Energy-aware Compilation for Wireless Sensor Networks

Daniel A. Sadilek
Humboldt-Universität zu Berlin
Institute for Computer Science
Rudower Chaussee 26
10099 Berlin, Germany
sadilek@informatik.hu-berlin.de

ABSTRACT

An application for a reconfigurable virtual machine for wireless sensor networks consists of both bytecode and native code. Identifying the abstraction boundary between bytecode and native code has not been addressed systematically, yet. Up to now, a programmer has to specify the abstraction boundary manually without being able to judge his decision. We present a mathematical model for benchmarking the energy consumption of bytecode and native code. This model allows a founded specification of the abstraction boundary. In conjunction with an extended reconfigurable virtual machine architecture, it provides a flexible development method with less development effort than existing approaches.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems;
D.2.8 [**Software Engineering**]: Metrics—*Product metrics*;
C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Network operating systems, Distributed applications*;
D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*

General Terms

Design, Measurement, Management, Theory

Keywords

Wireless Sensor Networks, Virtual Machines, High-level Languages, Network Reprogramming, Energy-Awareness, Compilation

1. INTRODUCTION

A wireless sensor network (WSN) consists of spatially distributed, wirelessly communicating, small sensor nodes. They are equipped with a radio transceiver, sensors, and a microcontroller. Often, sensor nodes are battery-powered and deployed in inaccessible places. When application requirements change after deployment, the sensor nodes must be reconfigured without physical access. At the same time, energy must be conserved as much as possible.

Programming WSNs with low-level languages like C or even Assembler is tedious and unsafe. Sensor nodes lack a memory protection unit, typical operating systems lack preemptive multitasking, and there is typically no automatic memory management. Easier programming as well as safer execution environments are needed. Easier programming can be achieved with special programming languages tailored for WSN programming, e.g. [3, 11, 13]. A safer execution environment can be achieved by code instrumentation [4] or interpretation of bytecode by a *virtual machine*. Here, we concentrate on the virtual machine approach. It does not only provide a safe execution environment but also eases deployment and allows the usage of special programming languages that are compiled to bytecode. Furthermore, it provides smaller code sizes and may support dynamic memory as well as preemptive multitasking.

The best-known virtual machine for WSNs is Maté [8], which was presented in 2002. Already then, the authors suggested that the virtual machine's instruction set should be extended with domain-specific instructions. This was to save bandwidth when transmitting the bytecode program by making it smaller and to reduce interpretation overhead by natively implementing processing intense tasks. However, Maté was a first prototype and monolithic. The complete system image had to be reprogrammed when the virtual machine's instruction set had changed.

Later, proposals from the same authors [9] and other approaches like DVM, DaVIM, and VM* [1, 10, 7] drove the flexibility of the virtual machine approach ever further. These approaches allow to *reconfigure* the virtual machine's instruction set at runtime: native code modules can be installed without physical access via wireless communication and can be made available as bytecode instructions.

Up to now, the decision which parts of an application should be compiled to bytecode and which ones to native code has not been addressed systematically. We fill this gap with a method for making the decision based on the energy consumption of the application.

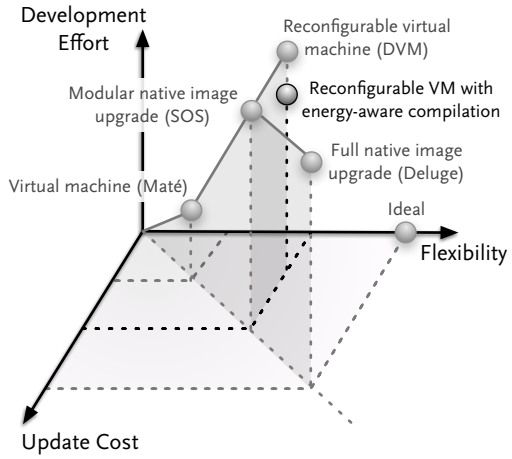

Figure 1: Classification of existing approaches and our proposed extension.

The rest of this paper is structured as follows. In the next section, we examine and classify related work on which we build. We propose an extension of an existing virtual machine architecture in Section 3. Our main contribution is the definition and theoretical analysis of a mathematical model for energy-aware compilation in Sections 4 and 5. We conclude in Section 6.

2. RELATED WORK

As already stated in the introduction, our work builds on enhancements of the Maté [8] idea, namely those proposed in [9, 1, 10, 7]. These approaches and other alternatives can be classified in three dimensions: flexibility, update cost and development effort (Fig. 1).[1] The flexibility of an approach is defined as the number of different applications that can be deployed on sensor nodes without physical access. The update cost is defined as the energy consumed per node when changing the deployed application. The development effort is defined as the work amount necessary to write, test, and deploy a new application.

The ideal approach would lie directly at the right-most position on the flexibility axis. It would allow to deploy literally any application at no update cost and without any development effort. But there is a tradeoff between all dimensions; the ideal approach does not exist.

The most dominant tradeoff is between flexibility and update cost. *Virtual machines* like Maté are at one end of this tradeoff. Using a virtual machine, one is restricted to its bytecode allowing only a limited number of different applications; but the update costs are very low due to the concise bytecode. *Full native image upgrades* like with Deluge [6] are at the other end of that tradeoff: changing the full image costs much energy but provides maximum flexibility. *Modular native image upgrades* like with SOS [5] are at the middle of that tradeoff: changing only modules costs not as much energy as changing the full image; but there is a system kernel that cannot be changed, which restricts flexibility. *Reconfigurable virtual machines* like DVM [1] reduce the tradeoff. They are a combination of the approaches virtual machine and modular native image upgrade: the virtual machine's instruction set can be extended at runtime by providing the implementation of additional instructions as native modules. Thus, reconfigurable virtual machines have the same flexibility as systems offering only modular binary image upgrades; at the same time, they have lower update costs because large parts of an application are encoded as concise bytecode.

Another tradeoff is between flexibility and development effort. Writing bytecode for a virtual machine is inexpensive but inflexible. Writing code for native image upgrades (both modular and full) requires very detailed knowledge about the module and/or operating system and bugs can easily creep in. With a reconfigurable virtual machine, parts of an application can be developed as bytecode. This is less error-prone and requires less detailed knowledge than writing native code. However, writing the native instruction implementations remains complex. Additionally, choosing which parts of the application should be written as bytecode and which ones as native code remains a manual task.

In hardware/software co-design, a rating model similar to the one we propose is used to optimize hardware/software partitioning. This optimization has to take limited hardware space into account, e.g. by modeling the problem as a knapsack problem [12]. We do not consider such limitations, which makes our model easily solvable.

In [2], Dunkels et al. conclude that the combination of native code and virtual machine code may be more energy efficient than pure native code or pure virtual machine code solutions. We refine their energy model to provide a means for deciding where the abstraction boundary between native code and virtual machine code should lie.

3. ARCHITECTURE PROPOSAL

In the last section, we saw that reconfigurable virtual machines are advantageous as they reduce the tradeoff between flexibility and update cost. Still, using them requires high development effort.

To reduce the development effort, *the application developer must be relieved of writing low-level code and of deciding where the abstraction boundary between bytecode and native code lies.* We think that the existing reconfigurable virtual machine approaches can be extended to reach this goal. The extended approach will have the same flexibility at the same update costs like the ordinary reconfigurable virtual machine but with lower development effort (Fig. 1).

In order to relieve the developer from writing low-level code, we provide him with a *high-level language* that can be compiled to both bytecode and native code. This idea is sketched in Fig. 2.

But is this even feasible? To answer this question, we look at the reconfigurable virtual machine approach and differentiate between three tasks (or use cases) of code: control flow, signal processing, hardware access.

- The *control flow* of an application controls which actions should be taken when and in which order. In WSNs, we consider timer and event handlers to be part of the control flow. Control flow isn't very processing intense; also, it doesn't do fancy things with the hardware that couldn't be foreseen when the virtual machine was developed. Therefore, control flow

[1] We use the classification from [1] and add the dimension "development effort".

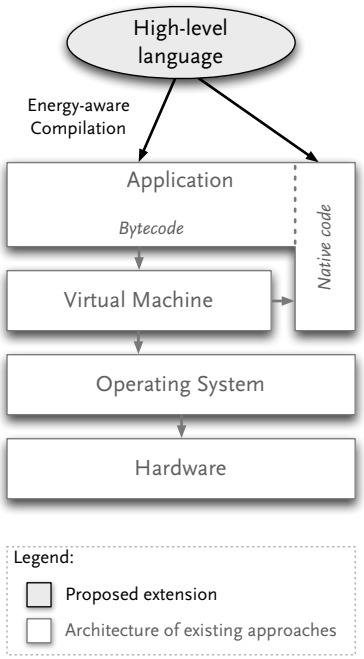

Figure 2: Energy-aware compilation as an extension of existing reconfigurable virtual machine approaches.

can be encoded in bytecode.

- WSN applications usually contain some form of *signal processing*. Signal processing tasks can be processing intense. Encoding them in bytecode is impractical because of the interpretation overhead the virtual machine involves. Therefore, signal processing must be encoded as native code and called from the control flow bytecode.
- Some WSN applications may need *access to the hardware* that couldn't be foreseen when the virtual machine was developed. For instance, an application may require to change the wireless medium access. If the virtual machine isn't prepared for that, the medium access must be implemented as native code.

We expect programming a WSN application solely in a high-level language to be feasible if the application consists only of control flow and signal processing. We expect it to be infeasible if the application needs fancy hardware access. For the high-level language would need a powerful native interface, which is against the high-level intention.

How could the high-level language look like? It may be, for example, a C-like language without C's intricacy; it may be a JavaScript-like language similar to TinyScript (part of Maté); or it may be a domain-specific language tailored not for WNSs in general but for a specific application domain.

With the proposed architecture, we relieve the developer from writing low-level code. Additionally, the architecture allows for relieving him from his second burden: deciding where the abstraction boundary between bytecode and native code lies. When the complete application is described as high-level code, the compiler can make this decision as described in the next section.

4. ENERGY-AWARE COMPILATION

In this section, we describe a mathematical model for the decision which parts of an application should be compiled to bytecode and which ones to native code—with the goal to minimize the overall energy consumption. Compiling a part of an application to bytecode introduces a tradeoff. On the one hand, bytecode gets interpreted and, on the other hand, bytecode is smaller than functionally equivalent native code. The interpretation consumes *more*[2] energy at runtime due to the interpretation overhead; the transmission of the bytecode costs *less* energy due to the smaller size. We need an instrument to benchmark this tradeoff.

4.1 Mathematical model

Let the energy consumption E of a program p be

$$E(p) = E_{\text{inst}}(p) + E_{\text{run}}(p) \ . \tag{1}$$

$E_{\text{inst}}(p)$ is the *installation cost*: the energy per node that is necessary for delivering the application over the network and installing it on the node. $E_{\text{run}}(p)$ is the *runtime cost*: the energy that the application consumes over its lifetime on one node.

We consider an application p to be composed of n individual *functions* f_i, $i \in \{1, \ldots, n\}$. Let the installation cost $E_{\text{inst}}(p)$ of an application be the sum of the installation costs $E_{\text{inst}}(f_i)$ of the individual functions:

$$E_{\text{inst}}(p) = \sum_{i=1}^{n} E_{\text{inst}}(f_i) \ . \tag{2}$$

Analogous, let the runtime cost $E_{\text{run}}(p)$ of an application be the sum of the runtime costs $E_{\text{run}}(f_i)$ of the individual functions. Furthermore, let the runtime cost of a function be

$$E_{\text{run}}(f_i) = a(f_i)\, e(f_i)\, T(p) \tag{3}$$

where $a(f_i)$ is the number of invocations per time of the function, $e(f_i)$ is the energy the function consumes in one invocation, and $T(p)$ is the length of time in which the application is installed and active on the sensor network. So we get for the runtime cost of an application:

$$E_{\text{run}}(p) = \sum_{i=1}^{n} a(f_i)\, e(f_i)\, T(p) \ . \tag{4}$$

Combining equations 1 to 4 leads to

$$E(p) = \sum_{i=1}^{n} E(f_i) \tag{5}$$

where $E(f_i) = E_{\text{inst}}(f_i) + a(f_i)\, e(f_i)\, T(p)$ is the overall energy consumption of the function f_i. The overall energy consumption $E(p)$ of an application can be minimized by minimizing $E(f_i)$ for each function individually. To decide whether f_i should be compiled to bytecode or to native code, we distinguish between the overall energy consumption of the function when compiled to bytecode $E_{\text{byte}}(f_i)$ and when

[2]compared to native code

compiled to native code $E_{\text{nat}}(f_i)$. The question whether a function should be compiled to native code can then be stated as

$$E_{\text{nat}}(f_i) < E_{\text{byte}}(f_i) \ . \tag{6}$$

To solve this inequality, we need to quantify the energy savings and expenses. This can be done with two factors: σ_i for the saving of energy at installation time and ρ_i for the higher energy consumption of bytecode at runtime. The factors are given by

$$E_{\text{inst,byte}}(f_i) = \sigma_i \cdot E_{\text{inst,nat}}(f_i) \ , \tag{7}$$

$$e_{\text{byte}}(f_i) = \rho_i \cdot e_{\text{nat}}(f_i) \ . \tag{8}$$

Assuming $\rho_i > 1$, it follows that

$$\boxed{T(p) \ > \ \frac{1-\sigma_i}{\rho_i - 1} \cdot \frac{E_{\text{inst,nat}}(f_i)}{a(f_i)\, e_{\text{nat}}(f_i)}} \ . \tag{9}$$

This inequality can be used to decide for each individual function f_i whether it should compiled to bytecode or to native code.

4.2 Example calculation

Let's consider a function f_k that is executed once per minute, $a(f_k) = 1/\text{minute}$, and consumes $e_{\text{nat}}(f_k) = 100\,\text{nJ}$ per invocation in its native code form. Let the installation cost of the function as native code be $E_{\text{inst,nat}}(f_k) = 10\,\text{mJ}$. Let the bytecode size be one tenth of the size of the native code, $\sigma_k = 0.1$, and the runtime overhead $\rho_k = 20$.

Now, we can ask how long the application must be running on the sensor network so that the function as native code has a lower overall energy consumption than the function as bytecode. Applying the input parameters to Inequality 9, we get approximately $T(p) > 3\,\text{days}$. This means that the *break-even time* is three days: if the application is running less than three days, the function f_k should be compiled to bytecode; if it is running longer, f_k should be compiled to native code.

5. THEORETICAL ANALYSIS

In this section, we analyze the mathematical model theoretically. Is it worthwhile trying to implement it or are there any principle obstacles? In the following subsections we discuss three issues that need clarification: Where do the input parameter values come from (Section 5.1)? How sensitive is the model to the input parameters (Section 5.2)? What are the implicit assumptions in the model and where may they fail (Section 5.3)?

5.1 Input parameter estimation

In order to apply our approach, input parameters are needed. In the following, we describe how they can be estimated.

$a(f_i)$: invocation frequency.

Most WSN applications run periodically. Timers start handler functions with a specific frequency $a(\text{timer})$. Handler functions can call other functions (dependent functions). The invocation frequency of a handler function f_h is $a(f_h) = a(\text{timer})$. To specify the invocation frequency of a dependent function f_d, an additional statistical parameter is needed: the calling rate r_d^h. r_d^h is the average rate the handler function f_h calls the dependent function f_d. f_d may be called conditionally, in this case $0 < r_d^h < 1$; or it may be called inside a loop with a fixed number of iterations, in this case $r_d^h \in \{2, 3, 4, \ldots\}$. r_d^h represents a combination of both conditional and looped calls; therefore, in the general case $r_d^h > 0$. Given r_d^h, the invocation frequency of f_d is $a(f_d) = r_d^h a(f_h) = r_d^h a(\text{timer})$.

In WSN applications there are also event handler functions that are invoked irregularly—for instance, a handler function for data packets received over the wireless network interface. For such an irregularly called function f_{irr}, $a(f_{\text{irr}})$ can only be given directly.

The invocation frequency of timer handler functions and non-conditionally called dependent functions can be obtained by static code analysis. Rate parameters of conditionally called dependent functions and the invocation frequency of irregularly called handler functions must be estimated based on the application characteristics. For this, simulating the application will be necessary.

$e_{\text{nat}}(f_i)$: invocation energy.

Estimating the energy consumption of a function invocation is difficult. One may count the instructions of a function if it has no conditional jumps (compiled loops and conditional statements). With a processor profile at hand, the energy consumed by the instructions can be calculated. When the function contains conditional jumps, profiling the application is necessary in order to get an average execution path for which the energy consumption can be calculated. For this, a simulation will be necessary, as well.

$E_{\text{inst,nat}}(f_i)$: native code installation energy.

If we assume that the installation cost $E_{\text{inst,nat}}(f_i)$ of a function is proportional to its code size $s_{\text{nat}}(f_i)$ (see Section 5.3), then the installation cost is simply $E_{\text{inst,nat}}(f_i) = \epsilon \cdot s_{\text{nat}}(f_i)$ where ϵ is the energy required to install one byte of code.

ϵ must be estimated only once for a particular hardware platform and network layout of the WSN by direct measurement. The code size $s_{\text{nat}}(f_i)$ of a function can easily be estimated by compiling it.

σ_i: bytecode installation saving.

σ_i is the ratio between the installation energy of bytecode and native code. We estimate the installation energy of code by its size.

Therefore, σ_i can easily be estimated by the ratio between the function's size as bytecode and as native code: $\sigma_i = \frac{s_{\text{byte}}(f_i)}{s_{\text{nat}}(f_i)}$.

ρ_i: bytecode runtime overhead.

The virtual machine introduces a substantial overhead. Consider, for instance, addition of two numbers. If implemented natively, this is only one processor instruction and will take very few processing cycles. If implemented as bytecode, the virtual machine needs to grab the instruction, dispatch to the corresponding instruction implementation, which has to move the instruction parameters from the virtual stack to the real stack, call the native addition, copy the result from the real stack to the native stack and return control to the virtual machine's main dispatch function. This

Table 1: Break-even time $T(p)$ for different input parameters. Varied parameters are highlighted gray.

Input parameters					
$1/a(f_k)$ [s]	$e_{\text{nat}}(f_k)$ [nJ]	$E_{\text{inst,nat}}(f_k)$ [mJ]	σ_k [ ]	ρ_k [ ]	T(p)
60.0	100	10	0.10	20	3 days
60.0	150	5	0.15	30	1 day
60.0	50	15	0.05	10	22 days
0.1	100	10	0.10	20	8 min.
3600.0	100	10	0.10	20	197 days

requires lots of processing cycles.

Let's assume that the virtual machine's core instruction set[3] contains only simple instructions like addition. In their native implementation such instructions require only very few processor cycles. This means that the interpretation overhead is so big compared to the native instructions that it hardly matters which native instruction is called.

Therefore, the runtime overhead ρ_k must be estimated only once for a specific virtual machine implementation.

5.2 Sensitivity estimation

In this section, we want to get a feeling for the sensitivity of the proposed model and we want to know how good the different parameters must be estimated in order to get a reasonable result. For this, we try variations of an example. We vary the input parameters and notice how the break-even time $T(p)$ changes (the time the application p has to be deployed before compiling the considered function to native code pays off). The different variations are listed in Table 1.

We start in the first row with the input parameters from the example given in Section 4.2. In the second and third row we vary all parameters except the invocation frequency. We vary them by 50 %—with the objective of a lower break-even time in the second row and a higher one in the third row. This causes the break-even time to range between 1 and 22 days, which is quite a big span but "only" within an order of magnitude.

In the fourth and fifth line we vary only the invocation frequency $a(f_k)$ between 10 calls per second and one call every hour. This causes the break-even time to range between 8 minutes and 197 days, which is within five orders of magnitude.

We can conclude: if the invocation frequency $a(f_k)$ is given and we a manage to estimate the other input parameters with an accuracy of 50 % we can calculate the break-even time $T(p)$ within an order magnitude. Furthermore, if we had only very rough estimates of the input parameters, the exact value of the calculated break-even time won't make much sense. However, the range of the invocation frequency $a(f_k)$ is so broad that even in this case we would get a hint whether a function should be compiled to bytecode or to native code.

[3]The instruction set without application-specific instructions.

5.3 Model assumptions

We made some implicit assumptions in the description of the mathematical model in Section 4. In the following, we list and asses them.

$E(p) = E_{\text{inst}}(p) + E_{\text{run}}(p)$ (Eq. 1): Total energy cost of a program is the sum of the installation cost and the runtime cost of the program.

This cannot turn out to be false as it isn't really an assumption. It doesn't matter how the energy cost of a program looks like; it is always possible to divide it into installation cost and runtime cost. However, this distinction may turn out to be artificial and inappropriate—for instance, if the runtime cost turns out to be negligible small in all cases.

$E_{inst}(p) = \sum_{i=1}^{n} E_{\text{inst}}(f_i)$ (Eq. 2): Installation cost of an application is the sum of the installation costs of the individual functions.

This assumes that the installation cost E_{inst} of an application is proportional to its size.[4] Otherwise, the installation cost of a function would depend on the overall size of the program. Our assumption is only an approximation. In reality, application code is transmitted in packets. Therefore, energy cost as a function of code size will be more staircase-shaped with steps at multiples of the packet size and a small slope in between.

$E_{\text{run}}(f_i) = a(f_i)\, e(f_i)\, T(p)$ (Eq. 3): The runtime cost of a function is the product of its invocation frequency, its energy consumption per invocation and the lifetime of the application.

Here we have two assumptions. One assumption lies in the parameters itself: We assume that the invocation frequency $a(f_i)$ and the invocation energy $e(f_i)$ are either constant or that reasonable averages can be estimated. As stated in Section 5.1, this may be false for irregularly called functions or functions with conditional execution branches.

Another assumption is that the invocation energy $e(f_i)$ is independent of the invocation frequency—or, more general, the processor load. This may be false, for example, if the processor heats up when processing, which causes higher energy consumption. Then, comparing bytecode and native code of a function based on their energy consumption would be no more appropriate.

6. CONCLUSION

We presented a development approach for WSNs that is an extension of existing reconfigurable virtual machine approaches. Our approach allows the same application flexibility with the same update costs as the existing approaches but with less development effort. The application developer is freed from writing low-level code by hand and from specifying manually which functions of an application should be compiled to bytecode and which ones to native code.

Our main contribution is a mathematical model for benchmarking the energy-tradeoff between bytecode and native code. We analyzed how the model's input parameters can be estimated, how sensitive the model is to the input parameters, which assumptions we put into the model, and where they may fail. For estimating the input parameters,

[4]We made the same assumption for estimating $E_{\text{inst,nat}}(f_i)$ and σ_i (Section 5.1).

we identified the need to simulate applications in order to get execution profiles. In our analysis, we did not encounter any principle obstacles for the proposed approach.

The next step will be to implement our approach. Then we will see if our assumptions hold and if we can estimate the input parameters accurately enough to get reasonable results.

Acknowledgments

Thanks to Joachim Fischer and Andreas Kunert for stimulating discussions. Also thanks to Kai Köhne, Timo Mika Gläßer and the anonymous reviewers for comments on preliminary versions of this paper. This work was supported by grants from the DFG (German Research Foundation, research training group METRIK).

7. REFERENCES

[1] R. Balani, C.-C. Han, R. K. Rengaswamy, I. Tsigkogiannis, and M. Srivastava. Multi-level software reconfiguration for sensor networks. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 112–121, New York, NY, USA, 2006. ACM Press.

[2] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, November 2006.

[3] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM Press.

[4] L. Gu and J. A. Stankovic. t-kernel: providing reliable os support to wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 1–14, New York, NY, USA, 2006. ACM Press.

[5] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176, New York, NY, USA, 2005. ACM Press.

[6] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM Press.

[7] J. Koshy and R. Pandey. Vmstar: synthesizing scalable runtime environments for sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 243–254, New York, NY, USA, 2005. ACM Press.

[8] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(5):85–95, 2002.

[9] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association.

[10] S. Michiels, W. Horré, W. Joosen, and P. Verbaeten. Davim: a dynamically adaptable virtual machine for sensor networks. In *MidSens '06: Proceedings of the international workshop on Middleware for sensor networks*, pages 7–12, New York, NY, USA, 2006. ACM Press.

[11] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 489–498, New York, NY, USA, 2007. ACM Press.

[12] A. Ray, W. Jigang, and S. Thambipillai. Knapsack Model and Algorithm for HW/SW Partitioning Problem. In *ICCS '04: International Conference on Computational Science*, number 3036 in Lecture Notes in Computer Science, pages 200–205, Kraków, Poland, 2004. Springer.

[13] K. Terfloth, G. Wittenburg, and J. Schiller. Facts — a rule-based middleware architecture for wireless sensor networks. In *COMSWARE 2006: First IEEE International Conference on Communication System Software and Middleware*, New Delhi, India, January 2006.