

Program Specialization for Wireless Sensor Networks

Background:

Wireless sensors networks collect information from remote locations in a cost effective and reliable manner. Examples include Harvard's volcano eruption monitoring project [7] and the SWE project at the University of Vermont to measure the snow-water equivalent in a snow pack [6]. These autonomous sensors are extremely efficient because memory, speed, and power are limited. The devices consume as little energy as possible in order to run off a small battery for an extended period of time. The goal of wireless sensor software is to run quickly and utilize the smallest amount of resources and energy [2].

TinyOS is an event-driven operating system specialized for memory constrained embedded systems, specifically wireless sensors. The language that is most commonly used to develop wireless sensor network applications on the TinyOS platform is nesC, which has the low-level control benefits of C, while optimizing the resource limitations of a wireless device. The flexible, component-based design of nesC uses tasks and events to support the particular requirements of this domain. NesC allows several software components to be statically wired together to form node-level applications [3].

The goal of this research project is to improve efficiency of a wireless sensor network in three different ways: first to increase the ease of programming with an improved programming abstraction, second to reduce the overall code footprint, and third to allow for better memory utilization. Benchmark testing will finally be performed and results will be processed and published with the assistance of Dr. Christian Skalka (Associate Professor, Computer Science) as part of his grant for Programming Language-Based Access Control for Wireless Sensor Networks [5].

The first phase of the implementation will be accomplished using staged programming. Multi-stage programming provides, in addition to the usual constructs of a general-purpose language, both partial evaluation and program specialization techniques. This causes a significant increase in efficiency, because code generation, reuse, and execution are supported [4].

Partial evaluation is a technique used to compact code for optimization by reducing computational effort for situations that have an unchanging temperament. The compiler performs pre-computations of these common situations in order to save time during execution. An example would be calculating a routing table in the first stage of computation for the wireless sensor network and using this as a constant value in a separate section of the application. In addition, constant values may then be stored in ROM, freeing up the limited availability of RAM (2kb on our devices). Releasing RAM for individual wireless devices will increase the efficiency of the program significantly allowing wireless sensors networks to not only run longer and more effectively, but also increase the sophistication of the programs they are running [1]. Giving otherwise impossible applications, such as crypto protocols using multiple keys, the ability play a part in the security of wireless devices.

Hypothesis:

As noted above, we hypothesize that the advantages of staged programming include ease of programming, code reduction, and greater memory utilization. Given that efficiency with wireless sensors is extremely important, this cannot be a purely theoretical issue. Implementation is essential. The language extension to nesC (named nesT) will consist of a parser, compiler and run-time support, including static analysis to ensure any generated program is syntactically correct and well typed. The multi-staged program built with the extension will allow for program optimization by runtime algorithm specialization. We also hypothesize that quantitative efficiency gains of the system will be revealed by benchmarking and empirical testing the application.

Specific aims:

Previous work with partial evaluation has been shown to benefit embedded systems as demonstrated [1], however very little has been done in the area of wireless sensor program specialization. Our objective is to realize this theory in practice as an extension of nesC. For this to happen we must implement the language so that compilation and over-the-air programming of communicating devices is supported at runtime. Our goal is to make this process part of nesT itself, rather than an ad-hoc process. Instead of compiling a file, saving it to a text file, then opening it with a shell command, we envision an

entirely new communication scheme. The new extension will combine all these steps into a single application that runs on a *hub* (a more powerful “master” processor such as a laptop) as the initial stage of the program. The second stage occurs on the wireless sensors after the code has been sent over the network and loaded with a bootloader on the low-power network *nodes*.

Herein lies the most interesting and novel part of this project, which is how we combine these actions into a single idiom. The basic syntax running on the hub is *run <p> at m* where *p* is a program run on a specific wireless sensor *m*. The hub will run its program normally until it comes to *<p>*. Only when the keyword *run* is used before *<p>*, will *p* be compiled, and sent over the network to the sensor. Furthermore, *p* may be dynamically constructed as part of the hub program computation- so there are *two-stages* of computation- one on the hub and one on the node(s). After the hub ships the code to the wireless sensor node, the hub will continue on where it left off in the program.

Multi-staging creates these generic programs, in which *<p>* can have any value, be called multiple times in the program and be sent to several sensors at the same time. With this programming technique, the more powerful processor can do all the memory tasking calculations and lift these calculations as a constant value into the program *p*. With this syntax we can calculate certain values only when we need and these pre-computed values can be brought into expressions with ease.

Following the implementation of this staging procedure, the components of nesT will undergo controlled tests for relative performance. Conservation of memory and speed will be analyzed, particularly for efficiency in the new extension and compiler versus systems that do not use multi-staging. Benchmark testing against systems that do not use staging will be created for the application and hub to node communication to determine if any additional performance gains are obtained via our methods. Using TinyDB [9] we can gather information about a network of sensors to benchmark the runtime efficiency of the program, memory usage and the code size. Since TinyDB is a common application and many benchmarking tests have been recorded already in wireless sensor network platforms, it will provide a good point of comparison.. Tests will also be created to stress the system for efficiency in other representative scenarios and any bugs will be ironed out. The data will then be empirically analyzed and assembled into at least one article, submitted for publication at a prestigious international conference such as the International Conference on Embedded Software (EMSOFT).

Methods:

In addition to the direct mentorship of Dr. Skalka, this project will be carried out under the student leadership of graduate students under the direction of Professor and Chair Dr. Scott Smith at John Hopkins University Department of Computer Science. We will use standard nesC and TinyOS development implementation techniques as specified in the documentation [3]. Any new syntax created within the extension nesT will be handled within our new compiler.. The staging deployment in our application relies on wireless communication from the hub to the sensors. The leveraging for this “over air programming” will come from Deluge [8], which is the current prevailing method to disseminate code to wireless sensors. Implementation of the application will be tested on the wireless sensor network within the University of Vermont on the third floor of the Votey building.

Interpretation of results:

The results will be analyzed by teammates at University of Vermont and John Hopkins University to test the results of the work. The efficiency of the new extension will be observed by the empirical data obtained through various benchmarks. We will observe the reduction in the code footprint as we combine compilation, shipping the code over the network, and building with the boot loader process into one application. Memory consumption can be easily measured by comparing the values of usage before and after the implementation. Finally, the program abstraction will be compared with past programs to test the ease of the use. Researchers and developers from academia, industry, and government from within the embedded software development community will also be able to interpret the results reported in publications and technical reports.