the text section (for code), the data section (for initialized data), the relocation table (for relocation), the symbol table (for internal and external symbols), and the string table (string representations of symbols). Note that, the bss section (for uninitialized data) are actually not contained in the file, because all variables in this section are known to be zero.

Tailoring the Relocation Table. The relocation table contains relocation entries for symbol relocation. In standard ELF, each relocation entry contains an index to the symbol and a pointer to the unresolved reference. The GCC compiler generates a relocation entry for each unresolved reference. To reduce the number of relocation entries, we employ the chained references technique [13]. The basic idea of this technique is trying to merge the relocation entries for the same unresolved symbol, because references to the same symbol in different locations of the program must point to the same address. To achieve this goal, we use references to the same unresolved symbol in the program (which contain useless values before relocation) to create a linked list. Therefore, in SELF, each entry in the relocation table requires only two values: an index to the symbol and

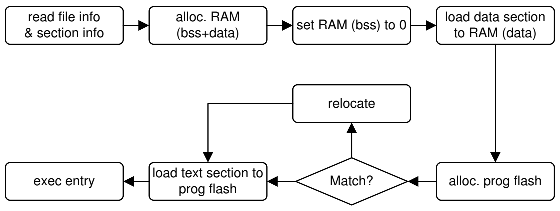
read file info & section info
alloc. RAM (bss+data)
set RAM (bss) to 0
load data section to RAM (data)
relocate
exec entry
load text section to prog flash
Match?
alloc. prog flash

Figure 3. Dynamic linking and loading

a pointer to the first reference (i.e., the header of the linked list) of that symbol. Hence, the relocation table grows with the number of unique symbols, instead of the number of references.

Tailoring the Symbol Table (and the String Table). In standard ELF, the symbol table and string table are separately stored. A symbol entry stores the pointer to the corresponding string representation. The reason is that this scheme saves storage when the string sizes are arbitrary. A symbol entry also contains the relative loading address used for address adjustment during relocating or linking. First, we remove the kernel symbols by the pre-linking (to kernel functions) approach (discussed in the next section). Second, to exclude useless symbols produced by the GCC utility, we tailor the symbol table by including symbols that need relocation or linking. The string table is tailored and restructured in accordance with the symbol table.

3.2. Loading Efficiency

After the code is disseminated and saved onto the external flash, our dynamic loader starts loading the code image for execution. As the dynamic linking and loading process is done at resource-constrained sensor nodes, it is important to improve loading efficiency to save energy. As a lot of work is spent on run-time relocating and linking, we try to reduce the relocating and linking overhead. Before we dig into the technical details, we first give an overview of the dynamic linking and loading process.

Figure 3 shows the dynamic linking and loading process. First, we read the SELF file header and section headers for general information. Second, we allocate data RAM for storing the bss and data sections according the specified sizes. We set the RAM for bss section to zero, and initialize the RAM for data section. Third, we allocate program flash for storing the program code. We do relocating if the actual allocated base address and the "pre-allocated" based address do not match. Finally, we load the relocated code onto program flash and execute the entry function defined in the code module.

In order to improve loading efficiency, we use pre-relocating and pre-linking (to kernel functions) to reduce the run-time relocating and linking overhead respectively.

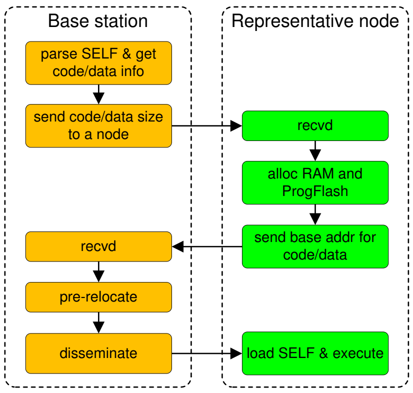

Figure 4. The pre-relocating process

First, we use the pre-relocating technique to reduce the run-time relocating overhead. As illustrated in Fig. 4, the base station first requests to a representative node for pre-allocating RAM space and program space. After obtaining the base addresses, we do pre-relocation at the base station. Finally, we disseminate the pre-relocated SELF to the network. It is worth noting that our approach is different from the simple pre-link approach as described in [5], where the pre-link process is totally done by the GCC utility and the relocation table is thus removed away after this process. We implement this functionality in our tool, i.e., elftoself, in which we still retain the necessary relocation information. The reason is that if the pre-allocated address does not match the actual loading address on certain nodes, the existence of the relocation information still allows the code module to be loaded onto the actual loading address. We do allow further removing the relocation entries for code references when the layouts of program flash are the same on all nodes (e.g., the compiling environments and the program logics are the same for all nodes).

Second, we use pre-linking (to kernel functions) to reduce the run-time linking overhead. To remain the flexibility of kernel code layout and application code layout, we maintain a kernel jump table in the program flash. A kernel function address called by an application is bound (i.e., pre-linked) to a pre-determined address of a table slot, which indirects it to the actual address of the kernel function. It is worth noting that this technique also reduces the code size as the kernel symbols are not stored in the SELF file.

3.3. Portability

The portability considerations in our design and implementation reflect in two main aspects.

First, we use relocatable code, instead of position independent code (i.e., PIC, as was used in SOS [8]). The major reason is that PIC is architecture dependent, and due to architectural limitations on common embedded platforms, the relative jumps can be only within a certain offset (such as 4K for the AVR platform). In addition, PIC is currently not supported by the GCC utility for MSP430 platforms. Finally, it is also worth noting that relocatable code eliminates indirection cost incurred by PIC, which also improves execution efficiency.

Second, after processing the relocatable ELF for various optimizations (see Section 3.1 and Section 3.2), the resulting SELF format is still the same across different platforms. Correspondingly, our dynamic loader is designed to be easily portable to different platforms. With the same design approach in CELF loader [6], our loader is split into one generic part and one architecture-specific part: the generic part parses the SELF file, finds relevant sections, looks up symbols, and performs the generic relocation logic; the architecture-specific part allocates data RAM and program flash, writes the code to the program flash, and understanding the relocation types in order to modify machine code instructions that need to be relocated.

3.4. Kernel/Application Isolation

As mentioned in Section 3.2, in order to improve loading efficiency and to reduce the code size, we always pre-link kernel functions. If not handled carefully, when the *SenSpire OS* kernel is modified, it will no longer support old applications that are pre-linked to old kernel functions. In order to address this issue and allow a clean isolation between kernel/application development, we have used a kernel jump table to implement the mechanism of lightweight system calls (as in LiteOS [1]). Each entry in the jump table represents a callgate (i.e., a special type of function pointers) to a predefined kernel function. These callgates are the only access points through which user applications access system resources. Therefore, they implement a strict separation between the kernel and applications.

The benefits of this scheme are twofold. First, as long as the system calls remain supported by newer versions of *SenSpire OS*, user binaries do not need to be recompiled. Second, more importantly, it allows in-situ reprogramming a specific kernel function without interfering existing applications by modifying a specified callgate to point to a new kernel function.

The kernel jump table takes up space of program flash. For example, on MicaZ, each system call gate takes 2 bytes, with 512 bytes of program space allocated for at most 256 system calls. Compared to the total program flash on MicaZ, i.e., 128K, this overhead is acceptable. Moreover, as this kernel jump table is programmed along with the *SenSpire OS* kernel before deployment, it would not incur dissemination cost. Indeed, the jump table adds execution overhead. For example, on MicaZ, compared to directly invoking kernel functions, each system call adds approximately 10 clock cycles (i.e., 1.36μs), a sufficiently low overhead to be supported on current mote platforms.

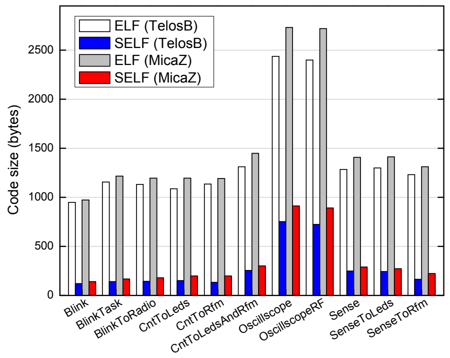

Figure 5. Code size comparisons (SELF vs. ELF)

4. Evaluations

In this section, we evaluate how well our design and implementation meet the requirements listed in Section 1.

- In Section 4.1, we examine the code size. We write a suite of benchmarks based on *SenSpire OS*'s linking and loading mechanism, and compare the code sizes of SELF with that of standard ELF. We also conduct a comparative study among Contiki OS, SOS, and our approach using three common benchmarks.

- In Section 4.2, we examine the execution and loading efficiency. Previous work already shows that the native code executes much faster than VM code, and relocate code executes slightly faster than position independent code (PIC). We quantify how the pre-relocating and pre-linking techniques improve the loading speed using two typical benchmarks.

- In Section 4.3, we examine whether our approach is easily portable to different mote platforms.

- In Section 4.4, we demonstrate how applications and the *SenSpire OS* kernel can be developed and updated in isolation.

4.1. Code Size

We first compare the code sizes of SELF to that of standard ELF, based on a suite of benchmarks implemented on *SenSpire OS*. Figure 5 depicts the results. We can see that on both MicaZ and TelosB, our SELF reduces the code sizes considerably compared to ELF: the code sizes of our SELF format are 15%–30% of that of standard ELF.

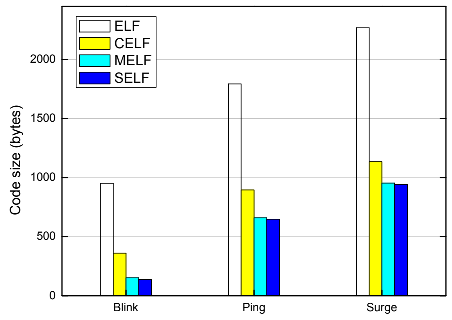

Figure 6. Code size comparisons (SELF vs. CELF, MELF, ELF)

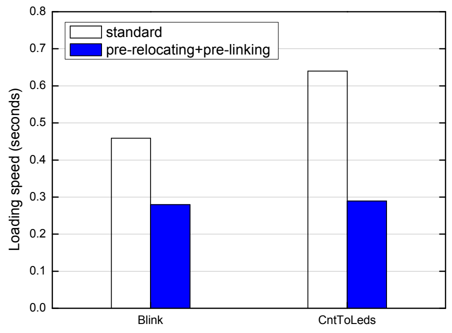

Figure 7. Loading speed comparisons (pre-relocating+pre-linking vs. standard mechanism used by SOS, RETOS, LiteOS)

We then conduct a comparative study among Contiki OS, SOS, and our approach. We select three benchmarks in the SOS distribution, i.e., Blink, Ping, and Surge. SOS's MELF file is generated using the tool in the SOS distribution, i.e., elftomini. We implement these benchmarks in *SenSpire OS*. The SELF file is generated using our tool, i.e., elftoself. Contiki's CELF file is also generated using the method described in [5]. Figure 6 depicts the results. We can see that the code sizes of SELF are 38%–83% of that of CELF, and both SELF and MELF generate the smallest code sizes.

4.2. Execution and Loading Efficiency

In order to see the execution efficiency of our approach, we first notice that the native code is inherently much more efficient than the VM code. For example, as reported in [5], for an object tracking benchmark, the native code executes more than three times faster than the JVM counterpart. Within the native code, relocatable code executes faster than PIC (which is employed in SOS [8]), as it eliminates indirection costs. For example, as reported in [19], for the Surge benchmark, the relocatable code executes approximately 12.8% faster than PIC.

In order to see the loading efficiency of our approach, we write two benchmarks, i.e., Blink and CntToLeds. We compare the loading speed of our approach (i.e., using optimizations described in Section 3.2) with the standard mechanism employed in SOS, RETOS, and LiteOS [1, 2, 8]. We run both benchmarks on Avrora [20], a cycle accurate sensor node simulator. Figure 7 depicts the results. We can see that the loading speed improvements vary from 40%–50% compared to the standard mechanism.

4.3. Portability

To evaluate the portability of our design, we have ported our implementation to two different mote platforms: MicaZ with Atmega128L microcontroller and TelosB with MSP430 microcontroller. First, we notice that based on relocatable ELF, our approach can support arbitrary programs on both platforms. Second, as we already mentioned in Section 3.3, the modular design of our approach facilitates porting to different platforms. Table 2 shows the lines of code (LoC) needed to implement each module. The main difference between the MSP430-specific module and the AVR-specific module is due to the different addressing modes used by the machine code of the two microcontroller. While the MSP430 has only one addressing mode, the AVR has 19 different addressing modes [5]. Each addressing mode must be handled differently by the relocation function, which leads to a larger amount of code for the AVR-specific module.

Table 2. LoC of different modules in our implementation

Module	Sub-module	LoC
elftoself	generic	496
	AVR-specific	129
	MSP-specific	52
dynamic loader	generic	302
	AVR-specific	149
	MSP-specific	58

4.4. Kernel/Application Isolation

We demonstrate how applications and the OS kernel can be separately developed and updated by two illustrative examples.

First, there is almost no visible difference to develop an application module (based on the dynamic linking and loading mechanism) and to develop a single application/kernel image. Figure 8 shows the Blink application module writ-

```
import System, Led, Task;

[@module]
static class Blink
{
  void start()
  {
    Task.startPeriodic(blink, DELAY);
  }
  void blink(uint8_t msg, void* data)
  {
    Led.redToggle();
  }
}
```

Figure 8. The Blink application module

```
import System;

static class Task2
{
  void startPeriodic(callback_t func,
                     uint8_t    period)
  {
    // ...
  }
  void start()
  {
    System.register(SYS_STARTPERIODIC,
                    startPeriodic);
  }
}
```

Figure 9. Code module to update the kernel function `Task.startPeriodic`

ten in the CSpire language which is used to develop *SenSpire* applications [4]. Note that, we only need to specify the `@module` attribute to the `Blink` class. According to this attribute, our *SenSpire* make system will handle all specific details to compile and link the Blink module.

Second, the *SenSpire OS* kernel can be incrementally upgraded on-site without interfering existing applications. Suppose that the kernel function `Task.startPeriodic` has to be changed for safety reasons. To accomplish this goal, we only need to disseminate the code module shown in Fig. 9, instead of the entire kernel. In Fig. 9, we re-implement this kernel function, using any other standard kernel functions if needed. In the initialization routine, we invoke a system call, i.e., `System.register`, to update the kernel jump table slot. When the code module is disseminated to a sensor node, the initialization routine will be executed by the dynamic loader, modifying the corresponding kernel jump table slot to point to the new kernel function. As long as the this function's semantic and signature keep consistent, applications residing on the sensor node can seamlessly invoke this new function for increased safety.

5. Related Work

Reprogramming wireless sensor networks have been an active research area in recent years [9, 12, 18, 23]. To enable sensor nodes to be reprogrammable, code must first be disseminated to all via a reliable code dissemination protocol. Then the loading mechanism on each sensor node is responsible to load and execute the new code image.

There is a lot of work devoted to code dissemination protocols in WSNs [9, 11, 12, 17, 18]. The loading mechanism also attracts research attentions in recent years because it can largely impact the code size to be disseminated. A small code size yields small dissemination cost, which improves the energy efficiency of WSNs. As we have already mentioned, the simple bootloader approach is not desired as it requires the replacement of entire application image. On the other hand, the virtual machine approach [14] allows disseminating a very compact code, but it is not efficient to execute in the long-term. The dynamic linking and loading approach can address the dissemination deficiency and execution deficiency of the above approaches. Thus, it is adopted in almost all recent sensornet OSes [3]. In the following, we will discuss the loading mechanisms on several notable sensornet OSes.

FlexCup [15] supports update of binary components in TinyOS. Compared to our approach, FlexCup is less portable as it is designed specifically for TinyOS. What's more, its loading overhead is large as it makes extensive use of Flash and requires a hardware reboot each time a program is to be installed.

SOS [8] natively supports dynamically-loadable modules. SOS uses position independent code (PIC) to avoid code relocation. PIC is architecture dependent, and due to architectural limitation on common embedded platforms, the relative jumps can be only within a certain offset (such as 4K on the AVR platform). Moreover, PIC is currently not supported by the GCC utility for MSP430.

Contiki [6] also supports loadable modules. Contiki supports both ELF and CELF (a compact ELF) for dynamic linking and loading. The CELF only uses "data type redefinition" to reduce the code size. As such, the CELF size is much larger than that of SELF. Besides, it does not consider issues relating to kernel/application developments. SELF also retain the portability of CELF by using relocatable code and the modular design approach. FiGaRo [16] enhances Contiki's loadable module support by providing a module reconfiguration system, which handles inter-module dependencies. This work is orthogonal to our work, and can be incorporated into our current implementation.

RETOS [2] and LiteOS [1] are two recent sensornet OSes that support dynamic linking and loading by using specific file formats. However, no specific techniques for improving the loading speed are employed.

Finally, it is worth mentioning that there is a tradeoff between the code size and loading efficiency. For example, data encoding and decoding [21] can be used to further reduce the native code size, but incurs a large decoding overhead during the loading process. Also, there is a tradeoff between the code size and flexibility. For example, with a prior knowledge about the program of the previous version, it is able to disseminate just the difference (i.e., delta) [7, 10, 22], but it limits the flexibility, e.g., it is only viable when all the sensor nodes have the same program layout, and is thus not easily applicable to heterogeneous sensor networks.

6. Conclusions

We present a holistic dynamic linking and loading mechanism in networked embedded systems, to meet the requirements of minimal code size, efficient execution and loading, portable design, and isolated kernel/application development.

First, we develop a tool to minimize the standard ELF format via many techniques in order to reduce the code dissemination cost. Second, we employ the techniques of pre-relocating and pre-linking (to kernel functions) to reduce the run-time linking overhead, thus improving the loading speed. Third, based on relocatable ELF and the modular design of the dynamic linker and loader, our approach can be easily ported to different platforms. Fourth, by maintaining a kernel jump table, we provide a clean isolation between kernel and application development.

We have implemented the dynamic linking and loading mechanism on *SenSpire OS*, a micro sensor node operating system. The evaluation results show that our design and implementation meet our design goals: the code size of our SELF format is only 15%–30% of that of standard ELF, 38%–83% of that of CELF, a compact ELF format for the Contiki operating system; the loading speed improvement varies from 40%–50% compared to the standard mechanism; our design is portable to both MicaZ and TelosB motes, and we allow updating both application modules and kernel services in isolation without prior knowledge about the whole system information.

Acknowledgements

The authors would like to thank Shuopei Meng for his initial implementation of this work. This work is supported by the National Basic Research Program of China (973 Program) under grant No. 2006CB303000, and in part by NSERC Discovery Grant 341823-07, NSERC Strategic Grant STPGP 364910-08 and FQRNT Grant 2010-NC-131844.

References

[1] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He. The LiteOS Operating System: Towards Unix-like Abstractions for Wireless Sensor Networks. In *ACM/IEEE IPSN*, St. Loius, Missouri, USA, April 2008.

[2] H. Cha, S. Choi, I. Jung, H. Kim, H. Shin, J. Yoo, and C. Yoon. RETOS: Resilient, Expandable, and Threaded Operating System for Wireless Sensor Networks. In *ACM/IEEE IPSN*, Cambridge, Massachusetts, USA, April 2007.

[3] W. Dong, C. Chen, X. Liu, and J. Bu. Providing OS Support for Wireless Sensor Networks: Challenges and Approaches. *IEEE Communications Surveys and Tutorials*, to appear.

[4] W. Dong, C. Chen, X. Liu, K. Zheng, and J. Bu. SenSpire OS: A Predictable, Flexible, and Efficient OS for Wireless Sensor Networks. Technical report, ZJU-CS-2008-01, Zhejiang University, 2008.

[5] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks. In *ACM SenSys*, Boulder, Colorado, USA, November 2006.

[6] A. Dunkels, B. Grönvall, and T. Voigt. Contiki—a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *EmNets*, Tampa, Florida, USA, November 2004.

[7] M. Ekman and H. Thane. Dynamic Patching of Embedded Software. In *IEEE RTAS*, Bellevue, WA, USA, April 2007.

[8] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A Dynamic Operating System for Sensor Nodes. In *ACM MobiSys*, Seattle, Washington, USA, June 2005.

[9] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *ACM SenSys*, Baltimore, Maryland, November 2004.

[10] J. Koshy and R. Pandey. Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks. In *EWSN*, Istanbul, Turkey, February 2005.

[11] M. D. Krasniewski, R. K. Panta, S. Bagchi, C.-L. Yang, and W. J. Chappell. Energy-efficient on-demand reprogramming of large-scale sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 4(1):1–38, 2008.

[12] S. S. Kulkarni and L. Wang. MNP: Multihop Network Reprogramming Service for Sensor Networks. In *IEEE ICDCS*, 2005.

[13] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann, 2000.

[14] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS*, 2002.

[15] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks. In *EWSN*, Zurich, Switzerland, February 2006.

[16] L. Mottola, G. P. Picco, and A. A. Sheikh. FiGaRo: Fine-Grained Software Reconfiguration for Wireless Sensor Networks. In *EWSN*, Bologna, Italy, February 2008.

[17] V. Naik, A. Arora, P. Sinha, and H. Zhang. Sprinkler: A Reliable and Energy Efficient Data Dissemination Service for Wireless Embedded Devices. In *IEEE RTSS*, 2005.

[18] R. K. Panta, I. Khalil, and S. Bagchi. Stream: Low Overhead Wireless Reprogramming for Sensor Networks. In *IEEE INFOCOM*, Anchorage, Alaska, USA, May 2007.

[19] H. Shin and H. Cha. Supporting Application-Oriented Kernel Functionality for Resource Constrained Wireless Sensor Nodes. In *MSN*, Hong Kong, China, December 2006.

[20] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: Scalable Sensor Network Simulation with Precise Timing. In *ACM/IEEE IPSN*, 2005.

[21] N. Tsiftes, A. Dunkels, and T. Voigt. Efficient Sensor Network Reprogramming through Compression of Executable Modules. In *IEEE SECON*, 2008.

[22] P. von Richenbash and R. Wattenhofer. Decoding Code on a Sensor Node. In *IEEE DCOSS*, Santorini Island, Greece, June 2008.

[23] Q. Wang, Y. Zhu, and L. Cheng. Reprogramming Wireless Sensor Networks: Challenges and Approaches. *IEEE Network Magazine*, 20(3):48–55, 2006.