THE CASE FOR INTERPRETED LANGUAGES IN SENSOR NETWORKS

Leonardo Steinfeld[1] and Luigi Carro[2]

[1] Instituto de Ingeniería Eléctrica, Universidad de la República, Montevideo, Uruguay
leo@fing.edu.uy
[2] Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
luigi.carro@inf.ufrgs.br

Abstract. As sensor networks gain popularity and technology scaling allows further processing in each network node, the programming of these distributed computational structures becomes a serious bottleneck. Interpreted languages adoption may allow a smaller programming effort, and since they show a denser code representation than their directly executed counterpart, interpreted code exhibits smaller power dissipation during over-the-air reprogramming. As technology scales, the processing energy cost tends to reduce more than communication energy, which is bounded by the required irradiated radio power. By allowing the execution of more complex software WSN can be used for more refined applications, like image processing, compression and recognition. Also, interpretation can allow the use of object oriented technology software, allowing high productivity gains. However, the interpretation overhead cost and the extra memory required in Java, for example, argue against interpreted languages adoption in WSN. In this paper we show the design space for interpreted languages, and demonstrate that there is a large application domain where interpretation benefits can be used together with energy efficiency.

Introduction

Wireless sensor networks are a new computing platform that combines computation, sensing, and communication with a physical environment. The sensor node, a new class of networked embedded computer, is characterized by severe resource constraints, especially energy, since they are powered from batteries or harvest energy from the surrounding environment. As technology scales, the capacity of integrated processors increase and new applications can be devised where previously their cost in terms of price and energy were unacceptable. Sensor network applications used to be tightly close to the hardware and after deployment the sensors distribution and function remained unaltered. As a result, current applications are unlikely to change much during network lifetime, since they have not been designed for that at all. However, we envisage a new generation of nodes equipped with a more rich set of sensors, like the artificial retina [1], were costly local processing is mandatory. If more computational power is available at each node, the amount of possible

applications tend to explode, enabling a broader utilization of a WSN, and an increased lifetime thanks to reprogramability of new applications on the same platform.

However, this increasing complexity of applications using wireless sensor networks soon becomes a barrier to the adoption of these networks. The currently available wireless sensor network programming models do not scale well from simple data collection models to collaborative information processing ones. On a different scenario, complex distributed applications have been developed for powerful platforms (such PDA, laptops, etc.), but they are not appropriate for scarce resource platforms like the so-called Berkeley motes or even for more powerful but emerging ones, since the batteries would be drained too soon. New programming models are essential to develop complex distributed applications, and at the same time obtain a decent level of energy-efficiency.

Because of the large amounts of nodes present in a WSN, and since they usually are in an unreachable location, they are expected to run for years unattended. The necessity to perform software changes in deployed wireless sensor network is an important issue that increasingly calls the attention of the scientific community. Reprogramming the software of a running sensor network enables to correct software bugs, test new applications more easily and consequently helps to shorten the development time [2]. Moreover, application reconfiguration can be done by reprogramming the application software. Even though the application behavior adjustment could be performed by modifying operational parameters, a more profound modification, like algorithm changes or even completely updating the software application, cannot be achieved by simply adjusting a set of parameters. Since the new application needs to be transmitted though the network, the reprogramming has an associated energy cost. Moreover, the execution cost depends on the program representation level. An interpreted representation will have an execution overhead if compared to a direct executed program. On the other hand, an interpreted representation typically is smaller than its natively executed counterpart. Furthermore, application specific virtual machines could lead to dense program representation, thus reducing communication cost [3]. Therefore, a precise and more profound analysis is still required to build models for energy and its trade-offs with other system metrics [4].

In this paper we analyze the design space for interpreted languages, considering the different power modes of communication and processing components, and how these evolve with technology scaling.

The remainder of this paper is organized as follows. In Section 2, we survey related work. In Section 3 a first order analytical energy model is derived, considering the different power modes and the activity profiles of the communicating and the processing units. In Section 4 we present the results and delimit the actual space for interpretation in WSN. In Section 5 we discuss possible evolutions of the design space according to the foreseen technology evolution. Finally, Section 6 contains concluding remarks and future research directions.

Related work

The reasons for adopting interpreted languages in WSN are mainly two: the appropriate programming model and the opportunity for energy optimization, and both are interrelated.

Programming models suitable for developing complex distributed applications and at the same time being energy-aware are essential to enable more sophisticated applications. The difficulty in programming sensor networks comes from their inherently distributed nature, and also from their harsh operating conditions, such as unreliable communications [5]. The extremely constrained resources prevent the adoption of proposed solutions like PIECES [6]. To cope with the energy limited budget, sensor network programmers must deal with too many implementation-level details besides the application logic that they normally focus on, and usually to design extremely efficient systems, break the traditional networking and systems layers, thus compromising reuse and other good software engineering principles. Early node-centric programming models are inadequate and unable to scale up. New service architectures, inter-operation protocols, programming models that are resource-aware and resource-efficient, even across heterogeneous devices, are needed [7].

There are several benefits in using virtual machines (VMs) in WSN. First, VMs allow applications to be developed uniformly across WSN platforms, platform-independent applications can be written using VM abstractions whose implementations are scaled to meet resource constraints. VMs provide a clean separation of system software and application software, which reduces the cost of reprogramming after deployment. Finally, VMs mask the variations among the WSN platforms through a common execution framework [8].

Several works had explored the energy trade-off between communication and processing cost, adopting different approaches: dynamic linking of native code, interpreted code execution instead of direct execution, or a hybrid between these two approaches.

A reprogramming mechanism via in-situ dynamic run-time linking and loading of native code to enable application reconfiguration was proposed in [2]. The energy cost of dynamic linking and execution of native code is measured, quantified and compared to the energy cost of transmission and execution of code for two virtual machines (Java and an optimised one). The obtained execution overhead varied from roughly 4 to 100 times, and code reduction size was about 1/15 in the optimised version. The break-even point between direct and interpreted execution ranges from 100 to 40,000 iterations, that is the number of execution completed by a program before a new version is distributed.

Maté [3] is a bytecode interpreter that runs on TinyOS [9], implemented as a single TinyOS component that sits on top of several system components, including sensors, the network stack, and non-volatile storage. Code is broken up into small capsules of 24 instructions, which can self-replicate through the network for code distribution. Larger programs can be composed of multiple capsules. Maté's high-level interface allows complex programs to be very short (under 100 bytes), and consequently reducing the energy cost of transmitting new programs. The execution overhead of some typical instruction was measured by the execution of tight loops: 33.5 times for

a logical and on two words, and just 1.03 times to send a packet. The code reduction size obtained for some applications ranged from 1/100 to 1/400, approximately.

Many Java virtual machines implemented on bare metal microcontroller targeted for wireless sensor networks have been reported, like Squawk[10], and more recently Darjeeling [11] and Taka Tuka[12]. All of them perform some post processing, performing static linking within group of classes and optimising bytecodes to reduce code size. The achieved code reduction was up to 3-4 times w.r.t. the original Java classes.

A hybrid execution environment that enables the co-execution of platform-independent VM instructions with native instructions was proposed in [8]. Platform-independent byte code is interpreted by an interpretive execution engine, while a lightweight native interface is used to access natively implemented functionality. A proxy JIT-compilation on a powerful compilation server is used to compile the relevant bytecode for the node. The authors argue that the problems associated with purely native or purely virtual execution environments are addressed.

None of the previous reviewed works consider all the fundamental parameters involved in these new and complex WSNs, like execution and update rates. Some works establish some relations between those variables, but do not explore the whole space for the interpreted languages execution approach. In this work we develop an analysis of the usage of interpreted languages taking into account not only the ratio from interpreted to native code, but also some physical mote aspects that have been previously disregarded, and are shown to be very important.

Power consumption model

A precise analysis is required to build energy models, in order to analyze their trade-offs with other system metrics [4], in order to carefully design the system and extend its lifetime to the desired duration.

The total energy of the system node results from the sum of each sub-system module or component contribution, which in turn depends on the activity profile and the current consumption of the various operating modes, i.e. for a microcontroller: active, idle, sleep mode, among others. Longer time periods can be analysed based on a periodic behaviour of duration T.

Being T_i the time spent at the power level P_i, we define d_i as the ratio of T_i and the period T. For the rest of the time, the processor is in power P_0, the lowest possible power mode (power down or sleep). The average power can then be expressed as:

$$\overline{P} = \sum_{i=1} P_i \, d_i + P_0 \left(1 - \sum_{i=1} d_i \right) = \sum_{i=1} \left(P_i - P_0 \right) d_i + P_0 \ . \qquad \textbf{(1)}$$

Eq. (1) shows that the average power is the sum of the increment from the lowest power mode to the considered power consumption mode, weighted by the corresponding duty-cycle, plus the lowest power mode, which represents the

minimum power consumption. Thus, the total minimum power dissipated per node is the sum of the minimum power level of all components. Low duty cycle operation is a common approach to minimize the energy drain of the higher power modes. As a result, the energy drain in the lowest possible power mode becomes significant, and must be carefully considered when the average power consumption is calculated.

Since a component could be used for several purposes or be shared by other modules, the time spent at each level must be evaluated. For example, the transceiver can be used to transfer acquired data from the node to a base, or to receive an update of the software application. These services can be considered independent and modeled separately.

Certainly, the energy waste for transitions between different operating modes must be considered. To simplify the derived equations these contributions will be take into consideration increasing the time spend in the higher power level.

The node is basically a reactive system that responds to external stimulus: a successful reception of a packet, a time trigger to initiate some measurement, data ready interruption, and so on. Apart from the active mode the microcontroller must remain at an operation level suitable for using the internal timer/counter to be able to wake-up from the timer expiration interruption. For example, this lowest power mode for microcontroller of a Telosb sensor node [13] - MSP430 microcontroller [14]- is the LPM3, and for the CC2420 radio [15] is the off power mode (oscillator and voltage regulator being off).

We developed a first order analytical energy model using Eq. (1), considering the previously mentioned power modes, and the activity profiles of the communicating and the processing units. Nevertheless, the same procedure can be followed to include any other subsystem. When analyzing the interpreted code and native execution trade-off, the break even iteration is normally derived [3][8].

Average power for native code distribution and execution

The node computing activity can be modeled as a periodic processing system that process data with a period T_e. On average, the computation amount can be considered as a piece of code of size S that runs to completion. Furthermore, the program update size is also S. During computation time the microprocessor is in active mode dissipating power, P_e^{active}, executing bytes at a rate R_e. The rest of the time the microcontroller goes into low power mode with P_e^{sleep}.

The execution average power is calculated as a function of the duty cycle:

$$\overline{P_e^{native}} = P_e^{acitve}\, d_e + \left(1 - d_e\right) P_e^{sleep} = P_e\, d_e + P_e^{sleep} \quad \textbf{(2)}$$

where:

$d_e = S/(R_e \cdot T_e)$ is the execution duty cycle, and

$P_e = P_e^{active} - P_e^{sleep}$ is power increase from the sleep power baseline.

The distribution of new code is performed via radio-frequency communication. The average time between each code upgrade is considered to be T_d. The radio is in

active mode, consuming P_d^{active} power, during the time needed to transfer the code at a rate R_d. The final amount of bytes that goes through the radio is given by the multiplication of the code size by the protocol overhead, k_{mac}, and the overhead for relying packets through the network, k_{nwk}. The rest of the time the radio is in low power mode, draining power P_d^{sleep}.
The distribution average power is:

$$\overline{P_d^{native}} = P_d^{acitve}\, d_d + (1 - d_d) P_d^{sleep} = P_d\, d_d + P_d^{sleep} \quad . \tag{3}$$

where:
$d_d = S_d / (R_d \cdot T_d)$ is the distribution duty cycle,
$S_d = k_{mac} \cdot k_{nwk} \cdot S$, the distribution effective size, and
$P_d = P_d^{active} - P_d^{sleep}$ is power increase from the sleep power baseline.

The total average power for distributing and executing native code is calculated, assuming that distribution and processing are independent task and just adding them:

$$\overline{P^{native}} = \overline{P_d^{native}} + \overline{P_e^{native}} = P_d \frac{k_{mac} k_{nwk} S}{R_d T_d} + P_e \frac{S}{R_e T_e} + P^{sleep} \quad . \tag{4}$$

where:
$P_d = P_d^{sleep} + P_d^{sleep}$ is the total sleep power.

Eq.(4) can be written in the following form:

$$\overline{P^{native}} = \frac{E_d\, S}{T_d} + \frac{E_e\, S}{T_e} + P^{sleep} \quad . \tag{5}$$

where:
$E_d = k_{mac}\, k_{nwk}\, P_d / R_d$ is the energy to distribute a byte of code and,
$E_e = P_e / R_e$ is the energy to execute a byte of code.

Average power for interpreted code distribution and execution

The average power for interpreted code is straightforward to compute, considering that the time to execute interpreted code is increased by the execution overhead of the virtual machine, k_e. In the same way, the time to distribute the interpreted code, corresponding to certain piece of native code, is affected by the distribution factor k_d. This factor is the reciprocal of the bloat factor, term used to denote the code size increment when interpreted code is compiled to native.

The average power for the interpreted case is:

$$\overline{P^{\text{int}\,erp}} = \left(k_d \frac{E_d}{T_d} + k_e \frac{E_e}{T_e} \right) S + P^{sleep} \quad . \tag{6}$$

The trade-off factor

The interpreted versus native average power rate is defined as κ, and a value less than the unit means that interpreted language is preferable of over native, that is, it executes with less power.

$$\kappa = \frac{\left(k_d \frac{E_d}{T_d} + k_e \frac{E_e}{T_e} \right) S + P^{sleep}}{\left(\frac{E_d}{T_d} + \frac{E_e}{T_e} \right) S + P^{sleep}} \quad . \tag{7}$$

The trade-ff factor results from relation of the following values: T_e, T_d, and S, which are application dependant, E_d, E_e and P^{sleep}, which are technology parameters, and finally the interpreted language factors k_e and k_d, resulting from the virtual machine design.

For the case that P^{sleep} is much smaller than the average power of distribution and execution, the gain factor is simply the sum of each relative power weight multiplied by the corresponding factor.

$$\kappa = k_d \frac{\overline{P_d}}{\overline{P_d} + \overline{P_e}} + k_e \frac{\overline{P_e}}{\overline{P_d} + \overline{P_e}} \, . \tag{8}$$

where:

$\overline{P_d} = \frac{E_d S}{T_d}$ is the average power associated to native code distribution,

$\overline{P_e} = \frac{E_e S}{T_e}$ is the average power associated to native code execution.

Break even locus

The break even locus, where the trade-off factor equals the unity (κ=1), does not depend on the sleep power nor on the code size. The derived equation still has four degrees of freedom, but one can substitute the factor T_d/T_e by n, so the break even locus becomes a 3D surface. The parameter n represents the average number of iterations completed by a program before a new version is distributed.

The following expression must be satisfied,

$$(k_e - 1)\frac{E_e}{E_d}n + (k_d - 1) = 0 \cdot \quad (9)$$

and restricts the surface domains by:

$$0 < k_d \leq 1$$

$$0 < k_e \leq 1 - \frac{E_d}{E_e}\frac{T_e}{T_d} = 1 - \frac{\overline{P}_d}{\overline{P}_e}$$

The former expression limits the distribution factor to positives values less than the unit, since a reduction factor is considered. The last restriction comes simply by substituting the first one in the surface Eq. (9).

Results

Measurements of the energy parameters

We measured the total system current, radio plus microcontroller, of a Telosb mote powered by batteries (3.3 V) in steady state for the meaningful combination of operation modes. Then, the separated values were obtained subtracting different measurements. The protocol overhead k_{mac} is considered constant, and we have used an estimated value of 1.2 (range from about 1.1 to 1.3 for payload greater than 70 bytes). For the code diffusion we considered the Delunge protocol [16], thus the overhead factor, k_{nwk}, is about 3.35 times the number of received packets and one more time for retransmission. Table 1 shows the results.

Table 1. Telosb mote measured parameters.

	CC2420		MSP430	Units
P_d^{active}	68.10	P_e^{active}	1.20	mW
R_d	31250	R_e	1000000	B/s
E_d	11360	E_e	1.20	nJ/B
P_d^{sleep}	0.001	P_e^{sleep}	0.015	mW

Note that the energy values are considered for processing or communicating one byte of executed code and not per byte of information processed. The last one is usually used to analyse the trade-off between process-before-transmit information, while the first one is used to calculate the total energy when a certain amount code is executed or transmitted. The rate of distribution to processing energy is almost 10,000, stressing the huge advantage of locally processing information instead of transmitting it.

The radio oscillator startup time, i.e. transition from low to active power, is about 600 μs, and corresponds to the time used to transmit about 18 bytes, roughly the MAC protocol overhead. The time for the microcontroller to go into active mode is about a few clock cycles, considered negligible.

Code size

Assuming the evolution of technology, and also using this evolution to integrate more powerful processors in a WSN node, we compared the code size for a 10-tap FIR filter written in C and compiled to native MSP430 code. Also, we implemented the filter in Java and counted class bytecodes, discarding some bytes not useful during execution. Table 2 shows the results.

Table 2. Code and data memory comparison for a FIR filter (MSP430).

	text	Data	bss	ROM	RAM
Native (MSP430)	442	32	40	474	72
Java	262	-	-	262	-
Java/Native	0.59			0.55	

This simple example shows that Java code is denser than native code. However, the code reduction obtained is still modest, since a low size class leads to high overhead and because the low complexity of the application prevents code reuse.

Simulation results

The actual space for interpreted code can be obtained from the above equations and the corresponding hardware parameters of Table 1.

The break-even point triplets $\{n, k_e, k_d\}$ are plotted in the Fig. 1 as curves k_d (n, k_e). For example, considering a VM with a 30-fold execution time overhead and a distributing factor of 0.1, then the number of iterations (the number of times a program is executed before it is updated) required to have the same average power of interpreted and native code is about 300.

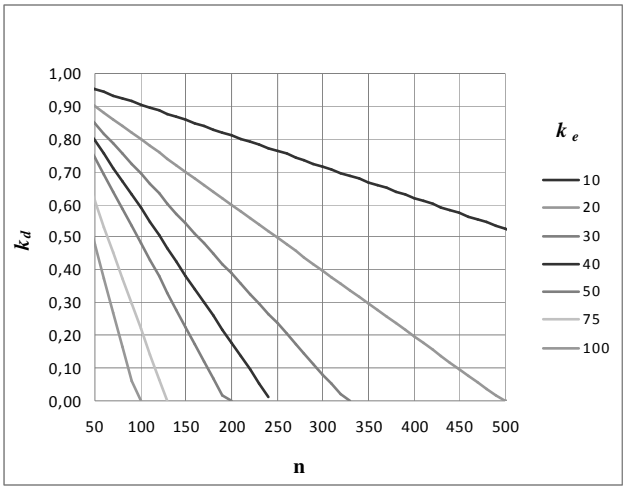

Fig. 1. Break-even locus.

Discussion

The energy consumed by the radio can be separated in two components: electronic power and transmission radio power. The former is consumed by digital and analog processing circuits required for wireless communication or, in other words, digital and analog circuits that perform the necessary RF, baseband, and protocol processing. The last component, associated to the irradiated power, is consumed by the radio power amplifier that depends on the signal power required by the receiver and the path loss suffered by the signal. The path loss increases proportionally with receiver-transmitter separation distance, and depends on the environment condition, being a power of two in free space but up to four in real life channels[17]. As it is dictated by the Shannon's Information Theory and Maxwell's Laws, this power cannot be reduced.

In medium-to-large range communication the radio-power dominates (over the electronic-power), and in many cases the transmission power is orders of magnitude greater than reception power. However, in short range communication the electronic-power takes about the same radio power needed for successful reception at the desired short-distance, so it can get some benefit from technology progression, limited by the remainder component, waveform propagation power, which does not benefit from any technology progression.

Consequently, the communication energy cost tends to get less benefit from technology scaling than processing cost. The current consumption of the new MSP430F5XX family product is about 30% less than the MSP430F1611 used in the Telosb mote, while the consumption of the new radio CC2520 is virtually equal to the old CC2420. This is a clear evidence of our argument. What is more, many

environmental variables vary little with the distance, pushing to increase the separation among nodes, and consequently raising the communication power.

Analyzing the break even curves given by Eq. (8), one can argue that as the rate E_e/E_d decreases, the number of iterations to reach the break-even point may increase, not affecting the energy budget. This fact poses the interpreted code execution approach in a promising position in applications where relatively high rate of code updates are needed, and especially but not exclusively, where communication covering medium-range distances are involved.

Conclusions and future work

In this paper we have developed an energy model to investigate the efficiency of using interpreted languages as the basic platform for software development of future complex applications built on top of WSN. Experimental results have shown that a huge savings in code space and amount of transmitted information can be obtained when an interpreted language like Java is used. Moreover, following technology scaling, it is very likely that future applications will be able to use complex processors, and saving energy during the transmission of information or code will be the most effective optimization procedure.

Currently we are developing complex dynamic applications to validate the approach here proposed.

References

1. Mitsubishi's M64282FP CMOS image sensor. Available on-line: http://www.seattlerobotics.org/Encoder/200205/downloads/M64282FP.pdf
2. Dunkels, A., Finne, N., Eriksson, J., and Voigt. T, Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th international Conference on Embedded Networked Sensor Systems* (Boulder, Colorado, USA, October 31 - November 03, 2006). SenSys '06. ACM, New York, NY, 15-28.
3. Levis, P. and Culler, D. Maté: a tiny virtual machine for sensor networks. SIGOPS Oper. Syst. Rev. 36, 5 (Dec. 2002), 85-95.
4. Zhao, F. 2008. Technical Perspective: The physical side of computing. Commun. ACM 51, 7 (Jul. 2008), 98-98.
5. Sugihara, R. and Gupta, R. K. Programming models for sensor networks: A survey. ACM Trans. Sen. Netw. 4, 2 (Mar. 2008), 1-29.
6. Liu, J.; Chu, M.; Reich, J.; Zhao, F. "State-centric programming for sensor-actuator network systems," *Pervasive Computing, IEEE* , vol.2, no.4, pp. 50-62, Oct.-Dec. 2003.
7. Feng Zhao, Challenges in Programming Sensor Networks, DCoSS 2005.
8. Koshy, J., Wirjawan, I., Pandey, R., and Ramin, Y. Balancing computation and communication costs: The case for hybrid execution in sensor networks. Ad Hoc Netw. 6, 8 (Nov. 2008), 1185-1200.
9. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. System architecture directions for networked sensors. SIGPLAN Not. 35, 11 (Nov. 2000), 93-104.

10. Simon, D., et al. Java on the bare metal of wireless sensor devices: the squawk Java virtual machine. In Proceedings of the 2nd international Conference on Virtual Execution Environments (Ottawa, Ontario, Canada, June 14 - 16, 2006). VEE '06. ACM, New York, NY, 78-88.
11. Brouwers, N., Corke, P., and Langendoen, K. Darjeeling, a Java compatible virtual machine for microcontrollers. In *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion* (Leuven, Belgium, December 01 - 05, 2008). Companion '08. ACM, New York, NY, 18-23.
12. Aslam, F., Schindelhauer, C., Ernst, G., Spyra, D., Meyer, J., and Zalloom, M. Introducing TakaTuka: a Java virtualmachine for motes. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems* (Raleigh, NC, USA, November 05 - 07, 2008). SenSys '08. ACM, New York, NY, 399-400.
13. Telosb Mote Platform (rev. B). Crossbow Technology, Inc. Available on-line: http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/TelosB_Datasheet.pdf
14. Texas Instruments Inc., MSP430x15x, MSP430x16x, MSP430x161x Mixed Signal Microcontroller (Rev. E).
Available on-line: http://focus.ti.com/docs/prod/folders/print/msp430f1611.html.
15. 2.4 GHz IEEE 802.15. 4/ZigBee-Ready RF Transceiver (Rev. B). Available on-line: http://focus.ti.com/docs/prod/folders/print/cc2420.html.
16. J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In Proc. SenSys'04, Baltimore, Maryland, USA, November 2004.
17. M. Srivatsava, "Power-aware communication systems," in Power Aware Design Methodologies. Norwell, MA: Kluwer, 2002.