OTL: On-Demand Thread Stack Allocation Scheme for Real-Time Sensor Operating Systems^{*}

Sangho Yi¹, Seungwoo Lee¹, Yookun Cho¹, and Jiman Hong^{2,**}

¹ System Software Research Laboratory
School of Computer Science and Engineering, Seoul National University
Tel.: +82-2-872-7431, Fax.: +82-2-875-7726

{shyi,solee,cho}@ssrnet.snu.ac.kr

² School of Computing, Soongsil University
jiman@ssu.ac.kr

Abstract. In wireless sensor networks, each sensor node has severe resource constraints in terms of energy, computing device, and memory space. Especially, the memory space of the platform hardware is much smaller than that of the other computing systems. In this paper, we propose a OTL, which is an on-demand thread stack allocation scheme for MMU-less real-time sensor operating systems. The OTL enables to adaptively adjust the stack size by allocating stack frame based on the amount of each function's stack usage. The amount of the function's stack usage is checked at compile-time, and the adaptive adjustment of the stack occurs at run-time. Our experimental results show that the OTL significantly minimizes the spatial overhead of the threads' stacks with tolerable time overhead compared with fixed stack allocation mechanism of the existing sensor operating systems.

1 Introduction

Many kinds of existing operating systems can be categorized into two classes according to how they operate and design kernel architecture. Historically, name of the two rough classes are defined as follows: event-driven(or message-oriented), and multi-threaded(or procedure-oriented). For the past several decades, many discussions have been made on these two canonical models of the operating systems[1,2,3,4,5,6]. In 1978, Lauer and Needham attempted to finish the discussion on the two models by showing duality between event-driven and multi-threaded operating systems[2]. As a result, they concluded that the two canonical models are "duals" in terms of characteristics, program operation, and system performance.

The above conclusions are correct theoretically. In the real implementations, however, the two models show significant difference on performance and efficiency of systems and applications. There are pros and cons of the event-driven

^{*} This Research was supported by the Soongsil University Research Fund.

^{**} Corresponding author.

and the multi-threaded system models on designing a sensor operating system. The event-driven system design gives us smaller context switching latency and memory usage based on single stack management. However, response time and preemptivity are much poorer than that of the multi-threaded systems[3]. The multi-threaded design enables preemption, but necessary memory space for the thread's stack is larger than that of the event-driven[5].

In this paper, we propose a OTL, which is an on-demand thread stack allocation scheme for MMU-less real-time sensor operating systems. The OTL enables to adaptively adjust stack size by allocating or releasing additional stack frame based on the amount of each function's stack usage. The information is calculated at compile-time, and adaptive adjustment of the stack occurs at run-time. Our experimental results show that OTL significantly minimizes the spatial overhead of the threads' stacks with tolerable time overhead compared with fixed stack allocation mechanism.

The rest of this paper is summarized as follows. In Section 2, we present some related works on sensor operating systems. Section 3 describes characteristics of the wireless sensor nodes, and explains the design and implementation of OTL in detail. Section 4 presents and evaluates performance of OTL compared with fixed stack allocation mechanism on the existing multi-threaded sensor operating systems. Finally, some conclusions and future work are presented in Section 5.

2 Related Works

In this section, we briefly introduce previous works relating to our proposing mechanism. Many kinds of research efforts[3,4,5,6,7,8] have been done on designing and implementing sensor operating systems and stack allocation mechanisms.

In [4], Adya et al. presented a "sweet spot" hybrid task management model of the two existing models(event-driven and multi-threaded). It can be implemented based on continuation and stack ripping mechanism, and the "sweet spot" enables cooperative task processing without manual stack management.

In [3], Ousterhout argued disadvantages of the multi-threaded task model with many usages and examples of the two existing models. As a result, the multi-threaded task model is better than event-driven if a target system is susceptible to the performance of response time or preemptivity of the task execution.

In [5], Behren et al. compared performance of the multi-threaded task model with event-driven. In addition, they insisted that the performance of the multi-threaded operating systems can be improved by compiler-assisted techniques.

In [9], Torgerson proposed an automatic thread stack management scheme on the MANTIS operating system based on compile-time calculation of the upper-bound of the thread stack size. The proposed scheme mitigates trade-off relation between stack overflow and stack space efficiency on the multi-threaded task model. However, this scheme cannot remove the space inefficiency which caused by static allocation of each thread stack area.

In [6], Gustafsson briefly introduced concepts of the stackless implementation of C/C++ programs to minimize memory usage. For example, C compiler can

predict maximum required stack space, the compiler would allocate space for function arguments and local variables on the heap instead of a stack.

In [10], Dunkels et al. proposed Protothread, which is a stackless threads in C programming to minimize memory usage on implementing multi-threaded applications for wireless sensor networks. However, the Protothread is not a method of multi-threaded programming, but a programming sugar of event-driven programming. In addition, it does not support automatic state management and preemption between multi-threaded tasks.

3 OTL: On-Demand Thread Stack Allocation Scheme

In this section, we describe requirements for wireless sensor networks including platform hardware of the sensor nodes with some existing constraints. Then, we present design and implementation of OTL in detail.

3.1 Requirements for Wireless Sensor Networks

In wireless sensor networks, it is very important to minimize production cost to enhance networks efficiency. Such restriction limits computing power, memory space, and batteries of each sensor node. For example, Berkeley's MICA motes have only 8-bit processor, 4 KB memory space, and 2xAA batteries[11]. Thus, developers of the sensor operating systems must consider these severe constraints on the sensor hardware platform. In addition, sensor operating systems need to take the multi-threaded task model to support real-time, concurrent processing of the various tasks such as communication, sensing, analog-to-digital conversion, data aggregation, and etc. For this reason, operating systems that run on the sensor nodes need a space-efficient stack allocation mechanism.

The following are the essential requirements when designing a new stack allocation mechanism for the multi-threaded task model.

- ◇ **Minimization of allocated stack space.** In general purpose operating systems, each thread has a relatively large stack area(4 KB in Linux), but the memory space of a sensor node is very limited(about 1~4 KB).
- ◇ **Minimization of time overhead.** Many existing software virtual memory or MMU mechanisms have large overhead on execution time, but computing power of a sensor node is much smaller than the other computing systems.

3.2 Design and Implementation of OTL

The major objective of the OTL is minimizing stack usage on multi-threaded operating systems. In order to efficiently utilize memory usage, it provides adaptive and on-demand stack allocation for each thread based on the calculation method of the amount of stack used for each function call.

Table 1 compares OTL with fixed stack allocation mechanism. In fixed(or static) stack allocation mechanism, thread stack allocation is performed only

Table 1. Comparison between fixed stack allocation mechanism and OTL

Feature	Fixed stack allocation	OTL
Allocation point	on creating a thread	on calling a function
Freeing point	on exiting a thread	on returning from a function
Stack frame size	fixed(or static)	adaptive(or dynamic)
Timing overhead	single allocation	multiple allocation

when a new thread is created, and the allocated space will be released only when the threads exits. In this mechanism, the allocated stack space is belong to a thread until the thread exits. If a thread stack’s upper bound is significantly larger than average stack usage, the system will be inefficiently operated in terms of memory space utilization. On the otherhand, OTL adaptively allocates the necessary thread stack space on demand. Therefore, in OTL, the space efficiency will be significantly improved, but the several times of stack allocation may increase total execution time.

Fig. 1. Building process of sensor operating systems when using OTL

Figure 1 shows the building process of sensor operating systems and a pplications when using OTL. First of all, "OTL Library" is used for allocating and releasing thread stack space at thread’s run-time. Second, the "Top-half Analyzer" is used for calculating the stack space usage of each function. AVR-GCC[12] is a C compiler for the AVR processor. The AVR-GCC with "-S" compiles a C source file(*.c), and it gives an assembly code file(*.s) as a output. The "Top-half Analyzer" parses and analyzes the output assembly code, and calculates the amount of stack space used for each function. The calculated results are stored at the file "otl.map". The third one is the "Bottom-half Analyzer". It is used for assigning the amount of stack space for each function call. The "Bottom-half Analyzer" parses the C source file(*.c), and then for each C function call, it finds the stack space usage of each function call from the file "otl.map". Then, the allocating and releasing OTL library routine for each function call is inserted to the before and after the call respectively. The modified C source files(*.c) is compiled. Finally, the kernel image of the operating system is built for wireless sensor nodes.

Figure 2 shows comparison between existing fixed stack allocation mechanism with the proposed scheme. This is an example of the same "pthread-net4: sensor110" application on the Nano-Qplus[8] sensor operating system. In case of

Fig. 2. An example of stack space of fixed stack allocation mechanism and the OTL

the fixed stack allocation, the stack space is managed as a single frame for each thread. However, in case of the OTL, the stack space is similar to the linked list structure, and the only necessary stack space is allocated for the threads. In the OTL, the maximum available number of threads are not affected by the fixed stack size, but the actual stack space used on the threads. Therefore, if the operating system uses the OTL, then the system can support more threads than existing fixed stack allocation mechanism.

Table 2. List of functions and variables used in this paper

Notation	Description
<i>allocMemory</i> (m_{size})	returning address of an allocated m_{size} memory block
<i>releaseMemory</i> (m_{addr})	releasing a memory block of an address m_{addr}
<i>saveSp</i> (m_{addr})	saving stack pointer to an address m_{addr}
<i>loadSp</i> (m_{addr})	loading stack pointer from an address m_{addr}
<i>changeSp</i> (m_{value})	changing stack pointer to a value m_{value}
<i>func().stackAddress</i>	stack frame address of a <i>func()</i>
<i>func().stackNecessary</i>	necessary stack frame size of a <i>func()</i>
<i>func().arguments</i>	arguments of a <i>func()</i>

We used the above notations to present algorithms of the proposed OTL. Some functions and variables that are used in protocol descriptions are defined in Table 2. The following shows the main algorithm of the proposed scheme.

In Algorithm 1, necessary stack allocation and saving stack pointer are performed before the function call. After that, the OTL moves the thread's stack pointer to the newly allocated stack area. Then, it pushes C function's arguments and returning address, sets the local variables, and runs the function's internal

Algorithm 1. Dynamic stack management algorithm of OTL

```

— when a thread calls a func(), do the following: —
func().stackAddress ← allocMemory(func().stackNecessary)
saveSp(func().stackAddress)
changeSp(func().stackAddress + func().stackNecessary - 1)
push func().arguments
call func()
loadSp(func().stackAddress)
releaseMemory(func().stackAddress)

```

routine. The OTL restores the thread's stack pointer from the saved data, and finally, the allocated stack space is released from the thread.

4 Performance Evaluation

In our performance evaluation, we compared the actual performance of the above mechanisms by making experiments on the real sensor platform. We used Octa-comm's Nano-24 wireless sensor platform[13], which is similar to the Berkeley's MICAZ sensor platform. The proposed scheme is implemented on the Nano-Qplus(ver.1.6.0e). This operating system supports a real-time memory allocation scheme with preemptive task scheduler.

Fig. 3. Organization of the pthread-net4 sensor application

We used two kind of sensor applications which presented in the above, and the average memory usage and the average execution time are shown in Fig. 4 and 5, respectively.

Figure 4 shows performance of the fixed allocation mechanism and the proposed scheme in terms of both time and space. In case of OTL, each function call needs additional 4 bytes to save previous stack pointer register. But it is very negligible space overhead, since it can save significant stack space compared with the fixed allocation mechanism.

Figure 5 shows the execution time when using the proposed scheme or not. The proposed scheme has a small timing overhead(about 10 percent) compared with fixed stack allocation mechanism. However, it can improve significantly the space-efficiency of the stack usage.

Fig. 4. Run-time stack space usage of the sensor applications

Fig. 5. Execution time of the sensor applications

Based on these results, we can convince that the proposed mechanisms significantly improves the space-efficiency of the threads' stack space allocation compared with the existing fixed stack allocation mechanism.

5 Conclusions

Wireless sensor networks are composed of tiny sensor nodes that usually have severe resource constraints in terms of energy, computing device and memory space. Sensor operating systems that run on tiny sensor nodes are the key to the performance of the wireless sensor networks. In order to process many tasks efficiently with the sensor nodes, the sensor operating systems need a space-efficient multi-threaded task model. In this paper, we propose a OTL, which is an on-demand thread stack allocation scheme for MMU-less real-time sensor operating systems. The OTL enables to adaptively adjust the stack size by allocating or releasing additional stack frame based on the amount of each function's stack usage information. The information of the function's stack usage is checked at compile-time, and the adaptive adjustment of the stack occurs at run-time. Our experimental results showed that the OTL can significantly minimize the spatial overhead of the threads' stacks with tolerable time overhead compared with fixed stack allocation mechanism on the existing multi-threaded sensor operating systems.

References

1. Hoare, C.A.R.: Monitors: An operating system structuring concept. *Communications of the ACM* **17** (1977) 549–557
2. Lauer, H.C., Needham, R.M.: On the duality of operating system structures. In: *Second International Symposium on Operating Systems, IRIA*. (1978)
3. Ousterhout, J.K.: Why threads are a bad idea (for most purposes). Presentation given at the 1996 Usenix Annual Technical Conference (1996)
4. Adya, A., Howell, J., Theimer, M., Bolosky, W.J., Douceur, J.R.: Cooperative task management without manual stack management. In: *Proceedings of the 2002 USENIX Annual Technical Conference*. (2002)
5. von Behren, R., Condit, J., Brewer, E.: Why events are a bad idea (for high-concurrency servers). In: *HotOS IX: The 9th Workshop on Hot Topic in Operating Systems*. (2003) 19–24
6. Gustafsson, A.: Threads without the pain. *ACM Queue* **3** (2005) 34–41
7. Yannakopoulos, J., Bilas, A.: Cormos: A communication-oriented runtime system for sensor networks. In: *Proceedings of the 2nd IEEE European Workshop on Wireless Sensor Networks (EWSN '05)*. (2005)
8. Lee, K., Shin, Y., Choi, H., Park, S.: A design of sensor network system based on scalable and reconfigurable nano-os platform. In: *IT-Soc International Conference*. (2004)
9. Torgerson, A.: Automatic thread stack management for resource-constrained sensor operating systems. (2005)
10. Dunkels, A., Schmidt, O., Voigt, T.: Using protothreads for sensor node programming. In: *Proceedings of the REALWSN'05 Workshop on Real-World Wireless Sensor Networks*. (2005)
11. Crossbow: <http://www.xbow.com/>. (website)
12. AVR-GCC: <http://www.avrfreaks.net/AVRGCC/>. (website)
13. Octacomm: <http://www.octacomm.net/>. (website)