Towards a Low Power Virtual Machine for Wireless Sensor Network Motes

Hitoshi Oi
The University of Aizu,
Aizu-Wakamatsu, Japan
hitoshi at u-aizu.ac.jp

C. J. Bleakley
School of Computer Science and Informatics
University College Dublin, Ireland
chris.bleakley at ucd.ie

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 analyses the sources of power consumption in the Maté VM for WSNs. The paper proposes a generalised processor architecture allowing for hardware acceleration of VM execution. The paper proposes a number of hardware accelerators for Maté VM execution and assesses their effectiveness.

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. In common with previous work on embedded Java Virtual Machines (JVMs) [6, 7], the approach taken is to provide hardware support for various aspects of the VM, thus migrating certain components of the VM from software to hardware. Clearly, a naive porting of the entire software VM to hardware would provide an expensive solution in terms of silicon area and cost. Therefore, the approach taken is to analyse the behaviour of the VM and identify frequently used operations, or "hot spots", and provide hardware support for them. Due to its popularity and availability, Maté was selected as the base VM.

This paper provides an analysis of the run-time behaviour of Maté from the point-of-view of power consumption. Based on this, the paper goes on to propose a number of potential optimizations for reducing the power consumption of the VM. Furthermore, the paper proposes a number of modifications to Maté bytecode which could enable lower power implementation. The paper forms the basis of planned future work to implement and measure the effectiveness of these optimizations. To the authors' knowledge, this is the first publication to investigate low power implementation of VMs for WSNs.

The paper is structured as follows. Section 2 provides an overview of related work in the area. Section 3 describes the generalized processor platform architecture used for the work. Section 4 proposes a number of hardware optimizations to reduce Maté power consumption based on the analysis of the design Maté as well as its execution behavior. Section 5 describes 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. So far, three VMs customized for WSN applications have been proposed - Maté [3], MagnetOS [11] and VM* [12].

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 [8, 9].

VM* provides a richer services 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.

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 [10].

There are significant differences between the requirements for a JVM on an embedded processor and an 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 not expected that all hardware acceleration concepts from the Java arena will work well for WSN processors.

The energy consumption of the processor used in the Tmote Sky sensor device was investigated in [13]. The processor is a variant of the Texas Instruments MSP430 16-bit RISC. The authors measured the variation in power consumption between different instructions with the same addressing mode to be less than 11%. In contrast, they found that the power consumption of a single instruction could vary by as much as three times depending on operand addressing mode. Obviously, the mix of instructions required for a given program is determined by the functionality of the program and the efficiency of the compiler. In general, the power consumption of large real-world programs tend of a weighted mean of the power consumption of the individual instructions in the instruction set. Hence, power consumption during program execution does not vary to such a large degree.

For the purposes of this work, we assume that the energy consumption of the processor is linearly proportional to number of instructions to be executed. That is,

$$E_t = N_c E_c \tag{1}$$

where E_t is the total energy consumed in program execution, N_c is the number of cycles required for program execution and E_c is the energy per cycle.

3. Hardware Architecture

The proposed generalized processor platform architecture is shown in Figure 1. As in the conventional WSN architecture, a Harvard architecture processor is utilized for software execution. Typically, WSN motes employ a low power microcontroller such as the Atmel ATmega128L [13]. Hardware interfaces are provided from the processor to a number of peripherals and a Radio Frequency (RF) transceiver. Peripherals are typically sensors connected with standard interfaces such as I2C and UART. The peripheral controllers provide IO register and interrupt interfaces to the processor.

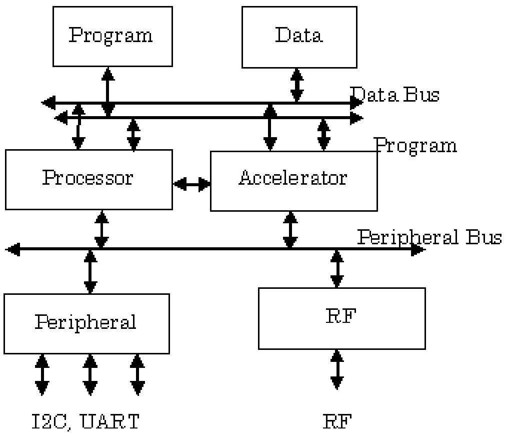

Figure 1. Generalized Processor Platform Architecture

It is proposed that a hardware accelerator module be interconnected in parallel with the processor in order to accelerate and reduce the power consumption of VM execution. The accelerator can communicate directly with the processor via an interrupt mechanism. Data transfer is via the Data RAM. The accelerator may also access the Program RAM in order to read and/or write machine instructions intended for execution by the processor. This allows the accelerator to intervene in the execution of VM applications. When inactive it is proposed that the accelerator be supply gated or implemented using low leakage transistors.

4. VM Power Reduction

In this section, we first analyze the sources of execution overhead in Maté and then we propose several optimizations that utilize the hardware accelerator.

4.1. Capsule Analysis and Installation

In [12], it is reported that the two main sources of Maté's execution overhead are scheduling and the code update mechanism. A capsule is the unit of program transmission. When a mote receives a capsule that has a newer version number, it analyzes and installs the capsule and then reboots. In this analysis, the program code contained in the capsule is checked for its execution properties, such as code length and resource usage, e. g. global variables and stack depth. This is done by scanning the code sequence and accumulating the resource usage information. It is considered that this analysis (or at least part of it) is suitable for implementing inside the hardware accelerator. Instead of running the analysis on the general purpose processor, using an optimized hardware module for this relatively simple task should reduce the energy consumption of overall system. During capsule analysis and installation, it is also possible to optimize the program code for efficient execution. In the next section, we present a number of possible optimizations that can be performed on the program code during analysis and installation.

4.2. Reducing the Execution Overhead of Bytecodes

Like JVM, Maté is a stack-based architecture which pushes and pops operands on the stack frequently. Therefore, for example, it takes four bytecodes to simply add two variables (two pushes, add and a pop). Instruction folding is a technique to combine multiple instructions to increase the execution efficiency [14]. This technique can also be applied to Maté. For example, the bytecode interpreter may find a sequence of `puchc6` (pushing a 6-bit constant) and `add` and handle it as a single composite instruction of `addc6` (adding a 6-bit constant to the top of stack value)[1]. The effect of instruction folding is two-fold: a reduction in the number of executed instructions on the processor and a reduction in the scheduling overhead. The first effect is trivial, so we now explain the second effect briefly.

[1] It should be noted that this `addc6` is assumed to be not included in the instruction set. Otherwise, the compiler must have already used it

In the first version of Maté, each bytecode was executed as a TinyOS task, which caused a significant overhead. This overhead was especially significant for simple bytecodes such as `add` which was slowed down by up to 30 times compared to the same instruction executed on the bare TinyOS system. In Maté Version 2.19a, the users can define the number of bytecodes per task (MATE_CPU_QUANTUM $= 5$ by default) [15]. While instruction folding also increases the effective number of bytecodes executed for a task, it is done in a more adaptive way in terms of synchronization granularity. With the instruction folding scheme, only simple instructions are combined with others. Therefore, the increase in the synchronization granularity should be smaller than simply increasing the number of bytecodes per task.

Another source of execution overhead is that some instructions take operands with multiple data types. For example, `add` can operate on (1) two integers (arithmetic summation), (2) a message buffer and an integer (the integer is appended to the message) or (3) two message buffers (messages concatenated) (Figure 2). Static analysis of the bytecode sequence in the capsule disambiguates the operand types and avoids type retrieval and conditional branches during the execution.

```
if ((arg1→type == MATE_TYPE_INTEGER)
  && (arg2→type == MATE_TYPE_INTEGER)) {
  call Stacks.pushValue(context, arg1→value.var
    + arg2→value.var);
}
else if (arg1→type == MATE_TYPE_BUFFER) {
  if (arg2→type ≠ MATE_TYPE_BUFFER) {
    call Buffer.append(context, arg1→buffer.var, arg2);
  }
  else {
    call Buffer.concatenate(context, arg2→buffer.var,
      arg1→buffer.var);
  }
}
```

Figure 2. An Example of Bytecode with Multiple Operand Type. Extracted from `tinyos-1.1.15/tos/lib/VM/opcodes/OPaddM.nc`.

4.3. Customizability

In Maté Version 2.19a, users can define their own bytecodes to customize their VMs for the target applications. This is feasible because Maté is entirely implemented as software running on a mote. On the other hand, our architecture utilizes hardware accelerators for efficient execution and power reduction. In theory, it is possible to re-design the hardware modules according to the customized instruction set, but in practice it is not desirable since hardware bugs are harder to detect and fix than software ones.

Rather than a fully-customizable instruction set, we consider that a semi-customizable instruction set should suffice for the following reasons. First, while the design of the VM needs to be flexible for various types of applications, a large number of bytecodes are always needed due to the nature of stack-based architecture. These include pushing and pooping of variables and constants. Primitive arithmetic and logical operations are also mandatory. The only flexibility required for these instructions should be the number of bits to specify the variable or the constants. Note that, the instruction folding mentioned in the previous section should be applicable regardless of the encoding of each bytecode (we just need to identify the ones that push or pop a value and that consume it).

Second, we utilize hardware modules to accelerate the program execution. Therefore, it is better to provide a useful set of library hardware modules than to implement the same operations as (custom) bytecodes. The problem is how to provide such a useful hardware modules for each application field of the VM. Assume that the number of library hardware modules is limited to 256 for an instance of the VM (i. e. the top of stack single byte specifies the library to be invoked). If the entire (or most of) the mote in Figure 1 is implemented using System on Chip (SoC) technology and the hardware accelerator module is included in it, we can provide a set of redundant library hardware modules and a table with 256 entries that assigns library specifiers (an integer of 0 to 255) to the library modules provided on the system. If the hardware accelerator module is implemented by a programmable device (such as an FPGA), customization of library hardware modules is straightforward and even dynamic library update is (in theory) possible.

5. Conclusions and Future Work

This paper describes the results of a study of the power consumption of the Maté VM for WSN motes. The main sources of power consumption have been identified and described. The paper proposed a generalized hardware architecture for a mote processor platform including hardware acceleration of the VM. The paper proposed and analysed a number of alternate hardware accelerators to reduce the power consumption of VM execution on the WSN mote. The paper also proposed a number of improvements to Maté which could lead to lower power VM execution.

The paper forms the basis of planned future work which will implement and measure the power consumption of the most promising VM hardware accelerators. Low power implementation of the VM is part of an overall project to develop a low power processor platform for WSN motes. Recently, workload analysis of TinyOS applications has been conducted within our group. We are going to extend this work to the analysis of WSN applications running on Maté. The results of this analysis help us identify the possible targets of hardware acceleration and quantify the power savings as well as execution speed up.

Acknowledgment

This work is supported in part by grants from the University College Dublin (UCD) Seed Fund Programme and the University of Aizu Competitive Research Funding.

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, "On the Design of the Local Variable Cache in a Hardware Translation-Based Java Virtual Machine", in *Proceedings of ACM SIGPLAN/SIGBED 2005 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*, pp87–94, Chicago, IL, June 2005

[7] Hitoshi Oi, "Instruction Folding in a Hardware-Translation Based Java Virtual Machine", in *Proceedings of ACM International Conference on Computing Frontiers*, pp139–145, Ischia, Italy May 2-5, 2006.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] N. D. Lane and A. T. Campbell, "The influence of microprocessor instructions on the energy consumption of wireless sensor networks", in *Proceedings of Third Workshop on Embedded Networked Sensors (EmNets)*, May 2006.

[14] "PicoJava-II Microarchitecture Guide", Sun Microsystems, March 1999.

[15] P. Levis, "Maté Manual", Version 2.19a, `http://www.cs.berkeley.edu/~pal/mate-web/files/mate-manual.pdf`, November 30, 2004.