Automated Application-Specific Tuning of Parameterized Sensor-Based Embedded System Building Blocks

Susan Lysecky[1] and Frank Vahid[2]

[1]Department of Electrical and Computer Engineering
University of Arizona
Tucson, AZ 85721
slysecky@ece.arizona.edu

[2]Department of Computer Science and Engineering
University of California, Riverside
Riverside, CA 92521
vahid@cs.ucr.edu
Also with the Center for Embedded Computer Systems at UC Irvine

Abstract. We previously developed building blocks to enable end-users to construct customized sensor-based embedded systems to help monitor and control a users' environment. Because design objectives, like battery lifetime, reliability, and responsiveness, vary across applications, these building blocks have software-configurable parameters that control features like operating voltage, frequency, and communication baud rate. The parameters enable the same blocks to be used in diverse applications, in turn enabling mass-produced and hence low-cost blocks. However, tuning block parameters to an application is hard. We thus present an automated approach, wherein an end-user simply defines objectives using an intuitive graphical method, and our tool automatically tunes the parameter values to those objectives. The automated tuning improved satisfaction of design objectives, compared to a default general-purpose block configuration, by 40% on average, and by as much as 80%. The tuning required only 10-20 minutes of end-user time for each application.

1 Introduction

Silicon technology continues to becomes cheaper, smaller, and consume less power, following Moore's Law. This trend has not only enabled new complex computing applications such as military surveillance, health monitoring, and industrial equipment monitoring using what is commonly referred to as sensor networks [16], but opens up numerous possibilities for lower complexity applications within the embedded system domain. Such applications in the home might include a system to monitor if any windows are left open at night, an indicator to alert a homeowner that mail is present in the mail box, or an alarm that detects if a child is sleepwalking at night. In the office, employees may monitor which conference rooms are available, track

temperatures at various locations within the building, or wirelessly alert a receptionist away from his/her desk. Furthermore, scientists may setup a system to activate a video camera at night when motion is detected near an animal watering hole, or to monitor weather conditions over several weeks. Numerous possible examples exist that span varied domains, professions, and age groups. In this paper, an end-user is an individual developing a sensor-based computing application, such as a homeowner, teacher, scientist, etc., who does not have programming or electronics expertise.

With so many application possibilities, why aren't these sensor-based systems more prevalent? The reason is that creating customized embedded systems today requires expertise in electronics and programming. For example, a homeowner may want to create a seemingly simple system to detect if the garage door is left open at night. He would first need to figure out how to detect nighttime and would thus need a light sensor. However, searching for "light sensor" in popular parts catalogs [4,9,15] will not yield the desired results. Instead, the homeowner would need a light dependent resistor or photoresistor, along with a handful of resistors, an opamp, and/or transistors, depending on the specific implementation. Figuring out how to connect these components will require reading a datasheets and schematics. Next, he would need a power supply, and must consider voltage levels, grounding principles, and electric current issues. The homeowner would also need to determine what type of sensor to use to detect if the garage door is open, to implement wireless communication (the homeowner probably doesn't want a wire running from the garage to an upstairs bedroom or kitchen), to program microprocessors to send packets to conserve power, and so on. The seemingly simple garage-open-at-night system actually requires much expertise to build. Alternatively, an engineer could be hired to build a custom system, but the cost is seldom justifiable. Off-the-shelf systems [8,18] provide another option, but highly specific systems tend to be expensive due to low volume. Also, if the desired functionality is not found (e.g., a a system for two garage doors), customizing the system can be difficult or impossible.

We aim to enable end-users with no engineering or programming experience to build customized sensor-based systems. Our approach is to incorporate a tiny cheap microprocessor with previously passive devices. We incorporate a microprocessor with buttons, beepers, LEDs (light emitting diodes), motion sensors, light sensors, sound sensors, etc., along with additional hardware, such that those devices can simply be connected with other devices using simple plugs. Interfacing to hardware and communication between blocks is already incorporated within each individual block. We refer to such devices as *eBlocks* – electronic blocks – which we developed in previous work [2,5]. eBlocks eliminate the electronics and programming experience previously required to build sensor-based systems. The user-created block connectivity determines the functionality of the system, as shown in Figure 1. Furthermore, because the same blocks can be used in a variety of applications, high volume manufacturing results in low block costs of a few dollars or less.

The variety of application possibilities results in a variety of application objectives. For example, one application may require high responsiveness and reliability, whereas another application may require long battery lifetime. One way to support the variety of objectives is to include software-configurable parameters in each eBlock. Thus, the same eBlock may operate at any of several voltage levels and frequencies, may communicate using any of several baud rates, may utilize any of

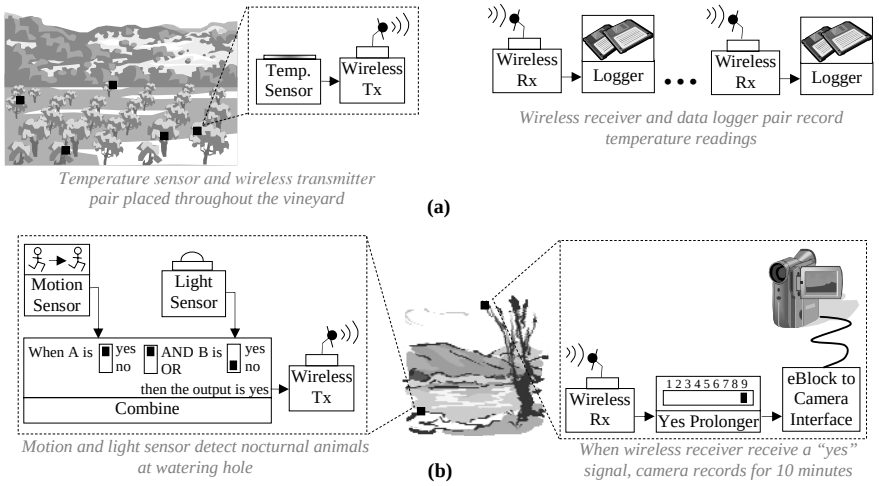

Figure 1: Sample applications built with eBlocks, (a) vineyard weather tracker and (b) endangered species monitor.

several error detection/correction schemes, and so on, depending on the configuration settings in software. An end-user could then tune a block's parameter values to optimize for particular design objectives. Existing sensor-based block platforms contain many such parameters [3,7,20]. Some parameters correspond to hardware settings, others to low-level software settings (such as low-layer network settings, sleep-mode settings, etc.). Other parameters may involve higher-level software settings, such as algorithmic-level choices impacting compression schemes or data aggregation methods. In this paper, we focus on the hardware and low-level software parameters, as those parameters most directly enable mass-produced blocks.

However, tuning a block's parameter values to an application's design objectives is hard, beyond the expertise of most end-users. A block's parameter space may consist of billions of possible configurations, and those parameters are heavily interdependent. Yet careful tuning of those parameters can have a large impact on design objectives. Adlakha et al [1] showed the impact and relationship of the parameters of a block's shutdown scheme, network routing algorithms, and data compression schemes. Yuan and Qu [21] showed the relationship and impact of the parameters of processor type, encryption/decryption algorithms, and dynamic voltage scaling. Tilak et al [19] studied the impact of the parameters of sensor capability, number of sensors deployed, and deployment strategy (grid, random, and biased deployment) on design metrics of accuracy, latency, energy, throughput, and scalability. Heinzelman et al [6] showed the energy impacts of the parameters of different communication protocols, transmit/receive circuitry, message size, distance between blocks, and number of intermediate blocks. Martin et al [14] considers the effects of number of sensors and sampling rate on the accuracy and power consumption. Shih et al [17] examined the impact of different protocols and algorithms on energy consumption, including use of dynamic voltage scaling and sleep states. Some research on block synthesis [13] has appeared, emphasizing the

different but possibly complementary problem of mapping an application's behavioral description onto a fixed or custom designed network of blocks.

Many of these previously researched parameters can be incorporated into a block as software-configurable parameters. Most previous works have only studied the parameters and then indicated the need for careful tuning. In this paper, we present a first approach to automating the support of such tuning. Essentially, our approach represents employment of established synthesis methodology to a problem until now investigated primarily as a networking problem. We refined the synthesis methods, especially that of objective function definition, to the problem. The contribution of the work is in enabling end-users, without engineering experience, to straightforwardly define design objectives, through our introduction of an intuitive graphical objective function definition approach for use by end-users, and our development of fast methods to automatically tune parameters according to those functions. The net result is that these block-based embedded computing systems can better satisfy end-user requirements on battery lifetime, reliability, and responsiveness.

We have also developed complementary computer-based tools that automatically generate an optimized physical implementation of an eBlock system derived from a virtual system function description [12]. End-users are able to specify optimization criteria and constraint libraries that guide the tool in generating a suitable physical implementation, without requiring the end-user to have prior programming or electronics experience. In contrast, this paper considers the resulting physical implementation and automatically tunes software parameters to meet high-level goals such as lifetime or reliability.

Section 2 of this paper provides an overview of our approach. Sections 3, 4 and 5 describe our approach's steps of block characterization, application characterization, and exploration/feedback. Section 6 highlights results of experiments using our prototype tool implementing the approach.

2 Approach Overview

Figure 2 provides an overview of our proposed approach for tuning a parameterized block to an application's design objectives. A block designer provides a block configuration tool, including pre-characterization of the block parameters, as a support tool to the end-user, along with other support tools like programming and debug environments (such as the TinyOS and NesC environments provided with a particular sensor block type [7]). An end-user characterizes application design objectives to the tool by modifying the default objectives, and then asks the tool to tune parameters to the objectives. The tool applies an exploration heuristic and finds parameter values best satisfying the objectives. Based on the values, the end-user may choose to modify the objectives – in case not all objectives could be met, the end-user may wish to modify the objectives – resulting in an iterative use of the tool. Once the end-user is satisfied, the tool outputs a set of parameter values (known as a configuration) for the blocks. The block support tools download those parameter values into the blocks, along with block programs and possible data, and those values configure the block's hardware and software components upon startup/reset of the

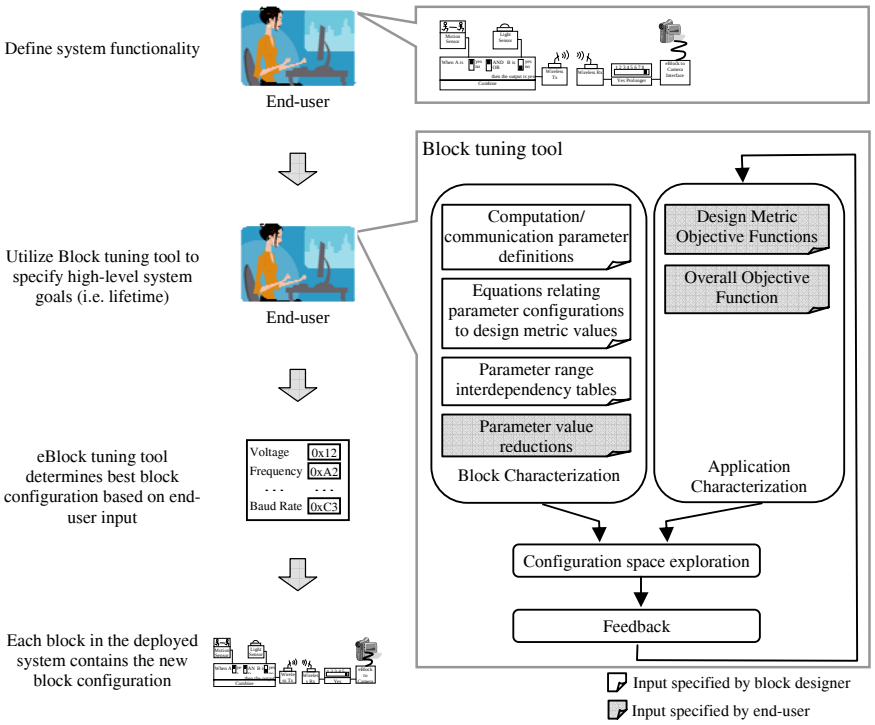

Figure 2: Block tuning tool overview. A block designer performs block characterization once. An end-user performs application characterization by customizing objective functions, optionally refines the block characterization by reducing possible parameter values, and then executes the parameter space exploration heuristic. The tool provides feedback on objective function achievement, based on which the end-user may choose to refine objective functions and re-iterate. Once done, the tool incorporates the chosen parameter values into the block's startup/reset software.

block in a deployed network. Presently, all blocks would have the same configuration, but future directions may support different configurations for different blocks.

We now describe the approach's parts in more detail, and indicate how we addressed each part in our prototype tool. We developed the tool with eBlocks in mind, but the approach can be applied and/or generalized for other block types.

3 Block Characterization by the Block Designer

A block designer must characterize the block for the block-tuning tool. Such characterization consists of creating three items: computation/communication parameter definitions, equations relating parameter configurations to design metrics,

and a parameter interdependency description. Note that these items are created by a block designer, who is an engineering expert, and *not* by end users.

3.1 Computation/Communication Parameter Definitions

The block designer must define the list of block parameters and the possible values for each parameter. The physical block we used supported the following parameters:

- Microcontroller Supply Voltage (V) = {3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 5.0, 5.1, 5.2, 5.3, 5.4, 5.5}
- Microcontroller Clock Frequency (Hz) = {32k, 100k, 200k, 300k, 400k, 455k, 480k, 500k, 640k, 800k, 1M, 1.6M, 2M, 2.45M, 3M, 3.6M, 4M, 5.3M, 6M, 7.4M, 8M, 8.192M, 9M, 9.216M, 9.8304M, 10M, 10.4M, 11.06M, 12M, 13.5M, 14.74M, 16M, 16.384M, 16.6666MHz, 17.73M, 18M, 18.432M, 19.6608M, 20M}
- Communication Baud Rate (bps) = {1200, 2400, 4800, 9600, 14.4K, 28.8K}
- Data Packet Transmission Size = {4 bits, 1B, 2B, 4B}
- Data Timeout = {0.2s, 0.3s, 0.4s, 0.5s, 0.6s, 0.7s, 0.8s, 0.9s, 1s, 1.25s, 1.50s, 1.75s, 2s, 2.5s, 3s, 3.5s, 4s, 4.5s, 5s, 6s, 7s, 8s, 9s, 10s, 20s, 30s, 40s, 50s, 1m, 2m, 3m, 4m, 5m, 6m, 7m, 8m, 9m, 10m, 15m, 20m, 25m, 30m}
- Alive Timeout = {0.1s, 0.2s, 0.3s, 0.4s, 0.5s, 0.6s, 0.7s, 0.8s, 0.9s, 1s, 1.25s, 1.50s, 1.75s, 2s, 2.5s, 3s, 3.5s, 4s, 4.5s, 5s, 6s, 7s, 8s, 9s, 10s, 20s, 30s, 40s, 50s, 1m, 2m, 3m, 4m, 5m, 6m, 7m, 8m, 9m, 10m, 15m, 20m, 25m}
- Error Check/Correct (ECC) Strategy = {none, crc, parity, checksum1, checksum2, hamming1, hamming2}

The supply voltage, clock frequency, and baud rate possible values came from the databook of the physical block's microcontroller, in this case a PIC device, and were all software configurable in the physical block.

The data packet size, data timeout, alive timeout, and error check/correct strategies were all user-specified settings in the basic support software of the physical block. Data packet size is the number of bits in a data packet – in our block, the choice impacted the range of integers transmittable. Data timeout is the maximum time between successive data packets – shorter timeouts result in faster responsiveness as blocks are added/removed to/from a network. Alive timeout is the maximum time between short (and hence low power) "I'm alive" messages used by the blocks to indicate that the block is still functioning, again impacting responsiveness. The error checking/correcting (ECC) strategies are extra bits placed in data packets to detect or forward-error-correct incorrectly received bits. The *parity* strategy uses sends a single parity bit for every 8-bit data packet. The *crc* strategy transmits an extra packet containing the remainder of the data packets and a CRC-3 generating polynomial. The *checksum1* strategy transmits an extra packet consisting of the corresponding sum of the data packets. The *checksum2* ECC strategy negates the extra packet before transmission. The *hamming1* ECC strategy considers the data as a matrix and generates row and column parity error checking packets. The *hamming2* strategy embeds parity bits within the packets at every power of two-bit location (bit 1, 2, 4, etc.). All these methods are standard methods.

A set of values, one for each parameter, defines a block configuration. We presently require the block designer to explicitly list possible values for a parameter. A similar method would allow a block designer to specify the value range along with the step size between successive values for a parameter. However, a block designer must be careful to avoid introducing unnecessarily-fine granularity to a parameter's values, as such granularity increases the configuration space to be explored by the tool, and may increase the number of interdependency tables (discussed in the next section). For the same reasons, we require that the block designer explicitly quantize a parameter's possible values, rather than merely specifying the parameter's range.

3.2 Parameter Range Interdependency Tables

Not every parameter configuration is valid. For example, a particular voltage setting may limit the range of possible clock frequencies. Likewise, a particular frequency may limit the range of possible baud rates. A block designer indicates such interdependencies using tables. For a given parameter, the block designer may optionally create a table providing, for any parameter value, lower and upper bounds for any other parameter. The exploration tool will use these tables to exclude invalid configurations from consideration.

3.3 Equations Relating Parameter Configurations to Design Metric Values

The block configuration tool must map a given parameter configuration to specific values for each design metric supported by the tool. Our tool presently supports the design metrics of lifetime, reliability, block latency, connect responsiveness, and disconnect responsiveness, defined in Section 4.1. We derived equations from datasheet information, textbooks, and previous findings, and thus we do not claim those equations as a contribution of this work. Highly accurate equations can become rather complex if all parameter values are carefully considered. While verifying and improving the accuracy of those equations is an important direction of investigation, that direction is largely orthogonal to the development of our overall methodology.

3.4 Parameter Value Reductions

A block designer performs the three above-described block characterization subtasks only once, and then incorporates the characterizations into the tool. A fourth, optional, block characterization subtask may be performed by an end-user to reduce the number of possible configurations and hence speedup the exploration step. In this fourth task, the end-user reduces the number of possible values for a given parameter, either by restricting the parameter's range, or by reducing the granularity of steps between successive parameters. Our present tool allows the end-user to manually exclude

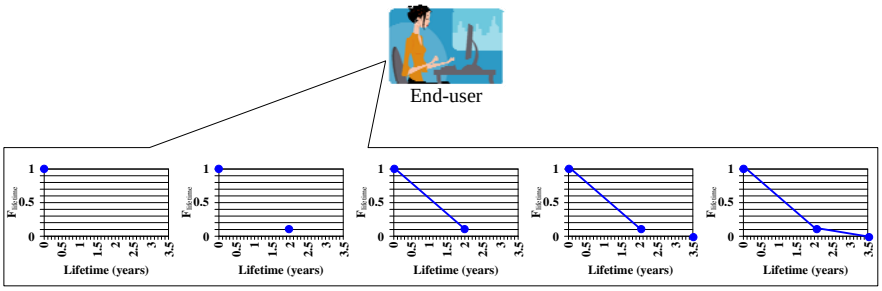

Figure 3: End-user specifies the "goodness" of a lifetime value by assigning normalize values between 0 (best) and 1 (worst) to various lifetimes. (a) End-user determines a lifetime of 0 years is inadequate, sets goodness to 1, (b) lifetime of 2 years is adequate, sets goodness to 0.1, (c) intermediate goodness are automatically determined by the tool, (d) lifetime of 3.5 meets system requirements, sets goodness to 0, (e) intermediate goodness determined by tool.

particular parameter values from consideration. However, the tool does not allow the end-user to add new values, because such new values would require new range interdependency tables, and because the equations mapping configurations to design metric values might not be valid for new values outside the range defined by the block designer. If an end-user deletes all but one possible value for a parameter (the tool requires that at least one value remain for each parameter), that parameter ceases to act as a parameter during exploration, being fixed at the chosen value.

4 Application Characterization by the End User

The previous section discussed block characterization, a job performed once by a block designer, and incorporated into the block configuration tool. Different end-users will then use this tool to tune the block to different applications. The end-user must characterize the application for the tool so that the tool can tune to that application. Thus, application characterization will be performed many times.

Application characterization consists of specifying the *design metric objective functions* and specifying the *overall objective function*. Our tool provides default functions targeted to general-purpose block use, so the end-user can merely customize particular functions that should deviate from the defaults.

4.1 Design Metric Objective Function

A design metric objective function maps a design metric's raw value to a normalized value between 0 and 1 representing the "goodness" of the value, with 1 being the worst and 0 being best. An end-user specifies a design metric objective function for each design metric by defining the range of the X-axis and then by drawing a plot that

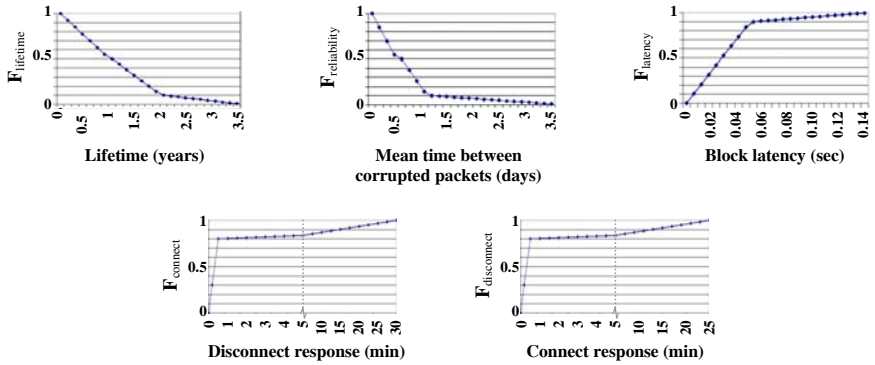

Figure 4: Default design metric objective functions for general purpose block usage.

maps each X-axis value to a value between 0 and 1, as shown in Figure 3. Our present tool captures the function as a table rather than plot, but the concept is the same. The end-user currently captures "goodness" as a piecewise linear function, however, the end-user can ideally specify "goodness" in any format desired (e.g. linear, quadratic, exponential) limited only by the formats supported by the capture methodology.

Our tool presently supports capture of the following design metric objective functions:

- Lifetime – the number of days a block can run powered by the block's battery.
- Reliability – the mean time in days between undetected corrupt data packets.
- Block Latency – the time in seconds for a single block to process an input event and generate new output.
- Connect Response – the time for newly connected blocks to receive good input and behave properly.
- Disconnect Response – the time for newly disconnected blocks to behave as disconnected.

Initially, in developing eBlocks, the above design metrics were most relevant. However, more design metrics are certainly possible. An end-user developing systems intended to monitor a battlefield for troop movements would likely be interested a design metric describing security. Alternatively, an end-user developing systems intended to process video or audio data would likely be interested in a design metric describing throughput of the system. The end-user determines which designs metric are important for their particular application and chooses which design metrics they want to consider in determining a block configuration.

Figure 4 illustrates example definitions of objective functions for these design metrics. The functions correspond to the default functions that our tool provides, intended for a general application. For example, the function for lifetime indicates that a lifetime of 0 years is the worst possible and of 2 years is nearly the best possible, with linear improvement in between. Lifetime improvements from 2 years to 3.5 years are only slightly better than 2 years, and improvements beyond 3.5 years are not important according to the function. As another example, the function for mean time between corrupted packets indicates goodness improvement from 0 days to 1 day,

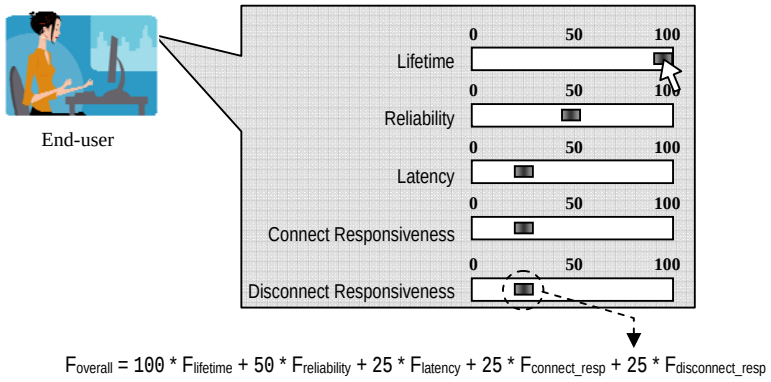

Figure 5: End-users utilize a web-based interface to specify the relative importance of each design metric. Individual weights are assigned by setting the corresponding slide switch to the desired position. Each design metric weight is then combined into an overall objective function.

with reduced improvement beyond 1 day. The function for block latency says that 0 second latency is the best, quickly degrading up to 0.06 seconds. Latencies beyond 0.06 seconds are very bad. Although all the functions shown are piecewise linear, the functions can also be non-linear.

Providing the ability to custom define each design metric's objective function is an important part of our approach. We observed that traditionally-used standard functions, such as those based on mean-square error, penalty, or barrier functions [11], do not readily capture the end-user's true intent – Figure 6, discussed later, illustrates this point.

Notice that the end-user need not know how those metrics are related to a block's parameters, and in fact need not even be aware of what block parameters exist. The end-user merely customizes the design metric objective functions.

4.2 Overall Objective Function

The end-user also configures an *overall objective function*, which captures the relative importance of the individual design metrics in a function that maps the individual metric values to a single value. We currently define the default overall objective function as a weighted sum of the individual design metric normalized values:

$$F_{overall} = (A * F_{lifetime}) + (B * F_{reliability}) + (C * F_{latency}) + (D * F_{connect\_resp.}) + (E * F_{disconnect\_resp.})$$

The end-user customizes the values of the constants A, B, C, D and E to indicate the relative importance of the metrics, as shown in Figure 5. A more-advanced end-user can instead use a spreadsheet-like entry method to redefine the overall objective function as any linear or non-linear function of the design metric function values.

A key feature of our approach is the separation of defining design metric objective functions, and weighing those functions' importance in an overall objective function. This separation enables an end-user to focus first on what metric values are good or bad – e.g., a lifetime below 6 months is bad for one application, below 2 years for another application – and then separately to focus on the relative importance of those metrics – e.g., lifetime may be twice as important as latency in one application, but one third as important for another application. Hence, the orthogonality of the definition of metric values as good or bad, and of the relative importance of metrics, is supported by our approach.

5 Configuration Space Exploration and Feedback

Configuration space exploration searches the space of valid parameter values for a configuration that minimizes the value of the overall objective function. For each possible configuration, the exploration tool applies the block designer specified design metric evaluation equations to obtain raw values for each design metric, which the tool then inputs to the end-user specified design metric objective functions to obtain normalized values, and which the tool finally combines into a single value using the end-user specified overall objective function.

One search method is exhaustive search, which enumerates all possible parameter value configurations and chooses the configuration yielding the minimized objective function value. For the parameter ranges and values defined earlier in the paper, the search space (after pruning invalid configurations caused by parameter interdependencies) consists of over 100 million configurations. Searching that space exhaustively is feasible, requiring 3-4 minutes on a 3 GHz Pentium processor. However, for blocks with more parameters or more values, exhaustive search may be infeasible. We thus investigated faster methods.

As our parameter search problem resembles an integer linear program, we considered integer linear program solution methods (optimal or heuristic), but a problem is that such an approach limits the objective functions to linear functions. Instead, an end-user might desire a non-linear function, to greatly penalize values over a certain amount for squaring, for example.

We also considered greedy or constructive approaches that used some knowledge of the problem structure to efficiently traverse the search space. However, we sought to keep the exploration tool independent of the particular block parameters and objective functions. Greedy or constructive heuristics that don't consider problem structure may perform poorly. However, the block-designer-specified equations and parameter interdependency descriptions can improve the design of such heuristics. We leave this direction for future investigation.

Ultimately, we chose to use an iterative improvement approach, namely the simulated annealing heuristic [10]. The heuristic has the advantage of being independent of block parameters and objective functions. Furthermore, the heuristic provides a simple means for an end-user to tradeoff exploration time with optimization amount. The end-user can indicate allowable runtime, from which the tool can derive an appropriate annealing cooling schedule. We presently utilize a

cooling schedule that executes for just a few seconds on a 3 GHz Pentium, while yielding near-optimal solutions. The time complexity of the simulated annealing heuristic is in general not known, depending heavily on the cooling schedule and problem features. Yet in practice, a specific cooling schedule yields roughly similar runtimes for the same general problem, as occurred in our case.

For the chosen best configuration, the tool provides feedback to the end-user in two forms. One form is the value of the overall objective function and the relative contribution of each design metric objective function value to the overall value end-user specified design metric objective function, for each design metric. Based on this information, the end-user may actually choose to refine his/her design metric or overall objective function definitions, iterating several times until finding a satisfactory configuration.

The block configuration tool converts the final configuration into software that appropriately fixes the sensor block parameters to the configuration's values. The tool achieves such fixing primarily by setting constant values for global variables uses by the microcontroller's startup/reset code. Many of those global variables actually correspond to special microcontroller or peripheral built-in registers, such a microcontroller registers that select clock frequency or baud rate, and a register in a digital voltage regular that controls supply voltage to the microcontroller. Other variables are used by software routines to choose among data structures and/or functions, such as for the error check/correct routine.

6 Experiments

We implemented our approach in a prototype tool, consisting of 8,000 lines of Java code, and interfacing with Excel spreadsheets to support equation capture and plot displays. We considered four different applications, all but the "Vineyard" example being derived from actual projects involving the physical blocks. Those applications' design metric objective functions appear in Figure 6.

The *Educational Science Kit* application utilizes eBlocks to introduce middle-school students to simple engineering concepts. Students combine and configure blocks to create customized sensor-based embedded systems in their classrooms. For the purposes of this paper, the students are *not* the end-users – rather, the end-user is the person putting eBlocks into the kit for student use. Acting as that end-user, we defined the design metric objective functions shown in Figure 6(a). The reliability function (mean time between corrupted packets) differs from its general case version in Figure 4, as reliability is less important because the systems being built in classrooms don't monitor or control important situations. In contrast, the block latency and connect response functions both demand higher performance than for the general case, as the students basic usage of the blocks will involve repeated adding/deleting of blocks, and students might be confused by long latencies or slow response. For the overall objective function, we weighed lifetime with 0.1 (unimportant), reliability with 0.5, throughput with 0.5, connect response with 1, and disconnect response with 1 (important).

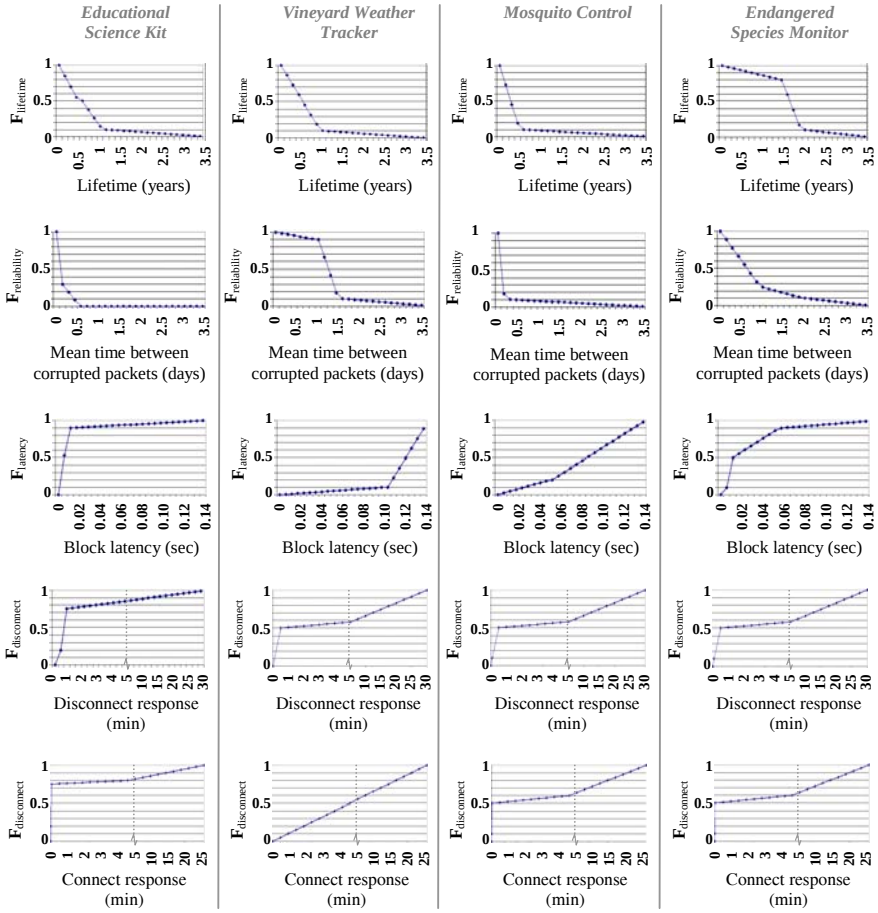

Figure 6: End-user defined design metric objective functions for four different applications.

The *Vineyard Weather Tracker* application is a long-life application deployed in a vineyard to track temperature, rainfall, and average hours of sunlight. Compared to the general case of Figure 4, Figure 6(b) shows that longer latency is acceptable because the items being monitored are not rapidly changing, and that slower disconnect and connect responses are also acceptable as blocks won't be disconnected/connected frequently. For the overall objective function, we weighed lifetime with 1 (important), reliability with 0.5, latency with 0.5, disconnect response with 0.1, and connect response with 0.1 (unimportant).

The *Mosquito Control* application reads data from a mosquito trap and meters out insecticide accordingly. Figure 6(c) shows that lifetime beyond 6 months is not necessary because the mosquito season lasts only 6 months, after which all blocks will be reclaimed and stored, with all batteries replaced the following season. We weighed

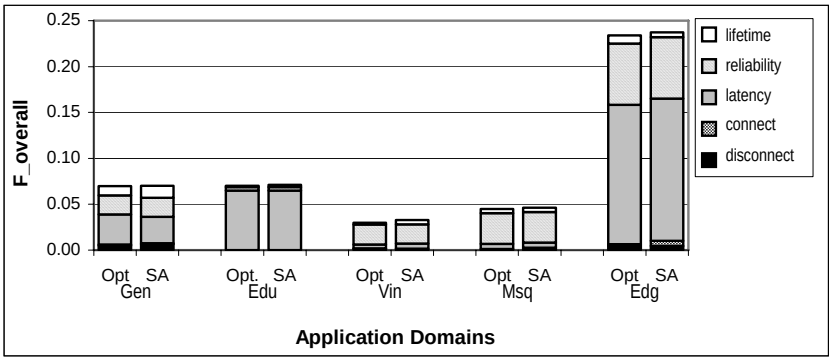

Figure 7: Normalized overall objective function results comparing the various application configurations obtained utilizing exhaustive search (Opt) verses simulated annealing (SA): General (Gen), Educational Science Kit (Edu), Vineyard Weather Tracker (Vin), Mosquito Control (Msq), and Endangered Species Monitor (Edg).

lifetime with 0.5, reliability with 1, latency with 0.5, and disconnect/connect responses with 0.2. The weights indicate that reliability is most important, as improper output of insecticide should be avoided.

The *Endangered Species Monitoring* system detects motion near a feeding site and video-records the site for a specified duration, for later analysis by environmental scientists estimating the population of endangered species. Figure 6(d) shows that the key difference from the general case is that lifetime less than 1.5 years is unacceptable, as the feeding site is in a remote location that is hard to access, and thus batteries should not have to be replaced frequently. We assigned lifetime a weight of 1, reliability 0.8, latency 0.8, and disconnect/connect responses weights of 0.1 each.

Using our tools, characterized each applications requires only 10 minutes.

For each application, we executed our automated tuning tool, for both the end-user application characterizations in Figure 6, and the general case of Figure 4. To verify that the tool was effectively finding good configurations, we also executed exploration using exhaustive search. The tool's execution time averaged 10 seconds per application, while exhaustive search averaged over 3.5 minutes, both methodologies running on the same 3 GHz Pentium computer. Figure 7 summarizes

Table 1: Configurations achieved by heuristic exploration for various applications. Numbers in parentheses indicate values obtained by exhaustive search, where those values differed from heuristically-obtained values.

	Applications				
	General	Vineyard Weather Tracker	Educational Science Kit	Mosquito Control	Endangered Species Monitoring
Voltage (V)	3.1 (3)	4.8 (3.0)	3.5 (3.0)	3.1 (3.0)	3.1 (3.0)
Frequency (MHz)	2.45	11.06 (9)	2 (1.6)	2 (0.64)	3 (2.45)
Baudrate	9600	14400	4800	9600	9600
Data Packet (bytes)	4 (2)	1	2	1	2 (4)
Data Timeout (sec)	1.25 (1)	0.2	20	1.75 (0.9)	7
Alive Timeout(sec)	0.3	0.1	2.5 (5)	0.1	0.7 (0.3)
ECC Strategy	crc	none	hamming1	none	hamming2

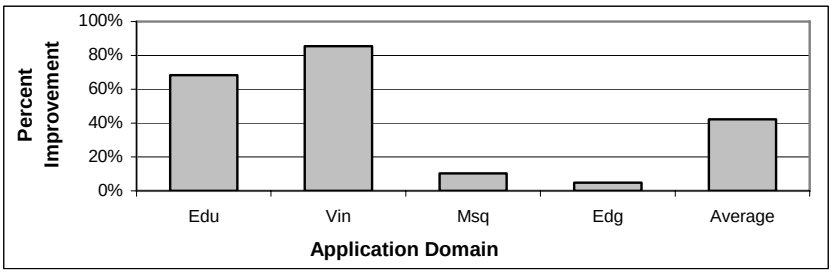

Figure 8: Percent improvement in overall objective function utilizing customized block configuration verses general block configuration across various applications: Educational Science Kit (Edu), Vineyard Weather Tracker (Vin), Mosquito Control (Msq), and Endangered Species Monitor (Edg).

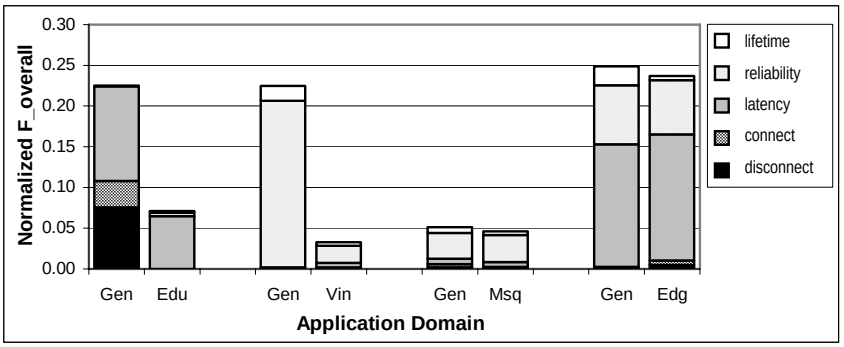

Figure 9: Normalized overall objective function results comparing the general configuration (Gen) verses the application specific configuration across various applications: Educational Science Kit (Edu), Vineyard Weather Tracker (Vin), Mosquito Control (Msq), and Endangered Species Monitor (Edg).

results, showing that the tool's simulated annealing heuristic found near optimal results for all the applications. The figure also shows the relative contribution of each design metric objective function to the overall objective function, showing similar achievements between heuristic and exhaustive exploration. Table 1 shows the specific configurations found by the heuristic and exhaustive exploration methods. The differing values obtained by the two methods show that different parameter configurations can yield similar overall objective function values.

Figure 8 shows the more important results that compare the use of the general block configuration to the configuration obtained through our tuning approach (using the heuristic exploration). The results show that the general block configuration works well for the latter two applications (Mosquito Control, and Endangered Species Monitor). However, the general configuration does not work well for the first two applications (Educational Science Kit, and Vineyard Weather Tracker) – tuning significantly improves the overall objective function value for those applications.

Figure 9 summarizes the percent improvement of overall objective function values for the tuned blocks compared to the generally configured blocks.

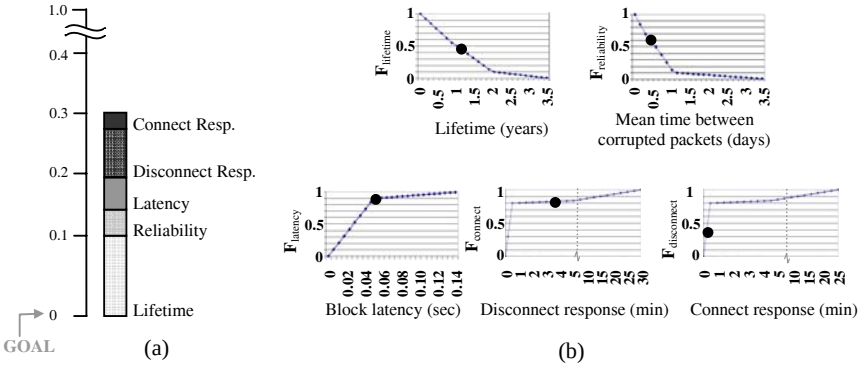

Figure 10: Goal of the tool is to minimize the overall objective function value. Tool provides feedback of the tool to the end-user after exploration illustrating the configuration's (a) overall objective function value achieved along with the relative contribution of each design metric and (b) the raw values and their mapping to normalized values for each design metric.

Figure 10 illustrates the type of feedback provided by the block-tuning tool to the end-user. Figure 10(a) provides the overall objective function value achieved by exploration, along with the relative contribution of each design metric to that value. Further, Figure 10(b) shows the configuration's raw values and their mapping to normalized values for each design metric. Note that the overall objective function value is not just a sum of the normalized design metric values, because of weights assigned by the end-user in the overall objective function. Based on the feedback, the end-user may decide to refine a particular design metric objective function, perhaps deciding that tolerating poorer performance on one metric (e.g., connect response) is acceptable in the hopes of improving another metric (e.g., lifetime). Alternatively, the end-user might modify the weights in the overall objective function.

Notice that the end-user need not have any awareness of what parameters exist on the block (e.g., voltage, baud rate), nor of the values of those parameters for a particular configuration (e.g., 3 V, 2400 baud). The tuning approach instead presents the end-user with an abstraction that only deals with objective functions. Such abstraction enabled the end-user to perform all necessary tuning steps, including application characterization, exploration, and feedback analysis, in just 10 minutes.

7 Conclusions

We presented an approach that enables end-users to automatically tune parameterized building blocks to meet end-user defined application goals such as battery lifetime, reliability, responsiveness, and latency. The block tuning approach consisted of the key steps of block characterization by the block design, and then application characterization, exploration, and feedback involving the end user. Our approach

provides an abstraction of the block to the end-user such that the end-user need only deal with characterizing the application through definition of intuitive graphical objective functions, requiring on the order of 10 minutes for a given application. The objective function definition approach separates definition of individual design metric objective values from definition of the relative importance among those metrics. Furthermore, experiments show that our tool can tune blocks in just a few seconds to near-optimal values, and that the tuned blocks exhibit greatly superior performance for two of the four applications we examined, compared to a block configured for general-purpose use. Our work represents use of established synthesis methodology, with some refinement, to a problem considered primarily in the networking domain. The work's contribution is in enabling end-users with domain experience, but without engineering experience, to effectively utilize mass-produced computing blocks intended to monitor and control the user's environment.

Future work will involve expanding the parameters and design metrics supported by the tool, requiring careful attention to design of accurate evaluation equations. Another direction involves allowing an end-user to characterize the network structure and environment, in which case the tuning tool might determine different configurations for different blocks in the network. Future directions involve higher-level parameters relating to algorithmic and high-layer networking choices.

8 Acknowledgments

This work is supported in part by the National Science Foundation under grant CCR-0311026. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

9 References

1. Adlakha, S., S. Ganeriwal, C. Schurger, M. Srivastava. Density, Accuracy, Latency and Lifetime Tradeoffs in Wireless Sensor Networks – A Multidimensional Design Perspective. Embedded Network Sensor Systems, 2003.
2. Cotterell, S., F. Vahid. Usability of State Based Boolean eBlocks. International Conference on Human-Computer Interaction (HCII), July 2005.
3. Cotterell, S., F. Vahid, W. Najjar, H. Hsieh. First Results with eBlocks: Embedded Systems Building Blocks. CODES+ISSS Merged Conference, October 2003.
4. Digikey, http://www.digikey.com, 2006.
5. eBlocks: Embedded Systems Building Blocks. http://www.cs.ucr.edu/~eblock
6. Heinzelman, W., A. Chandrakasan, H. Balakrishnan. Energy-Efficient Communication Protocols for Wireless Microsensor Networks. Hawaii International Conference on System Sciences, 2000.
7. Hill, J., D. Culler. MICA: A Wireless Platform For Deeply Embedded Networks. IEEE Micro, Vol. 22. No. 6, November/December 2002.
8. Home Heartbeat, http;//www.homeheartbeat.com, 2006.
9. Jameco, http://www.jameco.com, 2006.

10. Kirkpatrick, S., C. Gerlatt, M. Vecchi. Optimization by Simulated Annealing, Science 220, 671-680, 1983.
11. Lopez-Vallejo, M., J. Grajal, J. Lopez. Constraint-driven System Partitioning. Design Automation and Test in Europe, 2000.
12. Lysecky, S., F. Vahid. Automated Generation of Basic Custom Sensor-Based Embedded Computing Systems Guided by End-User Optimization Criteria. UbiComp, 2006.
13. Mannion, R., H. Hsieh, S. Cotterell, F. Vahid. System Synthesis for Networks of Programmable Blocks. Design Automation and Test in Europe, 2005.
14. Martin, T., M. Jones, J. Edmison, R. Shenoy. Towards a design framework for wearable electronic textiles. IEEE International Symposium on Wearable Computers, 2003.
15. Mouser, http://www.mouser.com, 2006.
16. National Research Council. Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers. National Academies Press, 2001.
17. Shih, E. S. Cho, N. Ickes, R. Min, A. Sinha, A. Wang, A. Chandrakasan. Physical Layer Driven Protocol and Algorithm Design for Energy-Efficient Wireless Sensor Networks. International Conference on Mobile Computing and Networking (MobiCom), 2001.
18. Smart Home, http://www.smarthome.com, 2006.
19. Tilak, S., N. Abu-Ghazaleh, W. Heinzelman. Infrastructure Tradeoffs for Sensor Networks. Int. Workshop on Wireless Sensor Networks and Applications, 2002.
20. Warneke, B., M. Last, B. Liebowitz, and K. Pister. Smart Dust: Communicating with a Cubic-Millimeter Computer. Computer Magazine, pg. 44-51, January 2001.
21. Yuan, L., G. Qu. Design Space Exploration for Energy-Efficient Secure Sensor Network. Conf. on Application-Specific Systems, Architectures, and Processors, 2002.