Hardware Support for a Wireless Sensor Network Virtual Machine *

Hitoshi Oi†
Department of Computer Science
The University of Aizu
Aizu-Wakamatsu, JAPAN
hitoshi@u-aizu.ac.jp

ABSTRACT

Virtual Machines (VMs) have been proposed as an efficient programming model for Wireless Sensor Network (WSN) devices. However, the processing overhead required for VM execution has a significant impact on the power consumption and battery lifetime of these devices. This paper describes the design of a hardware accelerator for Maté, a VM running on TinyOS. While faster execution speed is not important in the WSN applications, reduction in the number of clock cycles for performing the same task results in the lower duty cycle, which in turn saves the power consumption. With the dedicated data path, it is expected that the numbers of clock cycles can be reduced by one to two orders of magnitude compared to the software implementation of Maté.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems; C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms

Design

Keywords

Wireless Sensor Network, Virtual Machine, Hardware Support, Low Power Design

1. INTRODUCTION

Wireless Sensor Networks (WSNs) combine processing, sensing and communications into tiny devices (motes) that can be deployed over wide-areas to provide long-term monitoring [1]. It is expected that thousands of low-cost motes will be deployed over wide-areas to provide monitoring of conditions and/or activity. Potential applications include traffic monitoring, precision agriculture, habitat monitoring, building security, waste control and seismic sensing.

One of the key research challenges in the area of WSN is in providing an efficient programming model for such systems. Uniquely, the programming model must allow for a heterogeneous mix of processors, motes with different sensing capabilities, over-the-network software update and low power consumption. One attractive programming model for such systems is the use of Virtual Machines (VMs) executing on the mote processors [2].

VMs allow for a single programming model which will operate across a heterogeneous mix of processors. Allowance may be made for the various capabilities of motes by providing profiles and abstractions of mote capabilities in the programming model. However, execution of software using a VM incurs an overhead in execution time relative to execution of functionally equivalent native code. Typically, the overhead is of the range 1-33 times [3]. Since the delay between sensor or timer events is usually long compared to the processing time, increases in execution time are not an issue for WSN applications. However, the increase in power consumption due to the execution of an increased number of instructions is a limiting factor.

In field experiments, it has been found that current WSN motes have a battery life of just a few days when running native code [4]. Studies across a range of benchmark applications show that the processor consume between 28% and 86% of total mote energy [4]. Clearly, if applications are implemented using software running on VMs, a further reduction in battery life can be expected. In contrast, the target battery life for the use WSN motes in real-world applications is 12-18 months.

TinyOS [5] is currently the de facto operating system for sensor network motes. TinyOS was developed for the WSN mote specific requirements of small footprint, management of hardware, support for concurrency, modularity and robustness. Maté was developed by a team at Berkley as a bytecode interpreter that runs on TinyOS [3].

This paper describes work carried out with the goal of developing a low power Maté compliant VM suitable for WSN motes. Previously, we have pointed out that two major sources of execution overhead in Maté are synchronization and stack-based operations [6]. In this paper, we describe the design of a hardware accelerator that reduces the execution overhead of Maté and compare the execution overhead of Maté in the original software implementation and the proposed hardware-assisted implementation.

The paper is structured as follows. Section 2 provides an overview of related work in the area. Section 3 describes the design of the proposed hardware accelerator for the WSN VM. In Section 4, the

*An earlier version of this work was done under the collaboration with Chris Bleakley and his group at the University College Dublin, Ireland.

†This work is supported in part by grant from the University of Aizu Competitive Research Funding.

Mobilware'08, February 12-15, 2008, Innsbruck, Austria

effectiveness of the proposed design in reducing the power consumption of the WSN motes running VM applications are presented. Section 5 provides conclusions and future work.

2. RELATED WORK

An overview of middleware approaches for WSNs is provided in [2]. Of these various approaches, VMs have a number of advantages for implementation of WSN systems. They allow the programmer to write-once and execute many times across a range of heterogeneous processors. The modularity of VM code allows for concise bytecode. This reduces memory footprint and RF power consumption when dynamically updating applications via the network [3]. VMs intrinsically provide security and synchronization models which simplify the programming task. The VMs customized for WSN applications include Maté [3], MagnetOS [7], VM* [8] and SwissQM [9].

Maté is a bytecode interpreter which runs on top of TinyOS. TinyOS uses a component based software architecture. Each component can call or respond to a command; flag or process an event; or execute a task. Processing is based on interrupts which are managed via a simple FIFO scheduler implemented in software. Maté is a single TinyOS component that interfaces to various system components such as sensors, the network and non-volatile storage. Most instructions operate on an operand stack. A return address stack is provided for subroutine calls. Control flow instructions and instructions with immediate operands are available. There are three types of instructions: basic, s-class and x-class. Basic instructions include arithmetic and LED control. S-class instructions access in-memory structures for messaging. X-class instructions are push constant and branch on less than or equal. Eight instructions are set aside for users to define. Three operand types are supported - values, sensor readings and messages.

Maté uses a high level programming interface which allows for very short application programs. Code is split into capsules of 24 instructions which can be transmitted through the network. Capsules contain code, identification and versioning information. Subroutine capsules allow for more complex programs to be constructed across multiple capsules. Maté starts execution in response to an event, e.g. a timer wake up. Control then jumps to the start of the corresponding packet and completes with the halt instruction. The first version of Maté lacks flexibility and support for higher level languages as pointed out in the literature [2]. The specification of Maté has been upgraded as an application specific virtual machine (ASVM) for improved execution efficiency and customizability [10, 11].

VM* provides a richer service interface than Maté, allowing for easier programming. It uses software synthesis to tailor and scale the system software to each application. VM* also allows fine-grain updating of the VM itself, whereas Maté only allows updates to VM applications. This allows for greater flexibility and can reduce the energy of code dynamic update via the network. VM* is based on JVM but includes a number of innovations to reduce bytecode size.

SwissQM is another VM architecture for the WSN [9]. It is designed to enhance the data acquisition capability of the sensor networks. In the application level, SwissQM is designed to process SQL-like queries. The instruction set architecture of the VM is stack-based and the bytecodes include aggregation operations.

SOS is an operating system for the sensor network node that consists of a common kernel and application modules [13]. Application modules can be dynamically loaded and unloaded to a running node with a small system interruption. The authors compared SOS with Maté and TinyOS in their CPU overhead and power consumption to show SOS's higher level of functional flexibility and lower performance overhead.

MagnetOS differs quite significantly from Maté and VM*. It consists of a Single System Image layer which provides a high level abstraction of an entire WSN. The abstraction allows the whole network to appear as a single, unified VM. The system partitions applications into components and dynamically distributes them through the network.

The concept of using hardware accelerators to speed up or reduce the power consumption of VMs has been applied to execution of Java bytecode for some time [12]. There are significant differences between the requirements for a JVM on an embedded processor and a WSN VM. In particular, WSN VMs require efficient abstractions for sensing devices, must operate on devices with limited memory and processing resources, must support data aggregation techniques across multiple nodes and must be power aware when processing and communicating data. On the other hands, a JVM is often implemented on portable devices such as mobile phones and running interactive applications. Therefore, when compared to WSN VMs, embedded JVMs are more concerned on their performance. Hence, it is not expected that all hardware acceleration concepts from the Java arena will work well for WSN processors.

3. HARDWARE ACCELERATOR FOR VM EXECUTION

In this section, we describe the architecture of the WSN mote with proposed hardware accelerator and how it reduces the execution overhead of the VM. The design of our VM is based on Maté and the hardware accelerator consists of two modules: operand stack module and synchronization module. These two modules implement the operations of Maté that are identified as the major sources of the overhead for VM execution [6, 10]. Figure 1 shows the architecture of the wireless sensor network processor with the proposed hardware accelerator for virtual machine. The hardware accelerator is attached to the data bus of the microcontroller on which the bytecode interpreter of the virtual machine is executed.

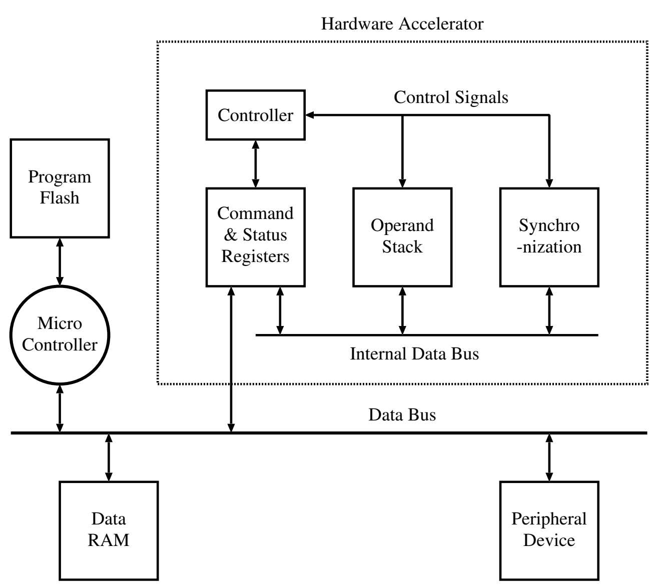

Figure 1: Proposed WSN Processor Architecture

The registers in Table 1 are used to initiate operand stack and synchronization operations and to receive the return status and value. These registers are mapped onto the data memory address space and the microcontroller writes command and parameters and read results using load/store instructions. A possible optimization is to swap part of general purpose registers with these these registers when the processor is running the VM. In this way, any instructions having registers as their operands can directly access the hard accelerator registers (i. e. can omit load/store instructions).

A context corresponds to a process in other operation systems and each context has its own operand stack and stack pointer. A context locks and releases shared variable(s). The Context register specifies the context for which an operand stack/synchronization operation is performed. Command register specifies one of operations to be performed that are described in the following subsections. The results of the operation performed, such as a success in locking a variable or a stack overflow, are reported by the Status register. The Data register is used to store the integer data or the address of other types of data (such as message buffer) to be pushed or popped. It is also used for specifying the starting address of the bytecode to be scanned. The data type is specified in the Type register. It is also used for specifying the length of the bytecode to be scanned in `AnalyzeVars` (Section 3.2).

Name	**Length**	**Mode**	**Descriptions**
Context	1	W	Context of the stack
Command	1	W	Stack/Synch. operation
Status	1	R	Return status
Data	2	RW	Stack Operand Code Address
Type	1	RW	Operand type.

Table 1: Hardware Accelerator Registers

3.1 Operand Stack Module

For the purpose of explanation, we use the VM configuration parameters, such as number of contexts, of the BombillaMica (an implementation of Maté included in TinyOS 1.1.15 distribution). Each entry in the operand stack consists of the Type (1 Byte) and ValBuf (2 Bytes) fields. In the BombillaMica implementation, there are 13 valid contexts defined. Each context has own operand stack which consists of a 8-entry stack and a stack pointer. A possible and preferred implementation is to make the operand stack as a 3-byte wide register file indexed by the concatenation of the Context register and the stack pointer.

	Cycles		
Command	SW	HW	Description
pushValue	93	14	Push 16-bit Int
pushReading	93	16	Push sensor reading
pushBuffer	93	14	Push message buffer
pushOperand	91	14	Push untyped operand
popOperand	65	14	Pop TOS operand

Table 2: Stack Operations

Table 2 shows the operand stack operations that are performed by the hardware accelerator. PushValue, pushReading, pushBuffer and pushOperand push a 16-bit integer, a value read from a sensor, the address of a message buffer and the address of an untyped value. Unlike push operations, there is only a single pop operation, popOperand. Therefore, the bytecode that uses the popped operand must check the type using the Type register. Push and pop operations can cause stack overflow and underflow errors. These errors are reported by the Status register. A possible encoding of the Status register is to use the MSB to indicate the occurrence of an error and the LSBs to indicate the number of entries in the stack (the latter is only useful for the debugging purpose).

The second and third columns (labeled SW and HW, respectively) in the table show the numbers of clock cycles when each operation is performed by the software and the hardware. The number of clock cycles for the software case is obtained by simulating the stack operations using Avrora [14]. The number of clock cycles for the hardware is calculated as follows. First, the microcontroller writes necessary parameters (if any) and the command to the hardware accelerator registers. Theses writes take two clock cycles per byte. After one clock cycle, the microcontroller reads the Status register and checks if the operation has been performed normally by executing a conditional branch instruction on the return status value. It is assumed that these post-operation steps take four clock cycles. The instruction set architecture of the microcontroller is based on [15].

```
   st Z+, r20          # Write Context
   st Z+, r21          # Lower 8−bit of data
   st Z+, r22          # Higher 8−bit of data
   ldi r23, pushValue
   st Z+, r23          # Command (pushValue)
   ld r24, Z           # Load return status
   andi r24, 0x80      # Return state is in MSB
   brne OK:            # check the return status
   call error
OK:
```

Figure 2: Sample Invocation of `pushValue`

A sample invocation of `pushValue` operation using the hardware accelerator is shown in Figure 2. It is assumed that `Z` register has the address of the Context register and registers `r20` to `r22` have already been loaded with the context and the value to be pushed. It should be noticed that some bytecodes perform multiple stack operations per bytecode and some redundant steps can be removed. For example, an `add` performs two `popOperand` and one `pushValue` operations. In the hardware implementation, we only need to set the `Context` register once for such bytecodes. On the other hand, each stack operation is invoked as a separate function call in the software implementation. Therefore, redundant operations in a single bytecode cannot be eliminated.

3.2 Synchronization Module

The objective of the Synchronization module is to avoid the race condition between contexts and for that purpose, it maintains a set of data structures for each context most of which are bit vectors representing shared variables (Table 3).

Each context runs a program (called `handler`) which is identified by a unique `handlerID`. `currentHandler` is an array that maps a `contextID` into a `handlerID` and represents the handler running on the context. `acquireSet`, `heldSet`, and `releaseSet` are arrays of bit vectors indexed by the contextID which is given by the Context register in Table 1. They represent shared variables to acquire, being held and to be released by the context. `usedVars` is an array of bit vectors indexed by

Data Structure	Description
CurrentHandler	Map context to handler
acquireSet	Vars to acquire lock by context
heldSet	Vars being held by context
releaseSet	Vars to release by context
userVars	Vars used in handler
locked	Vars locked
holder	Context locking vars
byteLength	Length of bytecode
lockNum	Var used in bytecode

Table 3: Data Structures in Synch. Module

the handlerID. Each bit vector represents the shared variables used in the handler. `holder` maps a shared variable number into a `contextID` and represtns the context holding the variable. `locked` is a bit vector representing which variables are locked: $\texttt{locked}_i = 1$ means that variable `i` is locked by some existing context. These data structures are constructed by `AnalyzeVars` which scans a newly received program capsule (described later). `byteLength` and `lockNum` are tables that map bytecode into its number of bytes and the variable used in the bytecode, respectively.

Command	Cycles SW	Cycles HW	Description
lock	62	10	Lock a variable
unlock	64	10	Unlock a variable
isLocked	23	8	If a var is locked ?
isHeldBy	29	10	Am I locking a var ?
AnalyzeVars	1.3×10^4	270	Find vars in a capsule

Table 4: Synchronization Operations

Table 4 shows the operations performed in the Synchronization module. Again, the columns labeled as SW and HW represent the numbers of clock cycles for the operation when implemented in software and hardware, respectively. In `AnalyzeVars`, it is assumed that the code length is 128 bytes and the hardware accelerator takes two cycles to access a byte in the memory.

First four operations are used in the bytecodes and also in the scheduling of the tasks. `lock` and `unlock` lock/unlock a variable. `isHeldBy` and `isLocked` check if a variable is held by the current context or somebody else. When a new program capsule is received, its bytecodes are scanned and shared variables used in the capsule are recorded in the corresponding entry of `usedVars`. This operation, `AnalyzeVars`, is performed by the synchronization module by providing the starting address of the program capsule stored in the data RAM, its length and the assigned handlerID using the Data, Status and Context registers, respectively.

4. DISCUSSION

In [3], Maté bytecodes are classified into four groups based on their relative costs against the native code implementations. Sample bytecodes in this classification and their relative costs are shown in Table 5. Please note that it is not appropriate to directly compare the numbers of clock cycles in Table 5 and those in Tables 2 and 4. These two sets of figures are measured by different methodologies in different environments (especially compiler versions and options which should affect the resulting codes significantly). However, it is also true that we run a stack-based virtual machine on top of a register-based microprocessor. Operations on stack and other data structures require handling memory pointers which are 16-bit long while the data path of the processor is 8-bit. All these overheads can be reduced by one to two orders of magnitude by means of dedicated hardware modules. Therefore, it is expected that the operations in the Simple group can be executed with similar costs as the native instructions. For operations in other groups, especially the Long Split, the numbers of clock cycles saved by the hardware accelerator should be smaller than those for Simple operations. However, the processor is not necessarily required to be turned on for the entire period of executing these types of operations. Depending on the design and functionality of the peripheral devices, it may be possible to put the processor into the low power mode while the peripheral is doing the necessary operation. In this case, the hardware accelerator can help the processor to setup the parameters for the operation and retrieve the results to/from the operand stack so that the processor can enter the low power mode quickly.

Operation	Clock Cycles Mate	Clock Cycles Native	Cost
Simple: `add`	469	14	33.5:1
Downstream: `rand`	435	45	9.5:1
Quick Split: `sense`	1342	396	3.4:1
Long Split: `sendr`	$685+ \approx 2^4$	$\approx 2^4$	1.03:1

Table 5: Maté Bytecodes vs. Native Code. Reproduced from [3]

Unlike other computing platforms, such as desktop PCs or servers, the objective of the hardware acceleration in the sensor network virtual machine is not to increase the throughput; rather, it is to finish the same amount of job in a shorter period of time so that the system can stay in low power modes and reduce the power consumption. In [13], the authors reported the power consumption of the Mica2 platform in active, idle and sleep modes as 47.1, 29.1 and 0.31mW, respectively. These large differences in the active and other power consumption modes also support the effectiveness of hardware acceleration. As mentioned in Section 1, the virtual machine approach is useful when the program is (relatively) frequently updated but it is executed infrequently. The latter restriction comes from the execution overhead of the VM and the hardware acceleration widen the field where the VM approach is feasible.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we analyzed the execution overhead of the Maté and presented a hardware acceleration approach to reduce the overhead. According to the literature as well as our source code analysis, two major sources of execution overhead are stack and synchronization operations. Handling the operand stack and other data structures having large data sizes by a 8-bit microcontroller takes large numbers of clock cycles. However, most of theses operations can be implemented with combinational circuits and dedicated hardware modules can perform them efficiently. We compared the VM execution overhead for these operations in terms of number of clock cycles when they were implemented in the software and hardware.

Bytecode execution has a large overhead but it is only a part of a WSN processor execution. To estimate the effectiveness of the approach presented in this paper more accurately, a complete model of the proposed architecture is necessary so that the effect of native code execution and peripheral devices can be taken into account. While our work is based on Maté, a virtual machine implemented on TinyOS, however, stack and synchronization operations are commonly seen in other sensor network VM approaches [2, 7, 8, 9]. Therefore, the ideas presented in this paper should be applicable to these VM approaches at a different degree.

Our plan is to develop a detailed model of the hardware accelerator using a hardware description language with the power consumption behavior. This model is to be incorporated into a WSN processor simulator (such as [14]) and to be used for the evaluation of total power consumption. In the course of designing such a detailed hardware model, we should also investigate the feasibility of implementation methods (such as ASIC, FPGA) and the robustness of the hardware supported VM against the buggy and malicious codes.

Wireless sensor networks are used in various ways and there do not seem to be standard benchmark programs existing in this area. To find out appropriate and realistic applications to evaluate the VM approach in the WSN is another important task for us.

6. REFERENCES

[1] D. Culler, D. Estrim and M. Srivastava, "Overview of sensor networks", *IEEE Computer*, vol. 37, no. 8, pp. 41–49, August 2004.

[2] S. Hadim and N. Mohamed, "Middleware challenges and approaches for wireless sensor networks", *IEEE Distributed Systems Online*, 2006.

[3] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks", in *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 85–95, San Jose, CA, USA, 2002.

[4] V. Shnayder, M. Hempstead, B. Chen, G. W. Allen and M. Welsh, "Simulating the power consumption of large-scale sensor network applications", *ACM Conf. Embedded Sensor Systems*, pp. 188–200, 2005.

[5] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler and K. S. J. Pister, "System architecture directions for networked sensors", in *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 93–104, Boston, MA, USA, Nov. 2000.

[6] Hitoshi Oi and C.J. Bleakley, "Towards a Low Power Virtual Machine for Wireless Sensor Network Motes", *Proceedings of the Japan-China Joint Workshop on Frontier of Computer Science and Technology (FCST'06)*, pp18–22, 2006.

[7] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou and E. G. Sirer, "On the need for system-level support for ad hoc and sensor networks", Operating Systems Review, vol. 36, pp. 1–5, Apr. 2002.

[8] J. Koshy and R. Pandey. "VM*: Synthesizing Scalable Runtime Environments for Sensor Networks", in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, pp243–254, San Diego, CA.

[9] Rene Muller, Gustavo Alonso and Donald Kossmann, "SwissQM: Next Generation Data Processing in Sensor Networks", in *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR 2007)*, pp1–9, Asilomar, CA, USA, January 7-10, 2007.

[10] P. Levis, D. Gay and D. Culler, "Active Sensor Networks", in *Proceedings of the 2nd USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, May 2005.

[11] P. Levis, D. Gay and D. Culler, "Bridging the Gap: Programming Sensor Networks with Application Specific Virtual Machines", UC Berkeley Tech Report UCB//CSD-04-1343, August 2004.

[12] "Wireless, ARM Product Information", `http://www.jp.arm.com/naviweb/pdf/wireless_flyer%20final01.pdf`.

[13] Chih-Chieh Han et. al, "A Dynamic Operating System for Sensor Nodes", in *Proceedings of the Third International Conference on Mobile Systems, Applications and Services (Mobisys)*, pp163–173, USENIX/ACM , Seattle, WA , June 2005.

[14] Ben Titzer, Daniel Lee and Jens Palsberg, "Avrora: Scalable Sensor Network Simulation with Precise Timing" in *Proceedings of the 4th international symposium on Information processing in sensor networks*, Article No. 67, 2005.

[15] "8-bit AVR Instruction Set", Atmel Corporation, Nov 2005.