Synthesizing Component-Based WSN Applications via Automatic Combination of Code Optimization Techniques [*]

Zhenyu Zhang
The University of Hong Kong
Pokfulam, Hong Kong
zyzhang@cs.hku.hk

W. K. Chan
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cs.cityu.edu.hk

T. H. Tse [†]
The University of Hong Kong
Pokfulam, Hong Kong
thtse@cs.hku.hk

Abstract

Wireless sensor network (WSN) applications sense events in-situ *and compute results* in-network. *Their software components should run on platforms with stringent constraints on node resources. Developers often design their programs by trial-and-error with a view to meeting these constraints. Through numerous iterations, they manually measure and estimate how far the programs cannot fulfill the requirements, and make adjustments accordingly. Such manual process is time-consuming and error-prone. Automated support is necessary.*

Based on an existing task view that treats a WSN application as tasks and models resources as constraints, we propose a new component view that associates components with code optimization techniques and constraints. We develop algorithms to synthesize components running on nodes, fulfilling the constraints, and thus optimizing their quality. We evaluate our proposal by a simulation study adapted from a real-life WSN application.

Keywords: Wireless sensor network, adaptive software design, resource constraint, code optimization technique.

1. Introduction

A wireless sensor network (WSN) is a computer network of sensor nodes interconnected by short-range and short-life wireless communication channels [1]. Each sensor node may capture data, such as temperature and light intensity, from the environment. Applications running on WSNs, such as animal surveillances, automatic detections of geological events, and hospital administrations, should sense physical events *in-situ* [8] and analyze the sensed data *in-network* [12].

In WSNs, communication consumes the highest amount of energy in the sensor nodes, followed next by processing and then storage. Akin to design patterns or code refactoring for general object-oriented development, WSN developers use diverse code optimization techniques such as loop unfolding and lookup tables to tune the WSN software applications to meet the resource constraints. They apply different tactics to cater for different needs. This paper will collectively refer to such tactics as *code optimization techniques*, or *COTs* for short.

However, incorporating a code optimization technique in a WSN program currently needs significant manual effort. When an application does not work according to a COT, a simple pragmatic approach is to tune it iteratively and manually by means of trial-and-error. This is tedious, low-level, and time-consuming. Also, the underlying WSN platforms, both hardware and software, are diverse in quality. A seemingly innocuous change may drastically alter the constraints that these programs need to fulfill. The WSN software fit for a specific resource-stringent environment will need to be adapted further to adjust to the changed environment. The lack of a system-wide concept to deal with code optimization techniques further complicates how developers can apply various COTs for different software units.

To tackle these challenges, this paper proposes a task-oriented component-based COT model. It represents a WSN application as a set of components. In the task view, resource constraints, known as *resource concerns* or simply *concerns*, are defined at both the application and node levels. In the component view, every component is associated with its basis resource usages and a set of COTs.

[*] This research is supported in part by a grant of the Innovation and Technology Commission in Hong Kong (project no. ITS/076/06), grants of the Research Grants Council of Hong Kong (project nos. 111107 and 716507), a grant of City University of Hong Kong, and a grant of The University of Hong Kong.

[†] All correspondence should be addressed to Prof. T. H. Tse at Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Tel: (+852) 2859 2183. Fax: (+852) 2858 4141. Email: thtse@cs.hku.hk.

The main contributions of the paper are threefold: Firstly, it proposes an application-level design optimization model for WSN applications. Next, it develops algorithms to construct components that support the automatic selection of a suite of COTs. Thirdly, it provides the first empirical study on the topic.

The rest of this paper is organized as follows: After reviewing the related work in Section 2, we describe a motivation example in Section 3. Section 4 presents our design model and algorithms, followed by an evaluation in Section 5. Finally, Section 6 concludes the paper.

2. Related Work

Many researchers have conducted studies to adapt WSN applications to resource constraints. Kuchcinski [7] synthesizes an embedded system to meet timing constraints. Similarly, Wang and Shin [14] construct tasks to tackle a similar issue with a view to minimizing the overall elapsed time. Other than timing constraints, Teich et al. [13] study the processing capability of partitioned processor arrays. Shin et al. [11] further investigate how to tackle the energy and code size constraints. Their study inspires our work.

In the above work, resource usages are optimized via different techniques including reconfiguration, task construction, code encoding, and compressing. These techniques are specific to different situations and, hence, may adversely affect the modifiability of the applications. On the other hand, Gay et al. [3] implements experimental design patterns in the context of WSNs. This inspires us to use combined code optimization techniques to optimize resource usages to cater for unanticipated fluctuations in environmental constraints. As in [6], code optimization techniques can be embedded into the code similarly to design patterns. A difference between our approach and that of [6] is that we consider aggregated effects of combined code optimization techniques while they do not.

Adopting code optimization techniques is related to program synthesis. In this field, Huselius and Andersson [5] introduce their model synthesis work for real-time systems, which focuses on architectures and observed behaviors. Kuchcinski [7] tackles timing constraints by assigning processes to processors. Our component-based model supports configurations with multiple resources, and we use combined code optimization techniques to optimize their overall usages. A similar concept is also introduced in [15], which only investigates the interaction relationships of optimization techniques but not their aggregated effect.

We treat WSN applications as components. Zhang and Cheng [16] use Petri nets as a model to cater for the design of adaptive behavior, while Sgroi et al. [10] propose a communicating finite state machines model with a similar aim. Their applicability to WSNs is yet to be studied.

```
inline static result_t
TimerM$Timer$fired (uint8_t arg_0xb76cb2c8) {
    unsigned char ret;
    switch (arg_0xb76cb2c8) {
        case 0U:
            ret = SurgeM$Timer$fired();
            break;
        case 1U:
            ret = PhotoTempM$PhotoTempTimer$fired();
            break;
        case 2U:
            ret = AMPromiscuous$ActivityTimer$fired();
            break;
        case 3U:
            ret = MultiHopLEPSM$Timer$fired();
            break;
        default:
            TimerM$Timer$default$fired();
    }
    return ret;
}
```

Figure 1. Timer.fired in Surge.

3. Motivation Example

This section describes a motivation example using the component Timer.fired from Surge,[1] a real-life application of TinyOS.[2] The component, as shown in Figure 1, resides in a task initiated by periodic time-driven events. Let us call this version P_0 for the ease of reference.

In P_0, a switch construct accepts a message-type identifier (parameter arg_0xb76cb2c8) and invokes the corresponding processing functions. To do so, the component needs to compare the value of arg_0xb76cb2c8 with the cases in switch. The mean number of comparison operations, denoted by mean($COMP$), is $\frac{1+2+3+4+252\times 4}{256} \approx 3.977$. This is because, for the uniform distribution of an unsigned 8-bit integer whose range is [0U, 255U], almost all of possible values will fall under the default branch, which means that they should pass through the first four case statements before reaching the default branch. In the worst case, all samples fall into [3U, 255U]. The maximum number of comparison operations, denoted

[1] Available at http://www.tinyos.net/tinyos-1.x/apps/Surge/.

[2] TinyOS, available at http://www.tinyos.net/, is an open-sourced operating system dedicated and widely used for wireless sensor network applications. Surge and Timer.fired are available at http://www.tinyos.net/tinyos-1.x/apps/Surge/.

```
inline static result_t
TimerM$Timer$fired (uint8_t arg_0xb76cb2c8) {
    unsigned char ret;
    if (arg_0x76cb2c8 >= 4U) { // old default
        return TimerM$Timer$default$fired();
    }
    switch (arg_0xb76cb2c8) {
        case 0U:
            ret = SurgeM$Timer$fired();
            break;
        case 1U:
            ret = PhotoTempM$PhotoTempTimer$fired();
            break;
        case 2U:
            ret = AMPromiscuous$ActivityTimer$fired();
            break;
        case 3U:
            ret = MultiHopLEPSM$Timer$fired();
            break;
    }
    return ret;
}
```

Figure 2. Optimized version 1 of Timer.fired.

by max$(COMP)$, is 4.

We observe that this code fragment adopts at least one COT. The variables arg_0xb76cb2c8 and ret as well as the case values 0U, 1U, 2U, and 3U are of the type uint8_t, that is, unsigned 8-bit integer. [3]

Suppose that, owing to the concern of low-end processors in sensor nodes, we plan to reduce the time complexity by reducing mean$(COMP)$. A simple COT is to add an if-then-else construct embracing the switch construct, which decides whether to call the default processing (see Figure 2). We denote this code optimization technique by cot_1 and the optimized version by P_1. The functional behavior of the example does not change after introducing cot_1, while mean$(COMP)$ becomes $\frac{2+3+4+5+252\times 1}{256} \approx 1.039$ and max$(COMP)$ increases to 5.

While COTs may reduce the amount of usage for one resource, they may increase another. Figure 3, for example, shows another version (P_2) that includes another code optimization technique (cot_2) on top of version P_1. cot_2 is designed to remove the time-wasting switch construct. This is achieved by introducing a lookup table to manage the pointers of the corresponding functions. P_2 has the same functionality as P_1 but needs only *one* comparison operation for any arg_0xb76cb2c8, so that mean$(COMP) =$ max$(COMP) = 1$. Still, it consumes an extra statically-allocated memory block whose size is 16 bytes, that is, the size of 4 pointers in a 32-bit environment.

[3] The use of unsigned 8-bit integer variables is a general code optimization technique for embedded applications to produce executable files of smaller sizes.

```
typedef (unsigned char)(*FuncEntry)(void);
inline static result_t
TimerM$Timer$fired (uint8_t arg_0xb76cb2c8) {
    FuncEntry entries[4] = { // lookup table
        SurgeM$Timer$fired,
        PhotoTempM$PhotoTempTimer$fired,
        AMPromiscuous$ActivityTimer$fired,
        MultiHopLEPSM$Timer$fired,
    };
    if (arg_0x76cb2c8 >= 4U) { // old default
        return TimerM$Timer$default$fired();
    }
    return *(entries[arg_0xb76cb2c8]); // dispatch
}
```

Figure 3. Optimized version 2 of Timer.fired.

COTs	Effect on mean$(COMP)$	Effect on max$(COMP)$	Effect on MEM
cot_1	-2.938	+1	0
cot_2	-0.039	-4	+16

Table 1. Effects of code optimization techniques on resource usages.

The effect of optimization of resource usages by such COTs may be estimated statically. A prerequisite for implementing cot_2 is that the case block in switch has no default case, which means that cot_2 depends on cot_1. The effects of optimization can be found by comparing version P_1 with P_0, and comparing version P_2 with P_1. Table 1 shows the effects of P_1 and P_2 in units of number of comparison operations and memory blocks.

Considering that cot_2 depends on cot_1, legitimate combinations of code optimization techniques to synthesize such a component include $\{cot_1\}$ and $\{cot_1, cot_2\}$. Their resource usages are shown in Table 2, in which $\widetilde{\gamma}_{MEM}$ stands for the basis memory usage of version P_0.

While it cannot be guaranteed that estimated resource usages will truly reflect runtime resource usages, developers in practice often assume an approximately monotonic trend between them. Thus, they target at code versions with reduced estimated resource usages. Considering mean$(COMP)$, max$(COMP)$, and MEM in this example, P_2 is the best version.

To deal with different concerns, developers often use different COTs or their combinations. While these COTs may have dependencies or conflicting relationships among one another, such as function inlining conflicting with function pointer table, most

Version	mean($COMP$)	max($COMP$)	MEM
P_0	3.977	4	$\widetilde{\gamma}_{MEM}$
P_1	1.039	5	$\widetilde{\gamma}_{MEM}$
P_2	1	1	$\widetilde{\gamma}_{MEM}+16$

Table 2. Resource usages of tasks synthesized.

of the work in synthesizing the COTs is done manually at present. Each time the environment and the corresponding resource constraints change, extra manual work must be done to search for and adopt suitable code optimization techniques. While many standard approaches to optimization are available (as in P_1 and P_2), there may be many functional components requiring different COTs and many WSN nodes imposing different environmental constraints. It is very difficult to manually manage the complexity involved.

4. Model and Algorithms

This section presents our model and algorithms. Our component-based model is built on top of a task view described in Section 4.1. The model consists of a skeleton component view, basis resource usages, and code optimization techniques, as described in Sections 4.2 to 4.4.

4.1. Task View

A *task* is a notion used in the real-time and system communities. It is often realized as a process or a thread on many platforms including TinyOS and Java. It provides a simple and direct means of partitioning components for the analysis of resource usages. We adapt the task model from [14] as the formal model to represent a WSN application, where a task has a run-to-complete semantics, meaning that the task will complete its execution before another copy of the same task is being run.[4] A *task* [14] is a tuple $\tau = \langle \Phi, Prd, d, o, \omega, loc\rangle$, where $\Phi = \langle \alpha_1, \alpha_2, \cdots, \alpha_m\rangle$ is a list of m WSN components, Prd is the invocation period of the task, d is its relative deadline, o is its release time offset, ω: $\tau \rightarrow Q_0^+$ maps the task to its resource usages, and loc: $\tau \rightarrow N^+$ maps the task to an integer representing the WSN node.

4.2. Skeleton Component View

By considering all lists Φ of components of all the tasks τ in a task model, we set up our component model of WSN applications. We define a *component* as a tuple $\alpha = \langle Prd, d, pre, post, loc\rangle$, where Prd is the invocation period of the component, d is its relative deadline, pre is its previous component in the original list Φ, $post$ is its next component in Φ, and loc: $\alpha \rightarrow N^+$ maps the component to an integer representing the WSN node. In this way, the execution schedule of tasks in the original task model is converted to that of the components.

[4] Note that tasks are statically allocated in embedded systems. When there are needs for, say, 10 copies of the same task, we simply regard them as 10 distinct tasks in our model.

The component view will not be useful for resource optimization unless we attach to it the basis resource usages and the code optimization techniques. These concepts will be introduced in Sections 4.3 and 4.4.

4.3. Resource Concerns and Resource Usages

Resource concerns: We model a concern imposed by the application environment by means of its bounds. A *concern* is a range $\kappa = [min,\ max]$, where min represents the lower bound, and max the upper bound. For instance, in the motivation example of Section 3, a concern for CPU may be $[0,\ 2000]$, which means that the CPU can support no more than 2000 operations per second. Similarly, a concern for memory may be $[0k,\ 30k]$, which means that the memory available to a node is no more than 30k bytes. We use $K = \langle \kappa_1, \kappa_2, \cdots, \kappa_n\rangle$ to denote a list of concerns for n resources, where κ_j denotes the constraint for the j-th resource.

Resource usages: For every component α of a WSN application, the *resource usage* γ_j^α of the j-th resource is a numerical value within the range specified by the appropriate concern κ_j. We use $\Gamma^\alpha = \langle \gamma_1^\alpha, \gamma_2^\alpha, \cdots, \gamma_n^\alpha\rangle$ to denote a list of n resource usages.

Basis resource usages: Components should have resource usages even if software developers do not optimize them. To acknowledge this fact in our model, we attach a list of *n basis resource usages* $\widetilde{\Gamma}^\alpha = \langle \widetilde{\gamma}_1^\alpha, \widetilde{\gamma}_2^\alpha, \cdots, \widetilde{\gamma}_n^\alpha\rangle$ to every component α of a WSN application.

After the resource usages Γ^α of every component α have been determined, we can assemble them to compute the resource usages of a node or the whole application, and compare them with the given K to evaluate the overall impacts. This assembling computation is related to the executing schedule of the components. It will be further discussed in Section 5.1.

The basis resource usage $\widetilde{\Gamma}^\alpha$ can be improved to Γ^α according to a code optimization technique. In the next section, we shall further formulate the COTs.

4.4. Code Optimization Techniques

Each *code optimization technique* (*COT*) is inscribed in a component. A COT usually has local effects on resource usages. In other words, it only affects the resource usages of the component where it is inscribed. We model it as effects of optimization of resource usages.

Thus, we define a code optimization technique x^α for a component α as a list $x^\alpha = \langle \delta_1^\alpha, \delta_2^\alpha, \cdots, \delta_n^\alpha \rangle$, where each δ_j^α represents an increment or decrement of a resource usage γ_j^α from the corresponding basis usage $\widetilde{\gamma}_j^\alpha$. In the example in Section 3, for instance, $\widetilde{\Gamma}^{\text{Timer.fired}} = \langle 1000, 1100, 20k \rangle$ is the list of basis resource usages of the component. After adopting a code optimization technique $x^{\text{Timer.fired}} = \langle -200, +5, +2k \rangle$, the resource usage will become $\Gamma^{\text{Timer.fired}} = \langle 800, 1105, 22k \rangle$.

For every component, developers may define a set of code optimization techniques $X^\alpha = \{x_1^\alpha, x_2^\alpha, \cdots, x_{|X^\alpha|}^\alpha\}$.

In this way, we complete our adaptive design framework $(\alpha, \widetilde{\Gamma}^\alpha, X^\alpha)$ for a WSN component.

4.5. Order of Priority

cot_1 and cot_2 in the example in Section 3 show very different effects on resource usages, as shown in Table 1. In general, one code optimization technique may increase a specific resource usage while another technique may reduce it. To remedy this situation, we propose to use an order of priority $P = \langle p_1, p_2, \cdots, p_n \rangle$ to optimize the n resources. Here, $\langle p_1, p_2, \cdots, p_n \rangle$ is a permutation of $\langle 1, 2, \cdots, n \rangle$ and each p_j means that the p_j-th resource is of the j-th highest priority in optimization. Finding an optimal solution for such a problem is NP-hard in general. We shall explain our algorithms in the next two sections.

4.6. Objective of Algorithms

Given the preambles introduced in Sections 4.1 to 4.4 above, we can formulate our problem statement as follows:

Problem statement: Consider a WSN application in which there is a resource concern K and each component α is associated with a basis resource usage $\widetilde{\Gamma}^\alpha$ and a set of code optimization techniques X^α. Our goal is to find a combination of code optimization techniques $Y_{opt} = \{y_1, y_2, \cdots, y_{|Y_{opt}|}\}$ that collectively satisfy all the given concerns K and minimize the overall resource usages $\Gamma = \langle \Gamma^{\alpha_1}, \Gamma^{\alpha_2}, \cdots, \Gamma^{\alpha_m} \rangle$ for a given order of priority P for resource optimization.

If the COTs only provide maximal local effects of optimization to their assigned components, and if we can adapt each COT independently, it is easy to prove that a sufficient condition for Y_{opt} to be an optimal solution for the entire wireless sensor network application is that there exists an optimal solution $Y_{opt}^{\alpha_i}$ for every component α_i such that $Y_{opt} = Y_{opt}^{\alpha_1} \cup Y_{opt}^{\alpha_2} \cup \cdots \cup Y_{opt}^{\alpha_m}$. Formally, the *optimal combination* of code optimization techniques Y_{opt}^α for component α satisfies the four conditions in Figure 4.

1. $\forall x_i \in X^\alpha$ and $y_j \in Y_{opt}^\alpha$, $y_j \rhd x_i \Rightarrow x_i \in Y_{opt}^\alpha$.
2. $\forall y_j, y_k \in Y_{opt}^\alpha$, $\neg(y_j \diamond y_k)$.
3. $Y_{opt}^\alpha \subseteq X^\alpha$.
4. $\forall Y \subseteq X^\alpha$, $\Psi\big(P, F(\widetilde{\Gamma}^\alpha, Y_{opt}^\alpha), F(\widetilde{\Gamma}^\alpha, Y)\big) \leq 0$.

Figure 4. Conditions for optimal solution.

The first condition ensures that, given any COT in Y_{opt}^α, all its dependencies are also included in Y_{opt}^α. The second condition guarantees that any two COTs in Y_{opt}^α will not conflict with each other. The last two conditions ensures that Y_{opt}^α is a subset of X^α and produces the optimal effects of optimization of resource usages.

Let us explain the notations in Figure 4 in more detail. The relation $y \rhd x$ denotes that y depends on x, so that x must be adopted whenever y is adopted. The relation $x \diamond y$ denotes that x conflicts with y, so that only x or y can be adopted but not both. $F(\widetilde{\Gamma}^\alpha, Y) = \langle f_1(\widetilde{\gamma}_1^\alpha, Y), f_2(\widetilde{\gamma}_2^\alpha, Y), \cdots, f_n(\widetilde{\gamma}_n^\alpha, Y) \rangle$ is a list of functions calculating the resource usages according to the basis usages $\widetilde{\Gamma}^\alpha$ after implementing a set $Y = \{y_1^\alpha, y_2^\alpha, \cdots, y_{|Y|}^\alpha\}$ of code optimization techniques $y_k^\alpha = \langle \delta_{1,k}^\alpha, \delta_{2,k}^\alpha, \cdots, \delta_{n,k}^\alpha \rangle$. Each function f_j for the j-th resource usage is given by

$$f_j(\widetilde{\gamma}_j^\alpha, Y) = \widetilde{\gamma}_j^\alpha + \textstyle\sum_{k=1}^{|Y|} \delta_{j,k}^\alpha. \qquad (1)$$

For a given P, we define $\Psi(P, \Gamma, \Gamma')$

$$= \begin{cases} -1 & \text{if } P = \langle p_1, p_2, \cdots, p_n \rangle \\ & \text{and } \gamma_{p_1} < \gamma'_{p_1}; \\ 1 & \text{if } P = \langle p_1, p_2, \cdots, p_n \rangle \\ & \text{and } \gamma_{p_1} > \gamma'_{p_1}; \\ \Psi\big(P \backslash \{p_1\}, \Gamma, \Gamma'\big) & \text{if } P = \langle p_1, p_2, \cdots, p_n \rangle \\ & \text{and } \gamma_{p_1} = \gamma'_{p_1}; \\ 0 & \text{if } P = \emptyset. \end{cases}$$

It compares two resource usage sets Γ and Γ'. A negative returned value means that Γ is preferred to Γ', a positive value means that Γ' is preferred, and a zero means no preference.

When a solution is found, we can follow the description in Section 4.3 to set up a list of formulas $G = \langle g_1, g_2, \cdots, g_n \rangle$ to compute the application-level or node-level resource usages based on the n resource usages at the component level, where $g_j\big(\langle \widetilde{\gamma}_j^{\alpha_1}, \widetilde{\gamma}_j^{\alpha_2}, \cdots, \widetilde{\gamma}_j^{\alpha_m} \rangle, Y\big)$ is a summary of the j-th resource usage of all m components. For each g_j,

```
Algorithm:  Sorting of Code Optimization Techniques
Inputs:     unordered list of COTs $X = \langle x_1, x_2, \cdots, x_{|X|} \rangle$;
            order of priority $P$
Output:     ordered list of COTs $Z = \langle z_1, z_2, \cdots, z_{|X|} \rangle$
01. let $U = \langle u_1, u_2, \cdots, u_n \rangle = \langle 0, 0, \cdots, 0 \rangle$
02. let $V = \langle v_1, v_2, \cdots, v_n \rangle = \langle 0, 0, \cdots, 0 \rangle$
03. let $A = \langle a_1, a_2, \cdots, a_{|X|} \rangle = \langle 0, 0, \cdots, 0 \rangle$
04. let $Z = \langle z_1, z_2, \cdots, z_{|X|} \rangle = \langle 1, 2, \cdots, |X| \rangle$
05. for $j = 1, 2 \cdots, n$ do
06.     let $u_j = \min \left( \bigcup_{k=1}^{|X|} \{\delta_{j,k}\} \right)$, $v_j = \max \left( \bigcup_{k=1}^{|X|} \{\delta_{j,k}\} \right)$
07. for $k = 1, 2, \cdots, |X|$ do
08.     let $a_k = \sum_{j=1}^{n}$ utility$(\delta_{j,k}, u_j, v_j)$
09. for $i, j = 1, 2, \cdots, |X|$ such that $i < j$ do
10.     if $a_{z_i} > a_{z_j} \vee \left( a_{z_i} = a_{z_j} \wedge \Psi(P, x_{z_i}, x_{z_j}) > 0 \right)$ then
11.         swap $z_i, z_j$
12. exit
```

Figure 5. Algorithm to sort code optimization techniques.

the first argument is a list of basis resource usages in respective components, and the second argument is a set of COTs. By comparing the resulting values of G with the given concerns K, we can evaluate the solution.

4.7. The Algorithms

Our algorithms cover two phases: the sorting of code optimization techniques and the generation of a combination.

Sorting of code optimization techniques: The algorithm will firstly estimate the *optimization capability* of each code optimization technique, which means how much the COT may optimize within given concerns. It will then sort all the COTs with respect to their estimated optimization capabilities. The algorithm, depicted in Figure 5, accepts a set of code optimization techniques and an order of priority P as arguments and returns an ordered list of COTs $Z = \langle z_1, z_2, \cdots, z_{|X|} \rangle$.[5]

For the purpose of flexibility when comparing effects of optimization in the algorithm, we use a utility function utility to estimate the optimization capability of components. Thus, the optimization capability for the j-th resource is represented as a function of the effect of optimization $\delta_{j,k}$ of the k-th COT as well as the minimum resource usage u_j and the maximum resource usage v_j of all COTs. We define our utility function in Section 5; software developers may define their own utility function instead. The result of this function should be monotonic to the value of the input δ_j.

[5] Standard bubble sort is used in the prototype algorithm. Faster sorting techniques may alternatively be applied to improve the efficiency.

Generation of combination: Given a sorted list of COTs produced in the first phase, the present phase generates a suboptimal combination. We use a hill-climbing strategy in the algorithm. Every possible combination of COTs fulfilling the order of priority P will be considered in turn. We rank the combinations before the algorithm begins. For every combination of r selections from $|Z|$ choices, denoted by $\{z_{s_1}, z_{s_2}, \cdots, z_{s_r}\}$, its lexicographical index [2] is the concatenated string "$s_1 s_2 \cdots s_r$". We simply sort all the combinations in ascending order of the lexicographical indexes, and use C_j to denote the j-th combination in the ordered list. (Since this is a fundamental concept in combinatorics, we do not include it in the skeleton algorithm in Figure 6.) The iteration will continue until the concerns have been satisfied and a locally optimal result has been found, which means that the first minimum point has been reached. Then, the algorithm returns a combination of COTs $Y = \{y_1, y_2, \cdots, y_{|Y|}\}$. If all legitimate combinations have been exhausted but the concerns cannot be fulfilled, the algorithm returns an empty set.

The procedure dependences in the algorithm accepts a code optimization technique z_j as input and returns a combination of code optimization techniques $Y = \{y_1, y_2, \cdots, y_{|Y|}\}$, which includes the COTs z_j depends on. The procedure valid accepts Y and returns a Boolean value indicating whether it is a legitimate combination of COTs that satisfies the concerns.

The main entry of this algorithm iteratively processes all legitimate selections of COTs. After some iterations, when sufficient number of COTs have been considered, the result may be able to meet the resource constraints of the WSN application. When the iteration process continues, the estimation result is expected to further improve, but only up to a certain limit. When the algorithm finds that the resultant resource usage begins to recede, a heuristic solution has been found and the algorithm terminates. The experimental results in the next section show that such a heuristic strategy can be very helpful in the search for good solutions.

Complexity of algorithms: The prototype algorithm for sorting code optimization techniques can be completed in $O(|X|^2 \cdot n)$ time, where $|X|$ is the number of COTs and n is the number of resource types.

The prototype algorithm for generating combination iteratively evaluates possible selections until a solution is found. A disadvantage of this prototype is its high time complexity in the worst case, which is $O(2^{|X|} \cdot |X|^2 \cdot n)$. On the other hand, we note from the experiment in Section 5 that the algorithm can find solutions much earlier than exhaustive search. We note also that, in practice, we may set an affordable upper

```
Algorithm:  Generation of Combination
Inputs:     ordered list of COTs $Z = \langle z_1, z_2, \cdots, z_{|Z|} \rangle$;
            basis resource usages $\langle \widetilde{\Gamma}^{\alpha_1}, \widetilde{\Gamma}^{\alpha_2}, \cdots, \widetilde{\Gamma}^{\alpha_m} \rangle$;
            order of priority $P$
Output:     combination of COTs $Y = \{y_1, y_2, \cdots, y_{|Y|}\}$
01.  let $Y = \emptyset$
02.  for $j = 1, 2, \cdots, 2^{|Z|} - 1$ do
03.      let $Y' = \emptyset$
04.      foreach $z_{s_k}$ in $C_j$ do
05.          let $Y' = Y' \cup$ dependences($z_{s_k}$)
06.      if valid($Y'$) then
07.          let $last = G(\langle \widetilde{\Gamma}^{\alpha_1}, \widetilde{\Gamma}^{\alpha_2}, \cdots, \widetilde{\Gamma}^{\alpha_m} \rangle, Y)$
08.          let $curr = G(\langle \widetilde{\Gamma}^{\alpha_1}, \widetilde{\Gamma}^{\alpha_2}, \cdots, \widetilde{\Gamma}^{\alpha_m} \rangle, Y')$
09.          if $\Psi(P, last, curr) < 0$ then
10.              exit
11.          let $Y = Y'$
12.  exit

Procedure:  dependences
Input:      COT $z_i$
Output:     set of COTs $Y$
01.  let $Y = \{z_i\}$
02.  foreach $z_j \in Z$ such that $z_i \triangleright z_j$ do
03.      let $Y = Y \cup$ dependences($z_j$)
04.  exit

Procedure:  valid
Input:      set of COTs $Y$
Output:     Boolean value
01.  foreach $y_i, y_j \in Y$ do
02.      if $y_i \diamond y_j$ then
03.          return false and exit
04.  foreach $g_j$ of $G$ do
05.      if $g_j(\langle \widetilde{\Gamma}_j^{\alpha_1}, \widetilde{\Gamma}_j^{\alpha_2}, \cdots, \widetilde{\Gamma}_j^{\alpha_m} \rangle, Y) \notin \kappa_j$ then
06.          return false and exit
07.  return true and exit
```

Figure 6. Algorithm to generate combination.

bound of the number of combinations to be checked to find a solution.

5. Experimental Study

In this section, we firstly select a few representative types of resource for experiment and set up their calculation formulas G. Then, we construct a model of a real-life application and evaluate the performance of the algorithms.

5.1. The Resources

We select three most common and widely-used resources for our experimentation on optimization. For every individual node, we study the average CPU operations per second (CPU), the maximum memory usage (MEM), and the volume of application-level communication[6] ($COMM$). Hence, in the following experiment, the resource usage can be represented by $\Gamma = \langle \gamma_{CPU}, \gamma_{MEM}, \gamma_{COMM} \rangle$ and the resource constraint by $K = \langle \kappa_{CPU}, \kappa_{MEM}, \kappa_{COMM} \rangle$.

$$\boxed{\begin{aligned}
&g^{cec}_{CPU}\big(\langle \widetilde{\gamma}^{\alpha_1}_{CPU}, \widetilde{\gamma}^{\alpha_2}_{CPU} \rangle, Y\big)\\
&= f_{CPU}(\widetilde{\gamma}^{\alpha_1}_{CPU}, Y) + f_{CPU}(\widetilde{\gamma}^{\alpha_2}_{CPU}, Y)\\
&g^{sec}_{CPU}\big(\langle \widetilde{\gamma}^{\alpha_1}_{CPU}, \widetilde{\gamma}^{\alpha_2}_{CPU} \rangle, Y\big)\\
&= \max\big\{f_{CPU}(\widetilde{\gamma}^{\alpha_1}_{CPU}, Y), f_{CPU}(\widetilde{\gamma}^{\alpha_2}_{CPU}, Y)\big\}\\
&g_{MEM}\big(\langle \widetilde{\gamma}^{\alpha_1}_{MEM}, \widetilde{\gamma}^{\alpha_2}_{MEM} \rangle, Y\big)\\
&= \max\big\{f_{MEM}(\widetilde{\gamma}^{\alpha_1}_{MEM}, Y), f_{MEM}(\widetilde{\gamma}^{\alpha_2}_{MEM}, Y)\big\}\\
&g_{COMM}\big(\langle \widetilde{\gamma}^{\alpha_1}_{COMM}, \widetilde{\gamma}^{\alpha_2}_{COMM} \rangle, Y\big)\\
&= f_{COMM}(\widetilde{\gamma}^{\alpha_1}_{COMM}, Y) + f_{COMM}(\widetilde{\gamma}^{\alpha_2}_{COMM}, Y)
\end{aligned}}$$

Figure 7. Calculation formulas.

Figure 7 shows the calculation formulas $G = \langle g_{CPU}, g_{MEM}, g_{COMM} \rangle$ for computing application- or node-level resource usages based on the usages in two components α_1 and α_2. In particular, g^{cec}_{CPU} is formula for concurrent execution of two components and g^{sec}_{CPU} is for sequential execution of the same. For the case of more than two components, their formulas can be reasoned hierarchically according to the execution schedule.

5.2. Subject of Experiment

The subject program is CntToLedsAndRfm[7] written in nesC for the project TOSSIM. TOSSIM is a representative emulator of TinyOS [9].

A TinyOS application on any node of a wireless sensor network is designed to support only sequentially and periodically executed tasks [4]. Although tasks on different nodes may be executed concurrently, those on the same node are executed sequentially. Each task is processed in a run-to-complete manner. Thus, we can work out the execution schedule of the components from the tasks and, hence, set up the functions F to compute the resource usages.

5.3. Setup of Experiment

CntToLedsAndRfm consists of two nodes of the same function. Each node periodically increases a local counter, shows the lower bit values of the counter on LEDs, and sends the counter value to another node.

For the purpose of experimentation, we remove the debugging task and expand the application by cloning nodes and components. The resultant program consists of three nodes, each having three to four components with fixed orders of execution without idle time. Each

[6] That is, the estimated total number of bytes sent or received.

[7] Available at http://www.tinyos.net/tinyos-1.x/apps/CntToLedsAndRfm/.

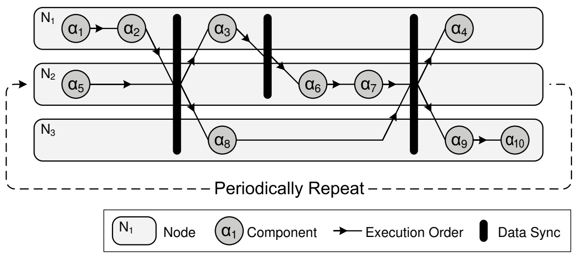

Figure 8. Infrastructure of testbed.

$$
\begin{aligned}
&g_{CPU}\big(\langle\widetilde{\gamma}^{\alpha_1}_{CPU},\widetilde{\gamma}^{\alpha_2}_{CPU},\cdots,\widetilde{\gamma}^{\alpha_{10}}_{CPU}\rangle,Y\big)\\
&=\max\big\{\textstyle\sum_{i=1}^{4} f_{CPU}(\widetilde{\gamma}^{\alpha_i}_{CPU},Y),\sum_{i=5}^{7} f_{CPU}(\widetilde{\gamma}^{\alpha_i}_{CPU},Y),\\
&\qquad\quad \textstyle\sum_{i=8}^{10} f_{CPU}(\widetilde{\gamma}^{\alpha_i}_{CPU},Y)\big\}\\
&g_{MEM}\big(\langle\widetilde{\gamma}^{\alpha_1}_{MEM},\widetilde{\gamma}^{\alpha_2}_{MEM},\cdots,\widetilde{\gamma}^{\alpha_{10}}_{MEM}\rangle,Y\big)\\
&=\max\{f_{MEM}(\widetilde{\gamma}^{\alpha_1}_{MEM},Y),f_{MEM}(\widetilde{\gamma}^{\alpha_2}_{MEM},Y),\cdots,f_{MEM}(\widetilde{\gamma}^{\alpha_{10}}_{MEM},Y)\}\\
&g_{COMM}\big(\langle\widetilde{\gamma}^{\alpha_1}_{COMM},\widetilde{\gamma}^{\alpha_2}_{COMM},\cdots,\widetilde{\gamma}^{\alpha_{10}}_{CPU}\rangle,Y\big)\\
&=\textstyle\sum_{i=1}^{10} f_{COMM}(\widetilde{\gamma}^{\alpha_i}_{COMM},Y)
\end{aligned}
$$

Figure 9. Calculation formulas for testbed.

component is equipped with COTs, some of which have dependences or conflicting relationships among one another. Figure 8 shows a schematic component-and-connector diagram of the program.

Suppose we have resource concerns regarding *CPU* and *MEM* at the node level and *COMM* at the application level. They can be calculated using the formulas in Figure 7. This is illustrated by Figure 9, where g_{CPU} represents the average number of CPU operations per second of an individual node, g_{MEM} represents the maximum memory usage of an individual node, g_{COMM} represents the volume of communication of the application, and f_{CPU}, f_{MEM}, and f_{COMM} are calculated by equation (1).

To apply the algorithms introduced in Section 4.7, we use $\text{utility}(\delta_{j,k},u_j,v_j)=\lfloor\frac{10\delta_{j,k}-5u_j-5v_j}{2v_j-2u_j}\rfloor$ in the experiment to evaluate the overall optimization capability of a COT. The lower the resulting value, the stronger will be the optimization capability. In general, a proper utility function can be chosen after a code review of the original subject program.

Our experiment is conducted on a Dell Inspiron 6400 laptop, which is equipped with an Inter Core(TM)2 T5600 @ 1.83GHz stepping 06 CPU and 1G memory. The operating system is Ubuntu 6.06 LTS Linux with kernel version 2.6.15-28-386 (buildd@terranova).

The subject application is from the TinyOS tool set TOSSIM version 1.1.15 (December 2005), which can be downloaded from http://www.TinyOS.net/download.html. Our driver programs are coded in C++. All the programs are compiled with ncc version 1.1.EF15 or gcc version 4.0.3 (Ubuntu 4.0.3-1ubuntu 5).

5.4. Experimental Evaluation Results

This section presents the experimental evaluation results with respect to the overall optimization capability, the order of priority for resource optimization, and sensitivity. For the space reasons, we shall not discuss efficiency results but concentrate only on the effectiveness of our approach.

Comparison with other solutions: Our experiment can be repeated deterministically. We report the results with the order of priority for resource optimization to be set as $P=\langle CPU, COMM, MEM\rangle$ and the concerns K to be 1.5 times the basis resource usages.

We compare our approach with three other solutions for code optimization, as shown in Figure 10. The three other solutions include: **(a) Fully optimal solution:** We iterate *all* legitimate selections and find the fully optimal solution. **(b) Randomly selected solution:** We randomly pick 300 COTs and then choose from them the COTs with the minimum resource usages. The magic number 300 is chosen from experience according to the scale of the problem. **(c) Unoptimized solution:** The original subject program is taken as an "unoptimized" solution. We should point out that the original subject program is manually crafted by professional developers. Since it targets for wireless sensor network applications, code optimization has been conducted, albeit not to an optimized level. The resource usages of the subject program are normalized as 1.0 as a benchmark for various solutions.

Resource usages are classified into three groups, namely (from top to bottom in Figure 10) *CPU*, *COMM*, and *MEM*; the usages in the four solutions are shown under each group. We notice that *CPU* usage is best optimized, followed by *COMM* usage, according to the order of priority specified by *P*. This is consistent with our hypothesis that *CPU* and *COMM* usages are reduced at the expense of increased *MEM* usage. We also notice that, for the *CPU* resource, which is the main objective of optimization in the empirical study, our model obviously produces a better usage pattern than a randomly selected combination of COTs. Our results are only overtaken by the optimal solution for the *MEM* resource, which is at the lowest priority of optimization.

Changes in resource usages for different orders of priority: To analyze the adaptive capability of our algorithm to different orders of priority, we submit all six possible priority orders for resource optimization

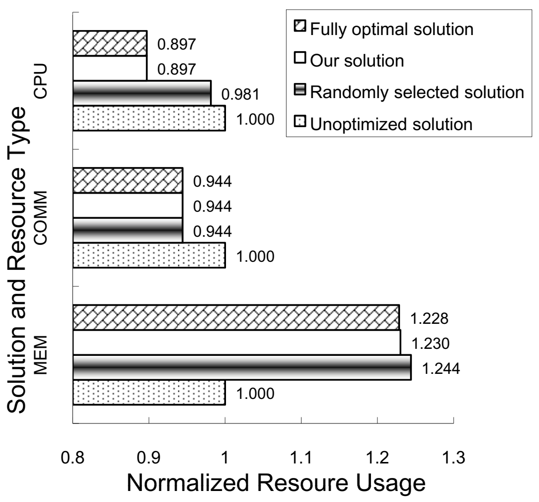

Figure 10. Comparison of solutions.

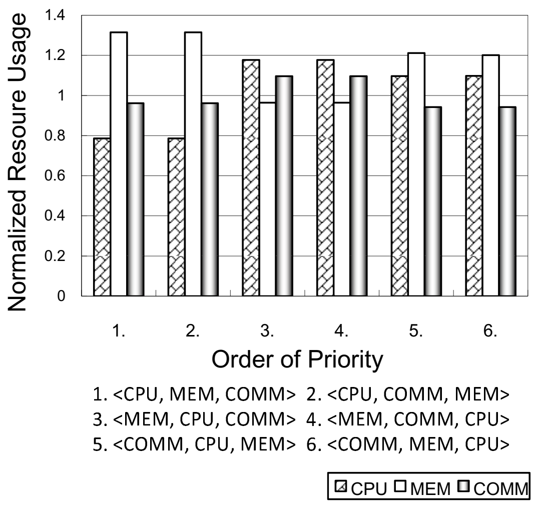

Figure 11. Effects of *P* on resource usage.

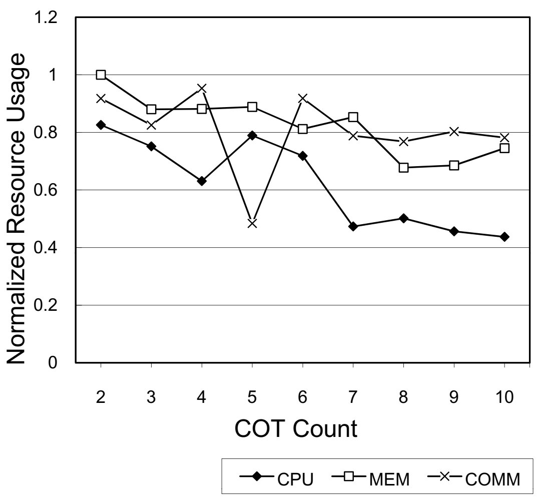

Figure 12. Effects of COT on resource usage

as inputs and present the results in Figure 11. The results are plotted in six groups, showing the results for six different orders of priority. Each group consists of three columns representing the resource usages of *CPU*, *MEM*, and *COMM*. We notice that, whenever we set a top priority to a resource, the usage of that resource will automatically be best optimized. This indicates that our model have a high adaptation capability for different orders of priority.

We also observe from the experiment that resources may have different properties when being optimized. In the example, many COTs that target at reducing *CPU* or *COMM* do so at the expense of increased *MEM*. This is because many code optimization techniques are achieved through additional memory usages, such as caches and lookup tables, which are very common in real-life. On the other hand, *COMM* is very difficult to be reduced. Even inconspicuous reductions in *COMM*, such as orders 5 and 6 in the figure, may result in disproportionate increases in *CPU* and *MEM* usages.

Variations in resource usages for different COT counts: Intuitively, the number of code optimization techniques used in an experiment (referred to as the *COT count*) should enhance the results. In our experiment, we vary the COT count from 2 to 10. Figure 12 shows the variations in resource usages with respect to different COT counts. The x-axis is the COT count while the y-axis is the normalized resource usages. We notice that when the COT count increases from two to three, all the three resource usages are reduced. When the COT count continues to increase, the resource usages showed fluctuations; however, they still show descending trends. We postulate that this is due to the hill-climbing strategy used in our algorithm. The results are expected to improve by implementing more advanced algorithms.

The results also show that having more choices of code optimization techniques may help improve *CPU* and *MEM*. When the COT count increases, however, the complexity in choosing a promising one from different combinations increases. It makes our automated approach to synthesize COTs in WSN applications more attractive.

5.5. Threats to Validity

A threat to internal validity is the assumption that code optimization techniques are applied in WSN applications. There is no guarantee that COTs are considered in any given real-world application.

A threat to external validity may be due to the resources chosen for experimentation. We have taken three representative kinds of resource for the study and set up the corresponding calculation formulas. A TinyOS application may include other kinds of concern.

Other threats to external validity include the use of a monotonic utility function and the use of the specific WSN platform TinyOS. As we have described in Section 4.7, the utility function should be monotonic. The quality of our results depends on this characteristic. Although developers may define their own utility functions, such functions must also be monotonic. Also, while TinyOS is the most widely used platform for WSN applications, we have not investigated any other platforms.

A construct validity is that we clone components and COTs in the experiment, which may affect the optimized result.

6. Conclusion

Optimization is indispensable in the design and implementation of wireless sensor network applications because of the stringent resource constraints referred to as concerns. Developers often need to iteratively select possible code optimization techniques (COTs) to meet the resource concerns. Such manual work is inefficient and error-prone.

In this paper, we present a model to manage COTs and evaluate its usefulness in optimizing the effectiveness under given concerns and a user-defined order of priority. The evaluation is conduced through estimated usages of resources based on the infrastructure of an application under study. An experimental study shows that our approach provides a promising solution to code optimization. As future work, it will be interesting to explore context-awareness, runtime adaptation, and more elaborate experimentation. We will also study how to specify COTs and how interactions among COTs may affect our approach.

References

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40 (8): 102–114, 2002.

[2] B. P. Buckles and M. Lybanon. Algorithm 515: generation of a vector from the lexicographical index [G6]. *ACM Transactions on Mathematical Software*, 3 (2): 180-182, 1977.

[3] D. Gay, P. Levis, and D. Culler. Software design patterns for TinyOS. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (*LCTES 2005*), *ACM SIGPLAN Notices*, 40 (7): 40–49, 2005.

[4] 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 *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI 2003*), *ACM SIGPLAN Notices*, 38 (5): 1–11, 2003.

[5] J. Huselius and J. Andersson. Model synthesis for real-time systems. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering* (*CSMR 2005*), pages 52–60. IEEE Computer Society Press, Los Alamitos, CA, 2005.

[6] K. Kaspersky. *Code Optimization: Effective Memory Usage*. A-List Publishing, Wayne, Pennsylvania, 2003.

[7] K. Kuchcinski. Embedded system synthesis by timing constraints solving. In *Proceedings of the 10th International Symposium on System Synthesis*, (*ISSS 1997*), pages 50–57. IEEE Computer Society Press, Los Alamitos, CA, 1997.

[8] M. Kuorilehto, M. Hännikäinen, and T.D Hämäläinen. A survey of application distribution in wireless sensor networks. *EURASIP Journal on Wireless Communications and Networking*, 5 (5): 774–788, 2005.

[9] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In *Proceedings of the 1st ACM Conference on Embedded Networked Sensor Systems* (*SenSys 2003*). ACM Press, New York, NY, 2003.

[10] M. Sgroi, L. Lavagno, and A. Sangiovanni-Vincentelli. Formal models for embedded system design. *IEEE Design and Test of Computers*, 17 (2): 14–27, 2000.

[11] I. Shin, I. Lee, and S. L. Min. A design approach for real-time embedded systems with energy and code size constraints. In *Proceedings of the 10th Real-time and Embedded Computing Systems and Applications Conference* (*RTCSA 2004*). Gothenburg, Sweden, 2004.

[12] M. Srivastava. Wireless sensor and actuator networks: challenges in long-lived and high-integrity operation. In *Lecture Notes of Croucher Foundation ASI Lecture on Wireless Sensor Networks*. City University of Hong Kong, Hong Kong, 2006.

[13] J. Teich, L. Thiele, and L. Z. Zhang. Partitioning processor arrays under resource constraints. *Journal of VLSI Signal Processing Systems*, 17 (1): 5–20, 1997.

[14] S. Wang and K. G. Shin. Task construction for model-based design of embedded control software. *IEEE Transactions on Software Engineering*, 32 (4): 254–264, 2006.

[15] E. Wohlstadter, S. Tai, T. Mikalsen, I. Rouvellou, and P. Devanbu. GlueQoS: middleware to sweeten quality-of-service policy interactions. In *Proceedings of the 26th International Conference on Software Engineering* (*ICSE 2004*), pages 189–199. IEEE Computer Society Press, Los Alamitos, CA, 2004.

[16] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th International Conference on Software Engineering* (*ICSE 2006*), pages 371–380. ACM Press, New York, NY, 2006.