An Online Reprogrammable Operating System for Wireless Sensor Networks

HSUNG-PIN CHANG, YU-CHIEH LIN AND DA-WEI CHANG

Department of Computer Science and Information Engineering
National Chung Hsing University
Taichung, 402 Taiwan

Wireless sensor networks enjoy some unique characteristics. For example, sensor network are often autonomous, long-lived and rely on battery as the power source. In this paper, we thus improve the SOS kernel to address these unique characteristics. Firstly, we design and implement the hot-swapping capability in SOS that allows a module to be upgraded on the fly. In our system, the hot-swapping procedure is an online process and the execution state of the old module can be properly transferred to the new module. We also allow interface changes during hot-swapping. Moreover, we enable hot-swapping of not only application modules but also kernel modules. Finally, our hot-swapping procedure is lightweight in that, during hot-swapping, the job of module-linking is offloaded to the server to reduce the reprogramming cost in sensor nodes.

In addition to supporting hot-swapping, we also enhance the system call performance in SOS by caching the access results of the system call jump table. Furthermore, we replace the first-fit flash memory allocation scheme in SOS by a technique that relies on erase counters to evenly distribute memory traffic around the flash memory. Moreover, the maintenance of erase counters of each block is also offloaded to the server. We have implemented our system on the Mica2 mote. Evaluations reveal that our kernel can effectively improve the performance of SOS by the above three mechanisms. For example, we support hot-swapping of both kernel and application modules while incurring a negligible overhead. Hot-swapping an application module with a size of 2.5 Kbytes can be done within 160 processor cycles. In addition, we also reduce the system call invocation overhead by about 15%. Finally, our new flash allocation scheme allows flash blocks to be erased more evenly to prolong the lifetime of flash memory.

Keywords: hot-swapping, reprogramming, dynamic software update, linking, SOS, operating systems, wireless sensor networks

1. INTRODUCTION

A wireless sensor network (WSN) enjoy some unique characteristics. For example, each sensor node offers integrated computing, communication, storing and sensing capabilities and behaves like a battery-operated embedded tiny computer. In addition, sensor nodes are often deployed far from human access, such as the forest canopy [1] and the backs of zebras [2]. Consequently, once deployed, it will be impractical to reach each individual node. Thus, wireless sensor nodes are expected to be both autonomous and long-lived. Due to above unique characteristics, a well-behaved operating systems designed...
for wireless sensor networks must take these issues into account.

For example, owing to the long-lived property, the software running on each sensor node will necessarily be reprogrammed during its lifetime. The need to reprogram the software arises from fixing bugs, improving the performance, re-tasking functions, and providing new features. Hence, the operating systems running on sensor nodes must have the ability to enable dynamic software updates. For another example, sensor node always relies on battery as the power source. Thus, a sensor network operating system must also address the power issue. There have been many operating systems proposed for WSNs. SOS is one of these operating systems and is well-know by its support of network reprogramming capability [3]. In this paper, we improve the SOS to make it further suitable for WSNs. Specifically, we propose a light-weight on-line reprogramming scheme, eliminate the system call overhead, and enhance the flash allocation scheme in SOS.

Although SOS supports reprogramming capability, nevertheless, it only allows application modules to be dynamically loaded. In fact, not only applications but also the kernel requires bug fixes, code updates and function extensions. Moreover, in SOS, an application module can be upgraded only by first deleting the old version and then inserting the new version, causing the loss of the module state during the upgrade. This would be a problem if the module encapsulates important run-time information. Consequently, we firstly modify the SOS operating system to support hot-swapping capability, in which a module can be upgraded online and has no impact on the normal operation of sensor nodes. During hot-swapping, the execution state of the old module can be properly transferred to the new module. Thus, the new module can resume from where the old module left off. Furthermore, previous studies have shown that interface changes are common during software evolution [4, 5]. Thus, we also support interface changes that allow a module to be upgraded with a different interface during hot-swapping. Moreover, not only application modules but also kernel modules are hot-swappable in our system.

Software hot-swapping was first implemented in the K42 operating system [6]. Nevertheless, K42 is a general-purpose operating system that runs on computers with rich resources. Due to the energy and memory constraints of sensor nodes, we reduce the runtime execution overhead during hot-swapping by offloading the job of module-linking to a more capable device, usually the server that is used to develop or disseminate modules. Thus, our hot-swapping is lightweight since a module in a sensor node can be hot-swapped with minimal memory, computation, and communication requirements, which is especially attractive in WSNs.

In addition to the lightweight hot-swapping with interface change support, our work also makes two furthermore improvements to the SOS kernel. In SOS, application modules invoke system calls through a jump table. Nevertheless, the jump table is stored in the flash memory. Compared with RAM, accessing flash memory is extremely time consuming [7]. Thus, we eliminate the flash access overhead while keeping the jump table intact by caching the addresses of the system call handlers in internal SRAM. In addition, SOS supports dynamic flash memory allocation and uses a first-fit policy as the allocation discipline. Nevertheless, first-fit allocation directs most memory traffic to the first few flash blocks and would quickly exhaust the lifetime of these blocks. Since sensor network are expected to be long-lived and thus the flash memory must be used for a long time. Accordingly, we replace the first fit policy with an erase-counter based scheme to ensure that all blocks are erased evenly. Furthermore, the maintenance of erase counters of each
block is offloaded to the server to not only avoid the losses of erase counters if power is lost when updating the erase counters in sensor nodes [8], but also reduce the run-time overhead of sensor nodes.

The remainder of this paper is organized as follows. Section 2 gives the background to our work. Section 3 shows our lightweight hot-swapping scheme. Firstly, we describe our implementation solutions in response to the requirements for hot-swapping and then present the implementation scheme of offloading module-linking function to the server. The improvements to the SOS kernel are shown in section 4. Finally, the performance evaluation and concluding remarks are shown in sections 5 and 6, respectively.

2. BACKGROUND

In this section, we first review previous work related to dynamic software updates in section 2.1. Then, section 2.2 introduces the SOS kernel architecture. Finally, section 2.3 describes the requirements for supporting hot-swapping.

2.1 Related Work

There have been many techniques proposed to support network reprogramming capability in WSNs. To facilitate the discussions, we classify them into four categories: full image replacement, differential patching, virtual machines, and dynamically loadable modules.

**Full Image Replacement:** This is the default reprogramming method adopted by XNP [9] and Deluge [10] provided in TinyOS [11]. TinyOS is a component-based operating system. A component ranges from low-level part of a device driver to application-level part of a specific task or a function. All required components together with a scheduler are statically linked at compiler time. To reprogram a sensor node, the new complete TinyOS system image is re-distributed from the server to the sensor nodes.

As a result, reprogramming cost is expensive even in the case that only a single variable or an instruction in a component is modified. Moreover, after the completion of downloading process, the sensor nodes must reboot themselves to activate the new image, causing the operation of sensor nodes to be stopped and the current execution state to be lost. Although an external mechanism can be provided to save the execution state to non-volatile memory and restore it after the reboot, this approach is still an expensive operation.

**Differential Patching Schemes:** To reduce the amount of data transmitted during reprogramming in XNP, differential patching schemes are proposed [12, 13]. Based on the assumption that image changes such as bug fixes are usually small, they only distribute the binary differences between the original system image and the new one. After receiving the differences, the new image could be generated from the received differences and the current image stored on the sensor node.

However, since previous differential patching schemes are based on XNP, sensor nodes still need to reboot themselves to execute the new image. Nevertheless, the idea of differential patching can be applied at the module level in SOS and in our hot-swapping system since it is orthogonal to the reprogramming methodology.
Virtual Machines: The need for flexible and efficient network reprogramming has inspired virtual machines [14, 15]. By running a virtual machine on each sensor node, a WSN can be reprogrammed by disseminating new application bytecodes for that virtual machine. Since virtual machines provide a higher level interface, the size of the bytecode is extremely small as compared to the native code image. This is attractive in WSNs since it reduces the energy for code dissemination during reprogramming.

Although reprogramming costs are cheap, the cost of running virtual machines on devices with strict resource constraints such as sensor nodes would be high. This is because dynamically interpreting bytecodes can have a significant computational overhead. Moreover, unlike the Java virtual machine that is generic enough, virtual machines in WSNs are often tailored for specific applications. Thus, the reprogramming capability is limited by the expressibility of the underlying virtual machine. Finally, the reprogramming ability is applied only to applications running on the virtual machine. Neither the virtual machine itself nor the kernel can be reprogrammed.

Dynamically Loadable Modules: Dynamically loadable modules can be used to solve the problems in the previous approaches. Only the module containing the updates needs to be re-distributed and reprogrammed. Actually, this approach has been widely used in traditional operating systems. SOS and Contiki are examples of operating systems that support dynamically loadable modules in WSNs [3, 16].

Nevertheless, SOS and Contiki only allow application modules to be dynamically loaded. Furthermore, when an existing module needs to be upgraded, it must be completely removed and its current state and allocated system resources must be discarded. Thus, the new module needs to be run from the beginning and to acquire the necessary resources again. This not only increases the reprogramming latency but also degrades the system availability.

2.2 SOS Architecture

Since our work is based on SOS, we describe the primary SOS system architecture in this subsection. SOS consists of a common kernel and a set of dynamic application modules. The common kernel includes scheduling, messaging passing, dynamic memory management, and module management. An application module that implements a specific task or function can be dynamically loaded or unloaded at run time.

Interaction between application modules can be achieved by either passing messages to a module or by calling the exported functions of a module. Message passing is asynchronous in that all messages are queued and scheduled by the kernel scheduler. In contrast, function invocations are synchronous. A module can choose which of its functions to export by explicitly registering these functions to the SOS kernel. The kernel keeps track of all the exported functions of a module in the module’s function control block (FCB). Before a module can call an exported function provided by another module, it has to subscribe to the SOS kernel. If the subscription succeeds, the kernel returns a pointer to the function pointer of the subscribed function, which can be dereferenced by the subscriber to invoke the function. Thus, function calls between application modules are possible only after the function registration and subscription procedures.

In addition to invoking functions exported by other modules, an application module
can also request kernel services. Nevertheless, each system call must go through a system jump table. SOS adopts the jump table to allow application modules to be loosely coupled to the kernel. Thus, when an SOS kernel needs to be upgraded, it does not require all SOS application modules to be recompiled, assuming the structure of the jump table unchanged.

2.3 General Requirements of Hot-Swapping

As shown in K42 [6], to support hot-swapping, a system must achieve four capabilities: component boundaries, mutual consistent states, state transfer, and external references. However, as shown in [4, 5], interface changes are common during software evolution. Thus, we extend the four requirements to include the interface changes.

Component Boundaries: A component must be self-contained and have a clear boundary so that the system can identify the target to be hot-swapped. Using object-oriented programming languages can easily satisfy this requirement. Nevertheless, in a system that uses traditional languages, achieving clear component boundaries requires special programming disciplines that properly encapsulate data and functions within a module and expose a well-defined interface to other modules.

Mutually Consistent States: The state of the target module must be consistent before and after hot-swapping. If the target module is currently active while it is under hot-swapping, it would face the race condition problem. One possible solution is to ensure that the target module is in a quiescent state before it can be hot-swapped [6, 17]. A module is said to be in a quiescent state when all active use of the module is finished, i.e., this module is not currently in use. In addition to the quiescent state scheme, synchronization technique, such as synchronization protocol and semaphore, can also be used to achieve a mutually consistent state [18].

State Transfer: Each online module encapsulates its own execution state. It would be a problem if this state were lost during hot-swapping. Thus, all of the state required to ensure proper execution must be exactly transferred to the new module. Since the new module may define a different data set and data usage, the selection of state that must be transferred can only be determined by the involved modules.

External References: A module can provide services to other modules through external references. Whenever a module is hot-swapped, all of the relevant external references must be redirected to the appropriate parts of the new module. Otherwise, these external references would become dangling and point to the wrong addresses. Reference count, reference list and indirection are three common solutions to resolve external references [18].

Interface Changes: During hot-swapping, a new module may have a different interface compared to the old one. This includes the addition, deletion, or renaming of functions or arguments. Since other modules may depend on the interface of the old module, after hot-swapping, we must provide a mechanism to enable these dependent modules continue to run correctly.
3. A LIGHTWEIGHT HOT-SWAPPING SCHEME

In this section, we first show our hot-swapping implementation in SOS. Then, we present our offloaded linking scheme that offloads the linking overhead to the server to achieve the lightweight hot-swapping for sensor nodes.

3.1 Hot-Swapping Implementation

In this section, we describe the hot-swapping implementation according to the five requirements mentioned above. For ease of discussion, the implementation schemes for application module and kernel module hot-swapping are presented separately if necessary.

3.1.1 Preserving component boundaries

**Application modules:** As shown in section 2.2, SOS has already supported application modules. To enable application module hot-swapping, we follow the module definition in SOS. Consequently, an original SOS application module can be hot-swapped in our system with only few or even no modifications. The only one possible modification is adding the state transfer mechanism if necessary, which is described in section 3.1.3. Thus, our system preserves backward compatibility to the original SOS application modules.

**Kernel modules:** SOS has a common kernel that implements message passing, scheduler, dynamic memory management, module management, etc. To support kernel module hot-swapping, we modularize the original SOS kernel based on the above functionality. The encapsulation scheme of the kernel modules is similar to that of the application modules. Furthermore, to decouple dependence among kernel modules, we replace the direct function calls between kernel modules with function pointer references. For example, the function call \texttt{mq\_enqueue(&schedpg, m)} in SOS is replaced with \texttt{(*mq\_enqueuePtr)(&schedpg, m)}, where the \texttt{mq\_enqueuePtr} is a function pointer that points to the address of function \texttt{mq\_enqueue()}. This scheme has been widely used in the design of Linux kernel module.

3.1.2 Guaranteeing mutually consistent states

SOS adopts the First-In-First-Out (FIFO) scheduling scheme; that is, a message is handled only when the processing of the previous message is completed. Thus, when the module management subsystem is activated to perform hot-swapping, the target module is already in a quiescent state.

3.1.3 Performing state transfer

In SOS, module state is stored in a block of RAM separated from the module code area and is managed by the kernel. Therefore, an intuitive approach to accomplish state transfer is to directly copy the address pointer to this memory block from the old module to the new one. In our implementation, we allow a new module to specify whether it has the same module state as the old one. If it does, state transfer is efficient since it only involves copying a pointer.
Fig. 1. Definition and declaration used in an old module.

```c
typedef struct {
    func_ch_ptr get_hdr_size;
    int16_t timer_ticks;
    uint32_t seq_no;
    sur_pid_t dest_pid;
    SurgeMsg* msg;
} surge_state_t;
```

(a) The MSG STATE PACK message handler.  
(b) The MSG STATE SEND message handler.

However, in some cases, the data set and data usage may be different between the old and new modules. In this situation, we allow both the old and the new modules to cooperate with each other to achieve state transfer. During hot-swapping, the module management subsystem sends a MSG_STATE_PACK message to the old module. After receiving the message, the old module packs its state into another MSG_STATE_SEND message and sends this message to the new module. Then, the new module unpacks this message and restores the state from the unpacked message to continue its execution from where the old module left off. Accordingly, to support state transfer during hot-swapping, a module only needs to implement two extra handlers for the messages MSG_STATE_PACK and MSG_STATE_SEND.

We give a simple example to demonstrate the state transfer procedure. Fig. 1 shows the structure `surge_state_t` defined in an old module. In the new module, the `surge_state_t` adds a new member, `avg`, for storing the average value. When the old module receives the MSG_STATE_PACK message, as shown in Fig. 2 (a), it dynamically allocates a memory block pointed by the `state` pointer. Then, it copies the current state of the variable `s`, whose data type is the old `surge_state_t` structure, into the allocated memory block. Finally, it sends a MSG_STATE_SEND message, whose content is the memory block pointed by `state`, to the new module. In other words, the memory block acts as the container of the module state. Thus, the old module’s state is transferred by sending this memory block to the new module. Fig. 2 (b) shows the MSG_STATE_SEND message handler in the new module. In this handler, the new module first retrieves the old module’s state from the message and stores the state temporarily in the `oldstate` variable, whose type is the original `surge_state_t` structure. Then, it restores the state from variable `oldstate` to variable `s`, whose data type is the new `surge_state_t` structure, and initializes the value of the new member, `avg` as zero. After restoring the module state, the received memory block can be freed.
Note that, the module state might include pointers, e.g., the `smsg` variable in the `surge_state_t` structure that points to a list of received messages. In this situation, as shown in Fig. 2, the old module only needs to pack and transfer the `smsg` pointer, instead of the complete list of received messages, to the new module. Then, the new module can access the list of received message after restoring the `smsg` pointer value to the \( s \rightarrow smsg \) variable.

The solution described above assumes that the programmers of the new module know the version number, and so the data structures, of the target old module. Nevertheless, in a heterogeneous sensor network, the module versions might not be the same among all nodes. Consequently, we provide a mechanism to help the programmers to obtain such an information. Since all modules are disseminated by the server, the server could maintain a data structure to keep track of the version number of each module on each sensor node. Fig. 3 shows our data structure. Each module has an associated linking list and each node in the list has three fields: `version_number`, `start_id`, and `end_id`, which mean that all the sensor nodes with IDs between `start_id` and `end_id` have the same module version that equals to `version_number`. For example, the version number of module \( a \) on nodes 1 to 10 and 16 to 21 is equal to 2.

![Fig. 3. The data structure for keeping track of the module versions on each node.](image1.png)

In addition to the module state, the old module may also hold some system resources and thus we have to transfer these system resources to the new module as well. Nevertheless, SOS does not support locks or any synchronous mechanism in kernel. In the current version of SOS, the only possible system resource is a timer that has not yet been expired. Transferring the timer only involves changing the owner of the timer to the new module. Thus, when the timer expires, the kernel can notify the new module by sending a message to the new module.

Finally, although the old module is in a quiescent state when the hot-swapping procedure begins, transferring the old module’s state by passing messages would face the race condition problem due to the asynchronous behavior of message passing. As shown in Fig. 4, assume that there have been two messages, \( A \) and \( B \), in the message queue when the message `MSG_STATE_SEND`, denoted as \( S \), is sent. Furthermore, assume that the destination of message \( A \) is the target module. Since SOS uses the FIFO queuing principle, message \( A \) will be de-queued and be processed before message \( S \). However, we cannot deliver message \( A \) to the old module since the old module’s state has been packed and transferred to the new module. Processing message \( A \) would change its state again and cause the state encapsulated in message \( S \) to be invalid. We can neither deliver message \( A \) to the new module since its state has not yet been initialized. Delivering message \( B \) before
message $S$, even though $B$'s destination is not the target module is also forbidden. This is because the destination module of message $B$ may call external functions exported by the target module, which in turn changes the state of the target module.

To solve this problem, we modify the SOS scheduler so that the state transfer messages (i.e., MSG\_STATE\_PACK and MSG\_STATE\_SEND) are set to the highest priority. Thus, the original FIFO queue is changed to a priority queue. By this method, we can guarantee that the hot-swapping process is atomic and would not be interrupted by the intervention of other modules.

3.1.4 Resolving external references

Application modules: As shown in section 2.2, function calls between application modules in SOS are indirect. This extra level of indirection allows us to easily resolve external references without needing to modify all the subscriber modules. When a module is hot-swapped, we change the function pointers, stored in the subscribers' FCBs, corresponding to the exported functions of the module to point to the new addresses. Then, the subscriber modules can continue to call the functions exported by the module.

Kernel modules: An external reference to a kernel module would be from an application module or another kernel module. According to the source of an external reference, the solutions to resolve external reference are discussed separately.

- References from application modules: In SOS, all system calls are exported in the jump table. Thus, if a kernel module having one or more functions exported in the jump table is hot-swapped, we only need to search the jump table to find the corresponding entries and modify their values to refer to the new addresses.
- References from other kernel modules: As shown in section 3.1.1, we decouple the dependence among kernel modules by replacing direct function calls between kernel modules with function pointers. Consequently, when a module is hot-swapped, we only need to modify the values of the corresponding function pointers.

3.1.5 Supporting interface changes

This problem is similar to previous work that allowed unmodified Linux device drivers to be used in other operating systems [19, 20]. For example, the developers of OS-Kit introduce glue codes to encapsulate Linux device drivers within their OSKit environment. In [4], the authors also adopt a similar approach that uses an adaptor object in the original K42 system. In this paper, we generalize the techniques explored in [4, 19, 20].

Like previous approaches mentioned above, we also introduce a converter to wrap the new module to support interface changes. The converter should be provided by the implementers of the new module. The design is shown in Fig. 5. The old dependent modules can continue to run by using the old interface exported by the converter. In contrast, newly uploaded modules can directly utilize the new interface. Notably, the converter can be removed when all of the old dependent modules are upgraded to use the new interface.

Nevertheless, our system can cope with some interface changes, such as renaming functions or arguments, without a converter. For example, assume that the name of a function is originally called func\_A in an old module module\_A and is changed be to func\_B in
the new module module_B. During the procedure of resolving external references mentioned in section 3.1.4, we only need to modify the values of those function pointers in subscribers, i.e., dependent modules, of module_A, that intend to point to the address of func_B to point to the address of func_A. However, we need to know which functions have their names changed in the new module and their corresponding new names. Thus, we require the module programmers to provide a description file to describe the above information. Consequently, for these kinds of interface changes, we can avoid the converter overhead.

Finally, not all interface changes can be solved by a converter, e.g., complex restructuring changes that are not backwards-compatible. Consequently, if the programmers do not supply with a converter or a description file, during the procedure of resolving external references, we will direct all invocations from old dependent modules to a kernel supplied exception handler, which sets a global error variable and returns immediately to the callers. In fact, if a module is hot-swapped with complex restructuring changes, a reasonable approach for the programmers is to delete or hot-swap all old and dependent modules first.

3.2 Offloading Kernel Module Linking to the Server

3.2.1 Offloaded run-time linking

A module may contain references to symbols, which can be functions or data variables, defined in other modules or in the kernel. During hot-swapping, all of the external references must be resolved to refer to the actual address before the module can be executed. The process of resolving these external references is called linking.

Most server and desktop operating systems use the in-situ dynamic linking approach, which links a module on a machine when the module is going to be executed on that machine. Some sensor network operating systems such as the new Contiki kernel [21] and SOS also use this approach that links a dynamically loadable module in each sensor node. In Linux, there is also a tool called prelink that resolves symbols used by a program in advance to reduce the time it takes to launch the program [22]. Nevertheless, prelink is also a kind of in-situ linking approach, although it is not a dynamic linking scheme. The advantage of in-situ linking is its flexibility. For example, it can easily support module stacking, which allows a module to export symbols to other modules.

Unfortunately, using in-situ linking in WSNs imposes a significant execution overhead. Firstly, it requires a linker in each sensor node. Furthermore, the linker must embed an object reader for each supported object file format. Secondly, it usually results in a larger code dissemination overhead since the size of an un-linked module is usually larger than that of a linked module. This is because the former is in the form of an object file that contains more information, such as a symbol table and other data structures, which is needed for linking. Thirdly, each sensor node must maintain a symbol table that contains
the names/identifiers of all of the external symbols defined in the systems and their corresponding addresses, which in turn increases the memory requirements. As a consequence, in-situ linking results in a larger processor, memory and energy overhead.

To reduce the overheads mentioned above during hot-swapping, we offload the linking task to resource-rich server nodes. This approach is similar to the method used in the original Contiki system [16]. However, three differences in module linking exist between the original Contiki system and ours. Firstly, a prelinked module in the original Contiki system cannot execute on a sensor node if the system core of the node has changed. This is because the Contiki used direct function invocation. On the basis of modular design and indirect function invocation mentioned in section 3.1, we allow a linked module to continue to execute on a sensor node after some of the applications or kernel modules on the node have been hot-swapped. Secondly, the original Contiki system performed module linking by consulting a single symbol table. By contrast, our system is not restricted to use a single symbol table. Multiple symbol tables can be used for a heterogeneous WSN that consists of multiple groups of homogeneous sensor nodes. Thirdly, the system core’s symbol table of the original Contiki system is not extensible, and hence it does not support module stacking. We support module stacking by allowing a module to export symbols to extend the kernel’s symbol table. In summary, our offloaded run-time linking combines the flexibility of in-situ linking and the light overhead of the prelink approach used in the previous WSNs. Specifically, our approach enjoys the following benefits.

- Eliminate the need of a dynamic linker and a kernel system table on each sensor node.
- Relieve the load of the processors on sensor nodes from doing the linking job.
- Disseminate linked and usually smaller modules.
- Support dynamic module stacking and multiple symbol tables.

Note that, in the current implementation, we only offload the task of kernel module linking to the server. For an application module, references to external symbols are still resolved by using the aforementioned registration and subscription procedures. This is because we wish to preserve the backward compatibility and to allow all the existing SOS application modules to become hot-swappable with as few modifications as possible. There is no backward compatibility issue on kernel modules since the original SOS does not support them.

### 3.2.2 Implementation scheme

To offload the linking task to the server, one possible implementation scheme is to modify the linker on the server [23]. Nevertheless, such an approach is complicated, and is prone to error and could even destroy the semantics of the linker. In this paper, we adopt a much simpler implementation path that does not need to modify the linker. Fig. 6 shows the overall flow of kernel module linking. We implemented a patch tool to process the external references in the object files. The processing steps are detailed as follows.

1. Firstly, before all the nodes are deployed, the patch tool constructs the kernel’s initial symbol table on the server based on the kernel object file (i.e., blank.elf). This is achieved by using the *avr-readelf* utility to extract the kernel symbol information from the Kernel object file.
2. Assume that we want to hot-swap an existing kernel module on sensor nodes with a new one, called `new_module`. The source file `new_module.c` is first compiled by `gcc` to produce the object file, i.e., `new_module.o`, with ELF format.

3. Then, our patch tool parses the `new_module.o` file and consults the kernel’s symbol table to resolve all the external symbols referenced by the module. In addition, the kernel’s symbol table is updated according to the symbols exported, removed, or updated by `new_module.o`. Note that, the addresses of symbols exported or updated are calculated by adding the starting address of the `new_module` with their offsets shown in the symbol table of the module. Furthermore, the starting address of a module in sensor nodes is determined by the server, which is shown in section 4.2.

4. After completion of the patch process, we invoke the `gcc` again to produce a linked kernel module image.

5. Finally, the linked kernel module image is transmitted to each sensor node.

6. Steps 2-6 are repeatedly executed when a kernel module is hot-swapped.

We demonstrate the detailed operation of our patch tool and the offloaded linking procedure with a simple example. An ELF object file contains a symbol table and a relocation table for the linking purpose. Fig. 7 shows the relocation table in `new_module.o`. The relocation table contains a list of entries with each entry corresponding to an instruction or data address that needs to be updated. The first column shows the offset that needs to be updated and the last column specifies how to perform the update. From this table, we can find that six places need to be updated. This is because, as shown in Fig. 8 that presents the symbol table of `new_module.o`, three symbols, i.e., `section_map`, `section_salt`, and `alloc_map`, are undefined.

To resolve these undefined symbols, our patch tool searches the kernel’s symbol table, a part of which is shown in Fig. 9, to find the actual addresses of these symbols. Nevertheless, instead of directly patching the six addresses shown in the relocation table, our patch tool only updates the symbol table as shown in Fig. 8 to reflect the actual addresses of these symbols. Finally, we run the `gcc` again. Since the addresses of all these three variables are now resolved and defined in the symbol table, `gcc` can automatically complete the relocation task and produce the final linked module image. As shown above, our patch tool offers a much simpler and attractive implementation path.

### 3.2.3 Correctness analysis

In this subsection, we verify the correctness of the implementation mentioned in section 3.2.2. Specifically, we show that server-side linking with our patch tool has the same
Fig. 7. The relocation table in new module.

Fig. 8. The symbol table in new module.

Fig. 9. The kernel’s symbol table.

effect as in-situ linking on the sensor node in all cases. Notably, the job of linking is to assign addresses to all undefined external references (e.g., section_map in Fig. 8) in the symbol table of a module and to perform relocation based on the relocation table. Thus, for correctness verification, we must assure that the addresses assigned to all undefined external references by the patch tool are the same as those assigned by the in-situ linking scheme in all cases (i.e., consistency of address assignment) and the offloaded linking scheme achieves the same relocation result as the in-site dynamic linking.

We first verify the consistency of address assignment. As mentioned in section 3.2.2, the patch tool only consults the kernel’s symbol table on the server to resolve the addresses of undefined external references. This is because we only address on the offloaded linking of kernel modules and a kernel module can only contain external references to external symbols defined in the kernel or other kernel modules. Thus, all undefined external references of a kernel module can be resolved from the kernel’s symbol table. Therefore, to guarantee the consistency of address assignment, we only need to assure that the kernel’s symbol table kept on the server by the offloaded linking is consistent with that kept on each sensor node in the in-situ linking scheme in all cases. In other words, the consistency of address assignment is guaranteed to be held by the consistency of the kernel’s symbol tables. We demonstrate the consistency of the kernel’s symbol tables by the following proof. In the following, we refer the kernel’s symbol table simply as symbol table.

**Lemma** Assume that $ST_i.ol$ and $ST_i.is$ denote the symbol tables of the offloaded and in-situ linking schemes after $i$th operation of module insertion, removal, or hot-swapping on the sensor node. Furthermore, assume that $ST_i.ol$ contains $p(i)$ symbols $\{f_1, f_2, \ldots, f_{o(i)}\}$ and $ST_i.is$ contains $q(i)$ symbols $\{g_1, g_2, \ldots, g_{o(i)}\}$. Then we show that $ST_i.ol$ is equal to $ST_i.is$ for all $i = 0, 1, 2, \ldots, n$, where $n$ is any of a positive integer. Specifically, we show that the following three equations hold if $ST_i.ol$ is equal to $ST_i.is$.

\[
p(i) = q(i),
\]
\[
\{f_1, f_2, \ldots, f_{o(i)}\} = \{g_1, g_2, \ldots, g_{o(i)}\},
\]
\[ \text{addr}(f'_j) = \text{addr}(g'_j), \text{ where } 1 \leq j \leq p(i), \]

where \( \text{addr}(f'_j) \) denotes the address of symbol \( f'_j \). Eqs. (1) and (2) indicates that the two symbol tables have the same set of symbols, and the Eq. (3) means that the values (i.e., addresses) of the symbols are also consistent in the two tables.

**Proof:** Initially, when \( i = 0 \), \( ST_{o,ol} \) and \( ST_{o,is} \) denote the initial symbol tables of the off-loaded and in-situ linking schemes, respectively. Then, the following equations hold.

\[
\begin{align*}
p(0) &= q(0), \quad (4) \\
\{f_1^0, f_2^0, \ldots, f_{p(0)}^0\} &= \{g_1^0, g_2^0, \ldots, g_{q(0)}^0\}, \quad (5) \\
\text{addr}(f'_0) &= \text{addr}(g'_0), \text{ where } 1 \leq j \leq p(0). \quad (6)
\end{align*}
\]

The equations hold since that \( ST_{o,ol} \) and \( ST_{o,is} \) are both constructed based on the kernel’s object file. The only one difference between them is that \( ST_{o,ol} \) resides on the server whereas \( ST_{o,is} \) is placed on the sensor node. From Eqs. (4)-(6), \( ST_{o,ol} \) must be consistent with \( ST_{o,is} \).

Assume that, which \( i = n \), \( ST_{o,ol} \) is consistent with \( ST_{o,is} \). That is, from Eqs. (1)-(3), the following equations must hold.

\[
\begin{align*}
p(n) &= q(n), \quad (7) \\
\{f_1^n, f_2^n, \ldots, f_{p(n)}^n\} &= \{g_1^n, g_2^n, \ldots, g_{q(n)}^n\}, \quad (8) \\
\text{addr}(f'_n) &= \text{addr}(g'_n), \text{ where } 1 \leq j \leq p(n). \quad (9)
\end{align*}
\]

Then, we show that, when \( i = n + 1 \), i.e., after \( (n + 1) \)th operation, \( ST_{o+1,ol} \) is still consistent with \( ST_{o+1,is} \). If the \( (n + 1) \)th operation is module insertion, we assume that the newly inserted module \( M \) exports \( k \) symbols \( \{h_1, h_2, \ldots, h_k\} \). Then, from Eqs. (7)-(9),

\[
\begin{align*}
p(n + 1) &= p(n) + k = q(n) + k = q(n + 1), \quad (10) \\
\{f_1^{n+1}, f_2^{n+1}, \ldots, f_{p(n+1)}^{n+1}\} &= \{f_1^n, f_2^n, \ldots, f_{p(n)}^n\} \cup \{h_1, h_2, \ldots, h_k\} \\
&= \{g_1^n, g_2^n, \ldots, g_{q(n)}^n\} \cup \{h_1, h_2, \ldots, h_k\} \\
&= \{g_1^{n+1}, g_2^{n+1}, \ldots, g_{q(n+1)}^{n+1}\}, \quad (11) \\
\text{addr}(f'_n) &= \text{addr}(g'_n), \text{ where } 1 \leq j \leq p(n) + k = p(n + 1). \quad (12)
\end{align*}
\]

Note that, as shown in step 3 of the proposed implementation mentioned in section 3.2.2, the starting address of \( M \) is determined by the server and thus, the patch tool can know the starting address of \( M \). Accordingly, the starting addresses of \( M \) derived in both offloaded and in-situ linking methods are the same. Therefore, Eq. (12) holds since the addresses of symbols exported by \( M \) are calculated by adding the starting address of \( M \) to their offsets shown in the \( M \)'s symbol table.

If the \( (n + 1) \)th operation is module removal, we assume that a module \( M \), which exports \( k \) symbols \( \{h_1, h_2, \ldots, h_k\} \), is removed from the sensor node. From Eqs. (7)-(9),

\[
p(n + 1) = p(n) - k = q(n) - k = q(n + 1), \quad (13)
\]
\begin{align*}
\{f_1^{n+1}, f_2^{n+1}, \ldots, f_{p(n+1)}^{n+1}\} &= \{f_1^n, f_2^n, \ldots, f_{p(n)}^n\} - \{h_1, h_2, \ldots, h_k\} \\
&= \{g_1^{n+1}, g_2^{n+1}, \ldots, g_{q(n+1)}^{n+1}\} - \{h_1, h_2, \ldots, h_k\} \\
&= \{g_1^{n+1}, g_2^{n+1}, \ldots, g_{q(n+1)}^{n+1}\}, \quad (14)
\end{align*}

\text{addr}(f_i^{n+1}) = \text{addr}(g_i^{n+1}), \text{ where } 1 \leq i \leq p(n) - k = p(n + 1). \quad (15)

For the both linking schemes, the symbols defined in \(M\) are removed from the symbol table. In addition, Eq. (15) holds since the addresses of the remaining symbols in both symbol tables are unchanged and must be same due to Eq. (9). Consequently, the two tables are still consistent after the module removal.

If the \((n + 1)\)th operation is module hot-swapping, we assume that an old module \(O\), which exports \(s\) symbols \(\{l_1, l_2, \ldots, l_s\}\), is hot-swapped by a new module \(M\), which exports \(k\) symbols \(\{h_1, h_2, \ldots, h_k\}\), on the sensor node. Therefore, from Eqs. (7)-(9),

\begin{align*}
p(n + 1) &= p(n) - s + k = q(n) - s + k = q(n + 1), \quad (16) \\
\{f_1^{n+1}, f_2^{n+1}, \ldots, f_{p(n+1)}^{n+1}\} &= (\{f_1^n, f_2^n, \ldots, f_{p(n)}^n\} - \{l_1, l_2, \ldots, l_s\}) \cup \{h_1, h_2, \ldots, h_k\} \\
&= (\{g_1^n, g_2^n, \ldots, g_{q(n)}^n\} - \{l_1, l_2, \ldots, l_s\}) \cup \{h_1, h_2, \ldots, h_k\} \\
&= \{g_1^{n+1}, g_2^{n+1}, \ldots, g_{q(n+1)}^{n+1}\}, \quad (17)
\end{align*}

\text{addr}(f_i^{n+1}) = \text{addr}(g_i^{n+1}), \text{ where } 1 \leq i \leq p(n) - s + k = p(n + 1). \quad (18)

For the module hot-swapping, both linking schemes remove the symbols defined in the old module but not defined in the new one, add the symbols defined in the new module but not defined in the old one, and update the addresses of symbols that are defined by both the modules. In addition, Eq. (18) holds for the same reason as that for Eq. (12). Thus, according to Eqs. (16)-(18), the two symbol tables are also the same after module hot-swapping.

From above discussions, when \(i = n + 1\), the consistency of the kernel’s symbol tables holds no matter the \((n + 1)\)th operation is module insertion, module removal, or module hot-swapping. Consequently, by the theory of mathematical induction, the symbol table maintained on the server by the offloaded linking scheme must be consistent with that kept on the sensor node by the in-situ linking scheme.

Next, we verify that the both linking schemes achieve the same relocation result even though our patch tool updates the symbol table of the module only and leaves the relocation table intact. As stated in the section 3.2.2, although our patch tool only updates the symbol table of the module by filling in the actual addresses of all undefined external references, however, \texttt{gcc} would automatically perform the relocation job by processing the relocation table. For each relocation entry in the relocation table, \texttt{gcc} would search the symbol table to find the address of the symbol shown in the relocation entry and use this address to patch the instruction or data to which the relocation entry points. Consequently, since (1) the symbol table on the server, used by offloaded linking, is kept consistent with that on each sensor node, used by in-situ linking, and (2) \texttt{gcc} would automatically perform the relocation job according to the patched symbol table of the to-be-linked module, the proposed offloaded linking scheme is ensured to have the same effect as the in-situ dynamic linking scheme, while having the advantage of offloading the linking overhead to the server.
4. IMPROVEMENTS OF SOS

In addition to introducing the hot-swapping capability to the SOS kernel, we also improve the SOS in two aspects. Specifically, we enhance the system call performance and improve the flash memory allocation scheme to achieve wear-leveling.

4.1 Reduction of System Call Overhead

As mentioned in section 2.2, in SOS, application modules invoke kernel functions via a system jump table. This helps the application modules and the SOS kernel remain decoupled from each other. Nevertheless, the jump table is stored in the flash memory. As shown in [7], compared to RAM, reading flash memory is extremely time-consuming. For example, reading one-byte data from a low power SRAM needs 70 ns in the Renesas M5M5256DFP SRAM [24]. However, it requires 150 ns reading one-byte data from Intel 28F128J3A-150 NOR flash [7]. Since system calls are usually invoked frequently, the accumulated overhead would be significant. One possible solution is to remove the jump table. However, such an approach would destroy the loosely coupled relationship between application modules and the kernel. Another solution is to move the jump table to the RAM. However, RAM is volatile. An accident power loss would cause all data, including the jump table, stored in RAM to be lost. Thus, a promising solution should reserve the jump table in the flash memory while reducing the system call overhead.

In this paper, we allow the application modules to cache the target addresses of kernel functions and provide a mechanism to notify the application modules when the cached results are invalid. After the first invocation of a kernel function, the application module can cache the target address of this kernel function. Then, in the following calls to the same kernel function, the application module bypasses the jump table and uses the cached address to directly access this kernel function. Once a kernel module that has functions exported in the jump table is hot-swapped, as mentioned in section 3.1.4, we must modify some entries of the jump table to reflect their new addresses. In addition, the kernel immediately broadcasts a `CACHE_INVALID` message, containing the names/identifiers of the kernel functions whose addresses have been changed to all application modules. Similar to the problem shown in Fig. 3, to prevent an application module from using the obsolete addresses, this message is also set to the highest priority, i.e., has the same priority as the state transfer messages. When an application module receives this message, it discards the corresponding cached addresses. New addresses can be cached again in the following system calls.

4.2 Improvement on the Flash Memory Allocation Scheme

SOS supports dynamic memory allocation. When a new module is inserted, the SOS kernel must dynamically allocate a memory area to place this new module. To simplify the design, SOS uses first-fit fixed-block memory allocation scheme.

In most sensor nodes, such as the Mote series, the program memory employs flash memory technology. Compared with conventional dynamic memory, flash memory reveals several unique features. One is that its content cannot be overwritten: it must be erased before new data can be written. Unfortunately, each flash memory block can only
endure a limited number of erase operations. Thus, the erase operations should be evenly distributed around the flash memory to prolong the flash memory lifespan. This technique is called wear-leveling or even-leveling [8].

However, due to first-fit allocation scheme in SOS, blocks with smaller numbers are erased and written more frequently. Furthermore, as described in section 1, sensor networks are expected to the long-lived and thus, the flash memory must be used for a long time. Consequently, a better allocation scheme should be adopted. Generally, there are two approaches to level the wear [8]. One way is to randomly select a free memory block upon receiving a memory allocation request. Nevertheless, techniques based on a random policy would not level the wear evenly since the life times of modules on each block are different [8]. The block allocated to a long-lived module would have less chance to be selected and is likely to be a less worn out block. The other approach is to keep track of the number of erase operations performed on each block. When receiving a memory allocation request, the block having the least erase counters is selected. Nevertheless, such an approach imposes an overhead on the sensor nodes since we have to keep track of the number of erase operations performed on each block. Furthermore, erase counters are often maintained in the flash memory. As a result, techniques using erase counters are susceptible to lose an erase counter if power is lost after a block is erased but before the new counter is written to the flash memory [8].

In this paper, we use an erase-counter based scheme. However, jobs of maintaining erase counters and selecting a least worn-out free block are done at the server in order to reduce the load of the sensor nodes and to avoid the loss of erase counters during power failure of the sensor nodes. The server can perform the maintenance and selection since all modules are disseminated by the server. When uploading or hot-swapping a module to the sensor nodes, the server can select the least worn-out free block, piggyback the selected block number with the module, and send the module to the sensor nodes. Finally, the erase counter of the selected block is incremented by one.

Note that, in many cases, the server does not need to maintain the erase counters of each block for each sensor node. A sensor network is often deployed as a homogeneous system or a heterogeneous system that consists of only a few groups of homogeneous sensor nodes. Thus, sensor nodes belonging to the same homogeneous system have the same erase counter for each block. Consequently, for each homogeneous group, we only need to maintain one copy of erase counters of each block. Furthermore, since the erase counters are maintained in the server, no loss of erase counters if power failures occur on the sensor nodes. In addition, we can also apply previous well-known techniques to avoid the loss of erase counters at the server since the server is more capable of tolerating the added complexity and cost of the techniques [25].

5. PERFORMANCE EVALUATION

In this section, we present the performance evaluation of our improvements on the SOS. The sensor node used in the experimental environment is a Mica2 MPR400 mote plus a MTS310 sensor board. The mote is comprised of a 900MHz Atmega128 processor, 512KB Flash ROM, and a Chipcon CC1000 radio operating at 915 MHz.

In the following, we first present the effectiveness and performance of the proposed
lightweight hot-swapping mechanism. Then, we show the effects of the other improvements on SOS. Finally, we give an analytical analysis of the saving of the energy and memory footprint caused by offloading the module-linking job to the server.

5.1 Effectiveness of Module Hot-Swapping

In the first experiment, we demonstrate the ability of application module hot-swapping by upgrading the surge application module during its execution time. Surge is a well-known data acquisition program. It periodically samples the light intensity via the light-sensing module on the MTS310 sensor board, and sends that value out to the sink. We tested two configurations in this experiment. In the first configuration, both versions of surge used the same data structures for their state information. Thus, there is no need for packing/unpacking state during the module hot-swap. Table 1 shows the result. The hot-swap took place after the old module had sent out the message with sequence number $0 \times 06$. From the sequence numbers shown in Table 1, the new module can successfully pick up from where the old module left off. Table 2 shows the result of the second configuration, in which the two modules use different data structures and thus packing/unpacking state is required. In this configuration, the old module does not have the average variable and the hot-swap took place after the old module had sent out the message with sequence number $0 \times 09$. From Table 2, the new module not only continues from where the old module left off but also extends the previous module state to include the average variable.

In the next experiment, we demonstrate the capability of hot-swapping the scheduler kernel module. In the original SOS kernel, application modules exchange messages via a single, FIFO ordered queue. To allow more flexibility and solve the race condition problem shown in sections 3.1.3 and 4.1, we implemented a new scheduler that replaces the single queue with two priority queues, \textit{ap-high} and \textit{ap-low}, where the priority of the former is higher than that of the latter.

To demonstrate that we can hot-swap the scheduler module dynamically, we implemented two application modules, APP01 and APP02. APP01 sends two messages, \textit{A} and \textit{B}.

Table 1. Module hot-swap without state packing/unpacking.

<table>
<thead>
<tr>
<th>Seq. No</th>
<th>0 × 4</th>
<th>0 × 5</th>
<th>0 × 6</th>
<th>HOT-SWAP</th>
<th>0 × 7</th>
<th>0 × 8</th>
<th>0 × 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>$0 \times 025E$</td>
<td>$0 \times 025D$</td>
<td>$0 \times 0253$</td>
<td>HOT-SWAP</td>
<td>$0 \times 0240$</td>
<td>$0 \times 0238$</td>
<td>$0 \times 0264$</td>
</tr>
</tbody>
</table>

Table 2. Module hot-swap with state packing/unpacking.

<table>
<thead>
<tr>
<th>Seq. No</th>
<th>0 × 7</th>
<th>0 × 8</th>
<th>0 × 9</th>
<th>HOT-SWAP</th>
<th>0 × A</th>
<th>0 × B</th>
<th>0 × C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>$0 \times 025E$</td>
<td>$0 \times 025D$</td>
<td>$0 \times 0253$</td>
<td>HOT-SWAP</td>
<td>$0 \times 213$</td>
<td>$0 \times 22A$</td>
<td>$0 \times 244$</td>
</tr>
<tr>
<td>Average</td>
<td>$0 \times 213$</td>
<td>$0 \times 21E$</td>
<td>$0 \times 231$</td>
<td>$0 \times 213$</td>
<td>$0 \times 22A$</td>
<td>$0 \times 244$</td>
<td></td>
</tr>
</tbody>
</table>

*The new module add a new variable: \textit{average}.

Table 3. Hot-swapping the scheduler module on message delivery (clock cycles).

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>HOT-SWAP</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3809928</td>
<td>4040384</td>
<td>4270846</td>
<td>HOT-SWAP</td>
<td>4501364</td>
<td>4731809</td>
<td>4962267</td>
</tr>
<tr>
<td>B</td>
<td>3809982</td>
<td>4040438</td>
<td>4270903</td>
<td>HOT-SWAP</td>
<td>4501428</td>
<td>4731703</td>
<td>4962161</td>
</tr>
</tbody>
</table>
In order every one second, and APP02 receives the messages and records the time when each message is received. During the experiment time, we replaced the old scheduler with the new one, and placed message $A$ in the ap-low queue and message $B$ in the ap-high queue. Table 3 presents the time (in clock cycles) each message was received in APP02. Before the hot-swap, message $A$ was received before message $B$ since the former was sent first and the messages were both queued in a single FIFO ordered queue. After the module hot-swap, however, message $B$ was placed in the ap-high queue and thus was delivered first.

Furthermore, we use the flash allocation module to demonstrate that we can update module interfaces during hot-swapping. In the original SOS, the alloc_section_in_map() function has two input parameters: start and map_size. Nevertheless, as described in section 4.2, the original FIFO allocation scheme is changed to an erase-counter based method and the block allocated is determined at the server. Consequently, after hot-swapping the flash allocation module, the above two parameters are removed in the new interface. Thus, we supply the new flash allocation module with a converter that ignores the two parameters and calls the new interface to return the starting address of the allocated block.

Fig. 10 shows the experimental result. The x-axis represents the block numbers and the y-axis represents the erase counter of each block. We randomly perform a sequence of two hundred operations such as module installation, removal, or hot-swap on a number of modules and record the number of erase operations on each flash block. At the first one hundred operations, the flash allocation scheme is FIFO. Then, we hot-swap the flash allocation module that changes the allocation scheme to be the erase-counter based one. Thus, in the last one hundred operations, the allocation scheme is changed to use the erase-counter based policy. The bars marked as FIFO show the number of erase operations performed on each block in the first one hundred operations and the bars marked as erase counters-based show the same result but in the last one hundred operations. As shown in Fig. 10, the dependent modules can continue to allocate flash memory blocks via the old interface even when the flash allocation module is hot-swapped with a new interface.

Finally, we demonstrate the module stacking capability achieved by our offloaded linking scheme. As shown above, we hot-swap the scheduler module so as to change the FIFO scheduling policy to the priority one. In addition, the new scheduler module implements a new function, get_info(), that records the number of messages that are placed in the ap-high and ap-low queues respectively. In this experiment, we insert a new module that periodically invokes the get_info() function exported by the new scheduler to verify
the module stacking capability. Table 4 shows the experimental results that present the accumulated number of messages placed in *ap-high* and *ap-low* queues in each invocation by the new module.

### 5.2 Performance of Module Hot-Swapping

In this section, we first present the space overhead of the hot-swapping mechanism. Then, we show the time for hot-swapping modules. Table 5 shows the code and data sizes of both the original and the modified versions of SOS. As shown in the table, adding the hot-swapping capability into SOS only causes an increment of the kernel code size by about 1 Kbytes and an increment of the kernel data size by 10 bytes. In addition to the kernel sizes, this table also shows the sizes of three modules, one application module (surge) and two kernel modules (flash-allocator and scheduler). The sizes of the kernel modules in original SOS are unavailable because SOS does not support kernel modules. Moreover, the surge application module has a slight increment on the size, which is due to the support of state transfer.

Fig. 11 (a) shows the time required to hot-swap application modules. We used three application modules with different sizes in the experiment, a null module (20 bytes), a surge module (688 bytes), and a tree routing module (2608 bytes). Module hot-swap involves checking the existence of the module, registering the new module, and removing the old one. As shown in the figure, hot-swapping a module that is 2.5 Kbytes in size can be done within 160 processor cycles. Fig. 11 (b) shows the time required to hot-swap kernel modules. Three kernel modules were used in the experiment, a null module (20 bytes), a scheduler module (478 bytes), and a flash memory management module (318 bytes).

Nevertheless, as shown in section 3.1.4, some kernel modules may provide services to the user modules and have their functions exported in the jump table. Thus, kernel module hot-swap may need to update the system jump table stored in the flash memory. We use the null kernel module, which only exports a single null function to the jump table, to measure the update cost of a flash memory. Table 6 shows the experimental result. From this table, it took 1202 clock cycles to update a single entry of the jump table, reflecting the fact that updating data in flash memory is extremely time-consuming and dominates the hot-swapping latency [7]. Table 6 also shows the respective times spent in the three

---

**Table 4. Accumulated number of messages in two queues.**

<table>
<thead>
<tr>
<th>Invocation Number</th>
<th>Accumulated Number of Messages in <em>ap-high</em> queue</th>
<th>Accumulated Number of Messages in <em>ap-low</em> queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>152</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>186</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>216</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>248</td>
</tr>
</tbody>
</table>

**Table 5. Size comparison between the original and the modified SOS.**

<table>
<thead>
<tr>
<th>Module Sizes (Bytes)</th>
<th>Program</th>
<th>Data</th>
<th>Surge</th>
<th>Flash Allocator</th>
<th>Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original SOS</td>
<td>38940</td>
<td>2829</td>
<td>688</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Modified SOS</td>
<td>40056</td>
<td>2839</td>
<td>736</td>
<td>318</td>
<td>478</td>
</tr>
</tbody>
</table>

---
Table 6. Time for insert, hot-swap a null application module and for hot-swap a null kernel module (clock cycle).

<table>
<thead>
<tr>
<th></th>
<th>Application Module</th>
<th>Kernel Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check the Module List</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>Register the New Module</td>
<td>47</td>
<td>45</td>
</tr>
<tr>
<td>Remove Old Module</td>
<td>53</td>
<td>54</td>
</tr>
<tr>
<td>Update Jump Table</td>
<td></td>
<td>1202</td>
</tr>
</tbody>
</table>

Table 7. State transfer overhead (clock cycles).

<table>
<thead>
<tr>
<th></th>
<th>State Packing</th>
<th>Message Transfer</th>
<th>State Unpacking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Avg</td>
<td>Max</td>
</tr>
<tr>
<td>Check the Module List</td>
<td>6</td>
<td>6.6</td>
<td>7</td>
</tr>
<tr>
<td>Register the New Module</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove Old Module</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

stages of hot-swapping. For comparison, we also show the times spent in the two stages of inserting a null application module.

Note that the time shown in Fig. 11 does not include the overhead of state transfer. To obtain the overhead of state transfer, we upgraded the surge module with a new one that stored the state in a different data structure, and measured the time for packing/unpacking the state and delivering the hot-swap messages. We performed twenty hot-swapping operations and the results are shown in Table 7. Compared with the results shown in Fig. 11, message delivery dominates the overhead since it involves queuing latency.

In the next experiment, we measure the overhead of the converter in supporting interface changes. We used a null converter and performed one hundred times of module invocations and module returns. The experimental results show that the indirection invocation and return caused by the converter has an average of 3.1 clock cycles. Compared with the total running time of a module, the overhead of indirection is negligible.

Finally, we compare the time of inserting a module under offloaded linking and in-situ dynamic linking. The in-situ dynamic linking scheme directly uses the subscription procedure in the original SOS. Table 8 shows the experimental results. From Table 8, the offloaded linking scheme not only reduces the insertion time, but also makes the download time a constant. For example, compared with the in-situ dynamic linking scheme, when the number of subscribed functions is one, the offloaded linking scheme takes only
Table 8. Time of inserting a module with and without offloaded linking (clock cycle).

<table>
<thead>
<tr>
<th>Number of subscribed functions</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-situ linking</td>
<td>14.9</td>
<td>24.6</td>
<td>33</td>
</tr>
<tr>
<td>Offloaded linking</td>
<td>2.9</td>
<td>2.9</td>
<td>2.9</td>
</tr>
</tbody>
</table>

Fig. 12. System call invocation overhead.

one fifth of time to download a module. Furthermore, the download time of a module increases with the number of subscribed functions in the in-situ dynamic linking scheme. This is because with the increasing number of subscribed functions, more symbols need to be resolved during the linking process. Since in-situ dynamic linking scheme performs linking in each sensor node, the insertion time of a module thus increases with the number of subscribed functions.

5.3 Effects of the Other Improvements on SOS

In this subsection, we measure the performances of our two improvements on the SOS. Fig. 12 presents the average time of a system call invocation, with and without the caching of the addresses of system call handlers. As shown in the figure, caching can reduce the system call invocation overhead by about 15%. This is because the original system call invocation needs two load instructions from flash memory during a total of six instructions. By contrast, if the target address of the system call is cached, flash memory access can be avoided and the two load instructions instead read data from internal SRAM, which is much faster than flash memory. As shown in [7, 24], the time of reading data from flash memory is at least two times that of reading data from SRAM.

Furthermore, we also measure the cost of invalidating cached addresses. As shown in section 4.1, the kernel sends a `CACHE_INVALID` message to notify application modules to invalidate their cached addresses. Consequently, the cost of invalidating cache is the message delivery time. Note that, as mentioned in section 4.1, the `CACHE_INVALID` message is set to the highest priority and thus has the least amount of queuing delay. We performed twenty experiments and, on average, it takes 24 clock cycles to invalidate the cache. However, the frequency of invalidating cache is quite low. It occurs only when a kernel module that contains functions exported in the jump table is hot-swapped. By contrast, system calls are invoked much more frequently and have a great impact on the performance of module execution.
To demonstrate the improvement on the flash block allocation, we directly use the results of Fig. 10. In addition to show the support of interface changes, Fig. 10 also presents our improvement on the flash block allocation. From Fig. 10 we can see that the erase-counter based allocation approach does allow the blocks to be erased more evenly than the FIFO-based scheme.

5.4 Analytical Analysis of Update Energy and Memory Cost of Offloaded Linking

The energy consumption $E_{\text{in\_situ}}$ of the reprogramming process in in-situ run-time dynamic linking scheme can be modeled with

$$E_{\text{in\_situ}} = E_p + E_s + E_l + E_f$$

where $E_p$ is the energy consumed in receiving a module object file over the radio, $E_s$ is the energy spent in storing the received object file onto the external flash memory, $E_l$ is the energy cost of linking the object file, and $E_f$ is the energy required for storing the linked module image in program memory [21]. To approximate the total energy $E_{\text{in\_situ}}$ during reprogramming, we use a simplified model that the energy of both communication and storing is directly proportional to the size of the new module. Consequently,

$$E_p = R_p U, E_s = R_s (U/h), E_f = R_f (L/h)$$

where $U$ is the size of the new module, which is in the form of an object file, to be transferred, $R_p$ is the required energy to receive a byte over the radio, $R_s$ and $R_f$ are the energy of writing a page to the external flash memory and program memory, $h$ is the size of one memory page, and $L$ is the size of the new module that has been linked and is in the form of an image file. Substituting Eq. (20) into Eq. (19), Eq. (19) can be rewritten as

$$E_{\text{in\_situ}} = R_p U + R_s (U/h) + E_l + R_f (L/h).$$

In our offloaded run-time linking scheme, we offload the linking energy $E_l$ to the server. Furthermore, since the new module has been linked, the energy $E_p$ becomes $R_p L$ and the new module can be directly written into the program memory to eliminate the cost $E_s$. Consequently, the energy consumption $E_{\text{offloaded}}$ during the module reprogramming in our offloaded runtime linking scheme can be approximated as

$$E_{\text{offloaded}} = R_p L + R_f (L/h).$$

Thus, the energy saving by our offloaded linking scheme can be computed as:

$$E_{\text{in\_situ}} - E_{\text{offloaded}} = R_p (U - L) + R_s (U/h) + E_l.$$
In addition to reducing energy consumption, our offloaded linking approach also reduces the memory footprint in that it does not need to store a kernel’s symbol table in each sensor node. For example, the kernel’s symbol table of SOS has 767 entries with each entry consuming 16 bytes (assuming the ELF format), and thus the table occupies 12272 bytes of memory space in total. Storing this table in RAM would extremely reduce the available memory space. On the other hand, if the table is stored in flash memory, the energy $E_t$ would be further increased.

6. CONCLUSIONS AND FUTURE WORK

A sensor network operating system must address the characteristics of sensor network. For example, the ability to dynamically reprogramming a WSN is an important and necessary feature for developing and maintaining WSNs. Furthermore, energy conservation is also important to sensor networks since sensor nodes are usually battery-operated. In this paper, on the basis of the SOS kernel, we firstly support lightweight hot-swapping of application and kernel modules. During hot-swapping, the execution state of the old module can be transferred to the new one so that the latter can pick up where the former left off. In addition, we also extend the previous hot-swap capability to support interface changes. The hot-swapping is lightweight since we offload the job of module-linking to the server side to reduce the energy consumption and memory footprint during hot-swapping. Finally, we also improve the SOS kernel by reducing the overhead of system call invocation and by evenly wearing the flash blocks.

In the current implementation, a module fault can crash the entire sensor node, which may lead to the loss of some information (e.g. routing information) and require efforts (i.e. energy or time) to reconstruct that information. In the future, we will investigate issues of integrating fault isolation and recovery techniques into our hot-swappable sensor operating system so as to make sensor nodes self-recoverable.

REFERENCES


19-30.

**Hsung-Pin Chang** (張軒彬) received the B.S., M.S., and Ph.D. degrees in Computer and Information Science from National Chiao Tung University, Taiwan. Since 2004, he has been an Assistant Professor in Computer Science and Engineering at National Chung Hsing University, Taichung, Taiwan, R.O.C. He was a postdoctoral researcher in National Chiao Tung University in 2002-2003, and an Assistant Professor in Electrical Engineering at National Changhua University of Education, Chaunghua, Taiwan, from 2003 to 2004. His research interests include operating systems, embedded systems, wireless sensor networks and network protocols.

**Yu-Chieh Lin** (林郁傑) received the B.S. degree in Department of Computer Science and Information Engineering, Tam-Kang University, Taipei, Taiwan, in 2005, and M.S. degree in Department of Computer Science and Engineering, National Chung Hsing University, Taichung, Taiwan, in 2007. He is currently a member of the ehouserkeeper project of Superior e-Government program in the Research, Development and Evaluation Commission. His research interests include network protocols, operating system, and wireless sensor networks.

**Da-Wei Chang** (張大緯) received his B.S., M.S., and Ph.D. degrees in Computer and Information Science from National Chiao Tung University, Hsinchu, Taiwan. He has been a postdoctoral researcher in National Chiao Tung University in 2002-2005, and an Assistant Professor in Electrical Engineering at National Sun Yat-Sen University, Kaohsiung, Taiwan, in 2006. He is currently an Assistant Professor in Computer Science and Information Engineering at National Cheng Kung University, Tainan, Taiwan. His research interests include operating systems, file and storage systems, and embedded systems. He is a member of the ACM and IEEE.