Postprint of article in *Journal of Systems and Software* **82** (9): 1376-1387 (2009)

Resource Prioritization of Code Optimization Techniques for Program Synthesis of Wireless Sensor Network Applications[☆,☆☆]

Zhenyu Zhang[a], W.K. Chan[b], T.H. Tse[a,*], Heng Lu[a], Lijun Mei[a]

[a]*Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong*
[b]*Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong*

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. To meet these constraints, developers often design their programs by trial-and-error. Such manual process is time-consuming and error-prone.

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 provide a visualization mechanism to help developers select code optimization techniques. We also develop algorithms to synthesize components running on nodes, fulfilling the constraints, and thus optimizing their quality.

Key words:
Wireless sensor network, adaptive software design, resource constraint, code optimization technique

☆☆A preliminary version of this paper was presented at the 7th International Conference on Quality Software (QSIC 2007). This research is supported in part by a GRF grant of the Research Grants Council of Hong Kong (project no. 716507) and a grant of City University of Hong Kong (project no. 7002324).

*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.

Email addresses: zyzhang@cs.hku.hk (Zhenyu Zhang), wkchan@cs.cityu.edu.hk (W.K. Chan), thtse@cs.hku.hk (T.H. Tse), hlu@cs.hku.hk (Heng Lu), ljmei@cs.hku.hk (Lijun Mei)

1. Introduction

A wireless sensor network (WSN) is a computer network of sensor nodes interconnected by short-range and short-life wireless communication channels (Akyildiz et al., 2002). Each sensor node may capture data, such as temperature and light intensity, from the environment. Applications running on WSNs, such as animal surveillances (Szewczyk et al., 2004), automatic detections of geological events, and hospital administrations (Shnayder et al., 2005), should sense physical events *in-situ* (Kuorilehto et al., 2005) and analyze the sensed data *in-network* (Srivastava, 2006).

Power-aware applications are common in WSNs (Chan et al., 2007). Communication consumes the highest amount of energy in sensor nodes, followed next by processing, and then storage (Healy et al., 2007). 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. The resource usages of the COTs are visualized as a color palette. The developer can select COTs with higher *optimization capabilities* by choosing the COTs in darker colors in the palette, and thus decide on a favorable COT combination. We further present heuristic algorithms to determine the COT combination automatically. The empirical result shows that our method is effective and efficient.

The main contributions of the paper are fourfold: First, it proposes an application-level design optimization model for WSN applications. Second, it develops algorithms to construct components that support the automatic selection of a suite of COTs. Third, it provides a visualization mechanism to help developers manually decide a COT combination. Fourth, it provides the first empirical study on the topic, comparing the effectiveness of applying our heuristic algorithm to a case study with that of the corresponding manual process.

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 (1997) synthesizes an embedded system to meet timing constraints. Similarly, Wang and Shin (2006) construct tasks to tackle a similar issue with a view to minimizing the overall elapsed time. Other than timing constraints, Teich et al. (1997) study the processing capability of partitioned processor arrays. Shin et al. (2004) further investigate how to tackle the energy and code size constraints. These studies inspire 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. (2005) 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 Kaspersky (2003), code optimization techniques can be embedded into the code similarly to design patterns. A difference between our method and that of Kaspersky (2003) 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 (2005) introduce their model synthesis work for real-time systems, which focuses on architectures and observed behaviors. Kuchcinski (1997) 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 Wohlstadter et al. (2004), which only investigates the interaction relationships of optimization techniques but not their aggregated effects.

This paper is also related to searching. Kulkarni et al. (2004) describe two complementary general approaches, which are designed to achieve faster searches when genetic algorithms are used. The results show that evolutionary compilation can be used to tune embedded applications. Zhao et al. (2006) make use of an analytic model and heuristic algorithms to investigate the profitability of optimizations, which can be used to determine the effectiveness of applying optimizations. They suggest one can determined from the model whether an optimization is beneficial and should be applied, without the need to actually applying it. Özcan and Onbaşioglu (2007) propose a memetic algorithm to find the best number of processors and the best data distribution method for each stage of a parallel program.

Different crossover operators as well as hill-climbing methods are used to compare a steady-state memetic algorithm with a transgenerational memetic algorithm.

We treat WSN applications as components. Zhang and Cheng (2006) use Petri nets as a model to cater for the design of adaptive behavior, while Sgroi et al. (2000) propose a communicating finite state machine model with a similar aim. Their applicability to WSNs is yet to be studied.

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.

```
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.

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 $\frac{1+2+3+4+252\times4}{256} \approx 3.977$. 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 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 $\frac{2+3+4+5+252\times1}{256} \approx 1.039$ and max($COMP$) increases to 5.

```
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.

[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/.

[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.

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 $\text{mean}(COMP) = \max(COMP) = 1$. 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.

```
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.

The effects 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 $\text{mean}(COMP)$, $\max(COMP)$, 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 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, code optimization techniques, and optimization priority, as described in Sections 4.2 to 4.6. The visualization mechanism and the heuristic algorithms are given in Sections 4.7 to 4.8.

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 Wang and Shin (2006) 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* (Wang and Shin, 2006) is a tuple $\tau = \langle \Phi, Prd, d, o, \omega, loc \rangle$, where $\Phi = \langle \alpha_1, \alpha_2, \ldots, \alpha_m \rangle$ 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] 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.

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.

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.

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.

The component view will not be useful for resource optimization unless we attach to it the basis resource usages, the code optimization techniques, and the optimization priority. These concepts will be introduced in Sections 4.3 to 4.5.

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 30KB (or 30×1024 bytes). We use $K = \langle\kappa_1, \kappa_2, \ldots, \kappa_n\rangle$ 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 $\Gamma^\alpha = \langle\gamma_1^\alpha, \gamma_2^\alpha, \ldots, \gamma_n^\alpha\rangle$ 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* $\widetilde{\Gamma}^\alpha = \langle\widetilde{\gamma}_1^\alpha, \widetilde{\gamma}_2^\alpha, \ldots, \widetilde{\gamma}_n^\alpha\rangle$ 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 will 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 $x^\alpha = \langle\delta_1^\alpha, \delta_2^\alpha, \ldots, \delta_n^\alpha\rangle$, 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, $\widetilde{\Gamma}^{\mathsf{Timer.fired}} = \langle 1000, 1100, 20K\rangle$ is the list of basis resource usages of the component. After adopting a code optimization technique $x^{\mathsf{Timer.fired}} = \langle -200, +5, +2K\rangle$, the resource usage will become $\Gamma^{\mathsf{Timer.fired}} = \langle 800, 1105, 22K\rangle$.

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

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

4.5. Order of Priority

The two code optimization techniques 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 $P = \langle p_1, p_2, \ldots, p_n \rangle$ to optimize the n resources. Here, $\langle p_1, p_2, \ldots, p_n \rangle$ is a permutation of $\langle 1, 2, \ldots, n \rangle$ and each p_j means that the p_j-th resource is of the j-th highest priority in optimization.

4.6. Objective of our Model

Given the preambles introduced in Sections 4.1 to 4.5 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 $Y_{opt} = \{y_1, y_2, \ldots, y_{|Y_{opt}|}\}$ that collectively satisfy all given concerns K and minimize the overall resource usages $\Gamma = \langle \Gamma^{\alpha_1}, \Gamma^{\alpha_2}, \ldots, \Gamma^{\alpha_m} \rangle$ 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 $Y_{opt} = Y_{opt}^{\alpha_1} \cup Y_{opt}^{\alpha_2} \cup \ldots \cup Y_{opt}^{\alpha_m}$. 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. $F(\widetilde{\Gamma}^{\alpha}, Y) = \langle f_1(\widetilde{\gamma}_1^{\alpha}, Y), f_2(\widetilde{\gamma}_2^{\alpha}, Y), \ldots, f_n(\widetilde{\gamma}_n^{\alpha}, Y) \rangle$ is a list of functions calculating the resource usages according to the basis usages $\widetilde{\Gamma}^{\alpha}$ after implementing a set $Y = \{y_1^{\alpha}, y_2^{\alpha}, \ldots, y_{|Y|}^{\alpha}\}$ of code optimization techniques $y_k^{\alpha} = \langle \delta_{1,k}^{\alpha}, \delta_{2,k}^{\alpha}, \ldots, \delta_{n,k}^{\alpha} \rangle$. Each function f_j for the j-th resource usage is given by

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

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

$$\begin{cases} -1 & \text{if } P = \langle p_1, p_2, \ldots, p_n \rangle \text{ and } \gamma_{p_1} < \gamma'_{p_1}; \\ 1 & \text{if } P = \langle p_1, p_2, \ldots, p_n \rangle \text{ and } \gamma_{p_1} > \gamma'_{p_1}; \\ \Psi\big(\langle p_2, p_3, \ldots, p_n \rangle, \Gamma, \Gamma'\big) & \text{if } P = \langle p_1, p_2, \ldots, p_n \rangle \text{ and } \gamma_{p_1} = \gamma'_{p_1}; \\ 0 & \text{if } P = \emptyset \end{cases}$$

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 *calculation formulas* $G = \langle g_1, g_2, \ldots, g_n \rangle$ to compute the application-level or node-level resource usages based on the n resource usages at the component level, where $g_j\big(\langle \widetilde{\gamma}_j^{\alpha_1}, \widetilde{\gamma}_j^{\alpha_2}, \ldots, \widetilde{\gamma}_j^{\alpha_m} \rangle, Y\big)$ is a summary of the j-th resource usage of all m components. For each g_j, 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.

Finding an optimal solution for such a problem is NP-hard in general (Kulkarni et al., 2004, Zhao et al., 2006). We propose two options to tackle the problem: We provide a mechanism for developers to graphically visualize the optimization capabilities of the code optimization techniques. This visualization mechanism helps developers choose the appropriate combination manually. Alternatively, we can use heuristic searching to identify a suboptimal solution within a time limit.

Our model can be summarized as three steps.

(1) Represent the target WSN application with a component-based model, with attached COTs X. Generate calculation formulas G to compute the usages of concerned resources.

(2) *Either* display the COTs as a color palette, which helps developers manually choose a resultant COT combination Y, *or* use heuristic algorithms and the calculation formulas G to determine a resultant COT combination Y.

(3) Estimate the concerned resource usages by inputting the above resultant COT combination Y to the calculation formulas G.

We will explain our visualization mechanism and algorithms in the next two sections.

4.7. Visualization of Code Optimization Techniques

To help developers manually decide the combination of code optimization techniques, we use a color palette to represent the COTs, as shown in Figure 5. Each column shows the resource usage optimizations of a COT. Each row shows the resource usage optimizations of various COTs on one kind of resource. Consider, for example, the cell marked "•". It represents the optimization capability of the k-th COT on the j-th resource.

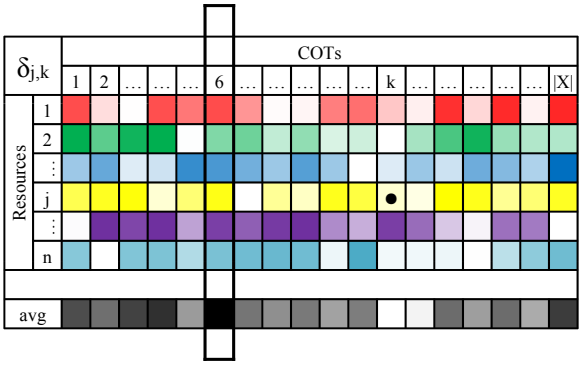

Figure 5: Using color palette to represent COTs.

We introduce the concept of *optimization capability*. The optimization capability of each code optimization technique means how much the COT optimizes within given concerns. To balance among all resources, we normalize the resource usage optimization of each code optimization technique. The *lightness* value of each cell in the grid is calculated using a utility function lum:

$$\mathsf{lum}(\delta_{j,k}) = \left\lfloor 255 \times \frac{\max\limits_{1\le k\le n}[\delta_{j,k}] - \delta_{j,k}}{\max\limits_{1\le k\le n}[\delta_{j,k}] - \min\limits_{1\le k\le n}[\delta_{j,k}]} \right\rfloor$$

Using lum, the resource usage optimization of a COT is normalized to $[0, 255]$. The darker the color, the higher will be the optimization capability.

The lightness values in each row represent the optimization capabilities with respect to each resource. To compare them directly with one another, we normalize them independently for each resource. Then, for every COT, we simply calculate its *average optimization capability* with respect to all resources. Let us focus on the column enclosed by a thick rectangle, which stands for the optimization capability of the 6-th COT. Since it has high optimization capabilities on most resources, its average optimization capability is the highest (indicated by the darkest cell in the bottom row). The average optimization capability is calculated by:

$$\mathsf{avglum}(\delta_{j,k}) = \left\lfloor \frac{1}{n}\sum_{k=1}^{n}\left[\mathsf{lum}(\delta_{j,k})\right] \right\rfloor$$

Thus, the developer may choose the 6-th COT together with some other COTs to form a combination; and iteratively refine the solution.

4.8. The Searching 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 first estimates the average optimization capability of each code optimization technique. Then, the COTs are sorted using traditional insertion sort. When the average optimization capabilities of two COTs are exactly the same, their optimization capabilities on different resources should be considered according to the given priority. The algorithm, depicted in Figure 6, accepts a set of COTs X and an order of priority P as arguments and returns an ordered list of COTs Z.

Generation of combination: Given a sorted list of COTs Z 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 $\{z_{s_1}, z_{s_2}, \ldots, z_{s_r}\}$, its lexicographical index (Buckles and Lybanon, 1977) is the concatenated string "$s_1 s_2 \ldots s_r$". 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 7.) 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 $Y = \{y_1, y_2, \ldots, y_{|Y|}\}$. If all legitimate combinations have been exhausted but the concerns cannot be fulfilled, the algorithm returns an empty set. If the iterations always produce better solutions than before, the algorithm returns the last one. However, the lexicographical order of the combinations naturally implies that there is a high chance for suboptimal solutions to appear early.

	Algorithm:	Sorting of Code Optimization Techniques
	Input:	set of COTs $X = \left\{x_1, x_2, \ldots, x_{\lvert X\rvert}\right\}$; order of priority $P = \langle p_1, p_2, \ldots, p_n\rangle$
	Output:	ordered list of COTs $Z = \langle z_1, z_2, \ldots, z_{\lvert X\rvert}\rangle$
1.	$H : \langle h_1, h_2, \ldots, h_{\lvert X\rvert}\rangle$	
2.	$T : \langle t_1, t_2, \ldots, t_{\lvert X\rvert}\rangle$	
3.	**for** $k = 1, 2, \ldots, \lvert X\rvert$ **do**	
4.	$h_k \leftarrow \mathsf{avglum}(\delta_{j,k})$	
5.	$T \leftarrow$ **sort**$(\langle 1, 2 \ldots, \lvert X\rvert\rangle)$	
6.	$Z \leftarrow \langle x_{t_1}, x_{t_2}, \ldots, x_{t_{\lvert X\rvert}}\rangle$	
7.	**return** Z **and exit**	
	Procedure:	**sort**
	Input:	index of list of COTs $T = \langle t_1, t_2, \ldots, t_{\lvert T\rvert}\rangle$
	Output:	index of ordered list of COTs
1.	$L \leftarrow \langle\rangle$, $R \leftarrow \langle\rangle$	
2.	**for** $k = 1, 2, \ldots, \lvert T\rvert$ **do**	
3.	**if** $h_{t_1} > h_{t_k}$ **or** $\left(h_{t_1} = h_{t_k}$ **and** $\Psi(P, x_{t_1}, x_{t_k}) \geq 0\right)$	
4.	$L \leftarrow L^{\wedge}\langle t_k\rangle$	
5.	**else**	
6.	$R \leftarrow R^{\wedge}\langle t_k\rangle$	
7.	**return sort**$(L)^{\wedge}$**sort**(R) **and exit**	

Figure 6: Algorithm to sort code optimization techniques.

The procedure construct in the algorithm accepts a combination C_j and an ordered list of COTs Z as inputs and returns a set of COTs $Y = \{y_1, y_2, \ldots, y_{\lvert Y\rvert}\}$, which can either be empty or includes all the COTs that each $y_j \in Y$ depends on. (The meaning of the symbols $\triangleright$ and $\diamond$ are first introduced and explained in Section 4.6.) Note that the set of COTs returned by construct can only be an empty set or a legitimate combination of COTs that satisfies the concerns. This algorithm is shown in Figure 7.

The main entry of this algorithm iteratively processes all legitimate selections of COTs. After some iterations, when a 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 searching for good solutions.

Determination of Ψ: To help understanding, the algorithm for determining $\Psi(P, x_i, x_j)$ is given in Figure 8.

Complexity of algorithms: The algorithm to determine Ψ can be completed in $O(n)$ time, where n is the number of resource types. The prototype algorithm for sorting code optimization techniques can be completed in $O(\lvert X\rvert \cdot \log(\lvert X\rvert) \cdot n)$ time, where $\lvert X\rvert$ is the number of COTs and n is the count of resource types.

The prototype algorithm for generating combinations iteratively evaluates possible selections until a *local optimum* is found. A disadvantage of this prototype is its high time complexity in the worst case, which is $O\left(2^{\lvert X\rvert} \cdot \left(\lvert X\rvert^2 + n\right)\right)$. On the other hand, we note from the experiment in Section 5 that the algorithm can find solutions much earlier than exhaustive search. We also note that, in practice, we set an affordable upper bound of the number of combinations to be checked to find a solution.

5. Experimental Study

In this section, we first select a few representative types of resource for experiment and set up their calculation formulas G. Then, we construct a simulation 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[5] (*COMM*). Hence, in the following

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

Algorithm: Generation of Combination
Inputs: ordered list of COTs $Z = \langle z_1, z_2, \ldots, z_{|Z|} \rangle$; order of priority P;
basis resource usages $\langle \widetilde{\Gamma}^{\alpha_1}, \widetilde{\Gamma}^{\alpha_2}, \ldots, \widetilde{\Gamma}^{\alpha_m} \rangle$;
list of combinations in lexicographical order $\langle C_1, C_2, \ldots, C_{2^{|X|}} \rangle$
Output: set of COTs $Y = \left\{ y_1, y_2, \ldots, y_{|Y|} \right\}$

1. $Y \leftarrow \emptyset$
2. $last \leftarrow G(\langle \widetilde{\Gamma}^{\alpha_1}, \widetilde{\Gamma}^{\alpha_2}, \ldots, \widetilde{\Gamma}^{\alpha_m} \rangle, \emptyset)$
3. **for** $j = 1, 2, \ldots, 2^{|Z|}$ **do**
4. $\quad Y' \leftarrow$ **construct**(C_j, Z)
5. $\quad curr \leftarrow G(\langle \widetilde{\Gamma}^{\alpha_1}, \widetilde{\Gamma}^{\alpha_2}, \ldots, \widetilde{\Gamma}^{\alpha_m} \rangle, Y')$
6. $\quad$ **if** $\Psi(P, last, curr) < 0$ **then goto step 8**
7. $\quad Y \leftarrow Y'$, $last \leftarrow curr$
8. **return** Y **and exit**

Procedure: **construct**
Inputs: combination $C = \left\{ s_1, s_2, \ldots, s_{|C|} \right\}$;
ordered list of COTs $Z = \langle z_1, z_2, \ldots, z_{|Z|} \rangle$
Output: set of COTs $Y = \left\{ y_1, y_2, \ldots, y_{|Y|} \right\}$

1. $C' \leftarrow C$
2. **foreach** $s_i \in C'$ **do**
3. $\quad$ **for** $j = 1, 2, \ldots, |X|$ **do**
4. $\quad\quad$ **if** $z_{s_i} \rhd z_j$ **then** $C' \leftarrow C' \cup \{j\}$
5. **if** $C' \backslash C \neq \emptyset$ **then** $C \leftarrow C'$ **and goto step 2**
6. **foreach** $s_i, s_j \in C$ **do**
7. $\quad$ **if** $z_{s_i} \diamond z_{s_j}$ **then return** $\emptyset$ **and exit**
8. **foreach** $s_i \in C$ **do**
9. $\quad Y \leftarrow Y \cup \{z_{s_i}\}$
10. **foreach** g_j of G **do**
11. $\quad$ **if** $g_j(\langle \widetilde{\Gamma}_j{}^{\alpha_1}, \widetilde{\Gamma}_j{}^{\alpha_2}, \ldots, \widetilde{\Gamma}_j{}^{\alpha_m} \rangle, Y) \notin \kappa_j$ **then return** $\emptyset$ **and exit**
12. **return** Y **and exit**

Figure 7: Algorithm to generate combination.

Algorithm: Determination of Ψ
Inputs: order of priority $P = \langle p_1, p_2, \ldots, p_n \rangle$;
two sets of resource usages Γ_i, Γ_j
Output: result of comparison: -1, 0, or 1

1. **for** $k = 1, 2, \ldots, n$ **do**
2. $\quad idx \leftarrow p_k$
3. $\quad$ **if** $\delta_{idx,i} < \delta_{idx,j}$ **then return** -1 **and exit**
4. $\quad$ **if** $\delta_{idx,i} > \delta_{idx,j}$ **then return** 1 **and exit**
5. **return** 0 **and exit**

Figure 8: Algorithm to compare two code optimization techniques over a given priority.

experiment, the resource usage can be represented by $\Gamma = \langle \gamma_{CPU}, \gamma_{MEM}, \gamma_{COMM} \rangle$ and the resource constraint by $K = \langle \kappa_{CPU}, \kappa_{MEM}, \kappa_{COMM} \rangle$.

Figure 9 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 the 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[6] written in nesC for the project TOSSIM. TOSSIM is a representative emulator of TinyOS (Levis et al., 2003). The CntToLedsAndRfm application updates a binary counter and sends a radio message containing the current value of the counter to LEDs for display.

A TinyOS application on any node of a wireless sensor network is designed to support only sequentially and periodically executed tasks (Gay et al., 2003).

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

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

Figure 9: Calculation formulas.

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. The original application consists of five components, namely *Main*, *Counter*, *TimerC*, *IntToRfm*, and *IntToLeds*.[7]

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 component is equipped with COTs, some of which have dependencies or conflicting relationships among one another. Figure 10 shows a schematic component-and-connector diagram of the program. The components α_1, α_2, and α_5 are cloned from the *TimerC* component. The components α_3 and α_8 are cloned from the *IntToLeds* component. The component α_6 is cloned from the *Main* component. The component α_7 is cloned from the *Counter* component. The components α_4, α_9, and α_{10} are cloned from the *IntToRfm* component.

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 9. This calculation is illustrated by Figure 11, 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).

[7] Available at http://cse.yeditepe.edu.tr/tnl/html/LOCAL/files/docs/tos-source-tree/apps.CntToLedsAndRfm.CntToLedsAndRfm.nc.html.

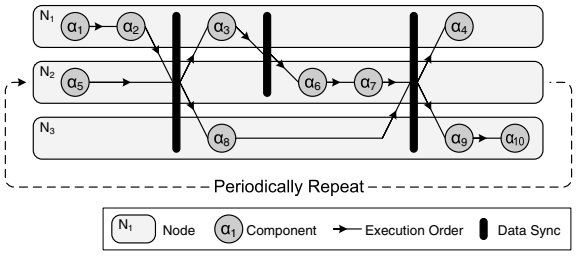

Figure 10: Infrastructure of testbed.

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 and simulation platform 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). Our experiment is conducted on Ubuntu 6.06 LTS Linux with kernel version 2.6.15-28-386.

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.

The COTs used in this experiment include "loop unfolding", "lookup table", "cache", "serialization", and so on. Their optimization effects, basis resource usages, and resource constraints are given in Table 3. These raw data are collected by monitoring the WSN application running on the TOSSIM simulator. Since the resource usages vary greatly in different WSN applications, we do not discuss their absolute values, but always use their normalized values to analyze the performance of our algorithms.

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 are arbitrarily chosen to be 1.5 times the basis resource usages.

We compare the result of our method with three other solutions for code optimization, as shown in Figure 12. The three other solutions include: **(a) Fully**

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

Figure 11: Calculation formulas for the target program.

	Resources		
	CPU (*cycles*)	*MEM* (*KB*)	*COMM* (*bytes*)
Constraints	1050 to 3000	280 to 1440	70 to 360
Basis usages of component	1500 to 2500	400 to 1200	100 to 300
Optimization effects of COTs	-600 to $+900$	-200 to $+300$	-20 to $+30$

Table 3: Raw data used in experiment.

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 (that is, the random solution) are normalized according to the unoptimized solution.

Resource usages are classified into three groups, namely (from top to bottom in Figure 12) *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.

We observe, as expected, that the *CPU* and *COMM* usages are reduced at the expense of *MEM*. For instance, *CPU* is reduced to 0.897, while *MEM* is increased to 1.230. If developers would like to reserve some resources (such as *MEM*) for the other applications, they can adjust the upper bound *max* for the related resources and rerun our algorithm.

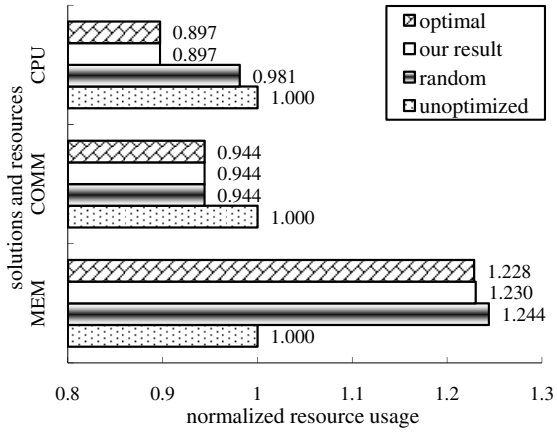

Figure 12: Comparison of solutions.

Since a mote has limited memory, the developers may have already tried their best to fit the existing code into the limited memory (as indicated by the unoptimized bar for *MEM* in Figure 12). One would like to know any alternative options that our approach may offer to the developers. The next experiment studies this question.

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 as inputs and present the results in Figure 13. 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*. Take the first group as an example. It means that *CPU* is given the highest

optimization priority, followed by *MEM*, while *COMM* is given the lowest optimization priority. 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.

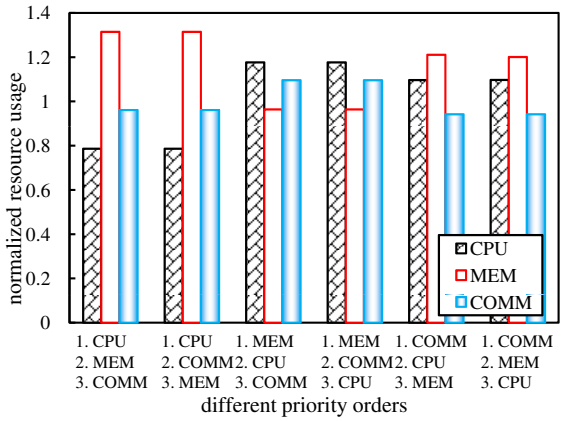

Figure 13: Effects of *P* on resource usage.

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 the increased *MEM*. This is because many COTs are designed to reduce other resources 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 groups 5 and 6 in the figure, may result in disproportionate increases in *CPU* and *MEM* usages. Consider, for instance, the first group again. The *CPU* usage is reduced to 79% of the unoptimized case. Such result comes with the cost of increasing *MEM* to 132% of the unoptimized case. At the same time, the *COMM* usage is also reduced a bit, to 96% of the unoptimized case.

Next, let us consider cases where *MEM* takes the topmost priority in the optimization process. These cases are shown as the third and fourth set of bars in Figure 13. We observe that we can only slightly optimize *MEM* (only a small percentage) at great expenses to *CPU* and *COMM*. It is understandable because mote application usually has been optimized for *MEM*.

We note that, in this experiment, the number of COTs available for individual components is limited. Intuitively, if the developers are not satisfied with the optimization, one way to allow a better search of an optimized solution is to increase the number of COTs for individual (bottleneck) components. Identifying these components may not be difficult because developers can use a memory profiler to find out existing memory usages of components. In the next experiment, we study whether increasing the number of COTs helps resolve the situation.

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 14 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 2 to 3, all the three resource usages are reduced. Take the *COMM* resource as an example. It changes from 91% of the unoptimized case to 83% of that.

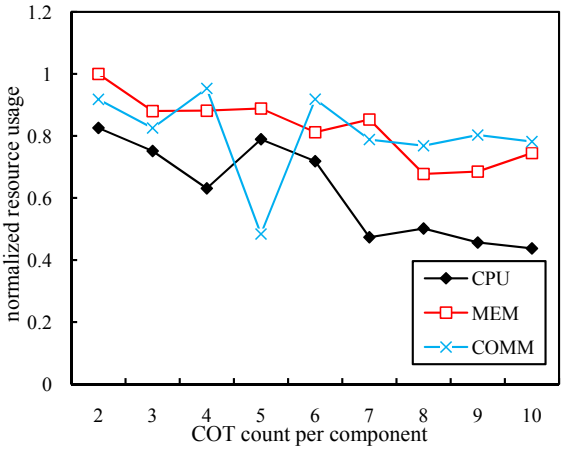

Figure 14: Effects of COT on resource usage

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 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 method to synthesize COTs in WSN applications more attractive.

Variations in solutions for different resource constraints:

When stringent resource usage constraints are given, can our algorithm find effective solutions? We conduct an empirical experiment to address this question.

We randomly select 50 COTs, varying the resource usage constraints from 70% of the basis resource usage to 120% of the usage, and record the resultant usage of the highest prioritized resource. We repeat our experiment by giving the highest priority to the resources *CPU*, *MEM*, and *COMM* in turn. The results are shown in Table 4.

In Table 4, the three rows in turn show the resultant resource usages when the highest priority is given to *CPU*, *MEM*, and *COMM*, respectively. The resource usage constraints are controlled in the range of $[70\%, 120\%]$. We observe that, for the *CPU* and *MEM* resources, when stringent constraints are given (such as limited to less than 80% of the basis resource usage), the resultant resource usages frequently appear as 1.00. It means that no legitimate and effective COT combination can be found. An empty set is therefore returned, so that the resource usage is equal to the unoptimized one. Consider, for example, the column entitled 85%. In each of the three independent tests, no legitimate and effective COT combination can be found.

Performance of solutions in order of iterations:

Since the algorithm iterates all solutions in lexicographical order of COT combinations, no legitimate solution are missed. On the other hand, the algorithm stops at the first encountered minimum point (in the sense of hill-climbing techniques). We are interested in the time cost to find a solution and the performance of the solution thus found.

Since it is not easy to figure out theoretically the capability of such a heuristic algorithm, we also evaluate it experimentally using the testbed subject program. We randomly take 50 COTs into consideration, and check the performance of solution in each iteration. The result is given in Figure 15. This test is very time-consuming (with 2^{50} iterations). Figure 15 shows only part of the searching domain (which includes the solution position) rather than the whole domain.

In Figure 15, the X-axis shows the iterations in order, while the Y-axis shows the normalized resource usages of the solution in each iteration. We notice that the resource usages are all equal to 1 when $X = 0$, which corresponds to the basis resource usages when no code optimization technique is adopted. Since *CPU* resource is of the highest optimization priority and the *MEM* resource is assigned a low priority, the optimization

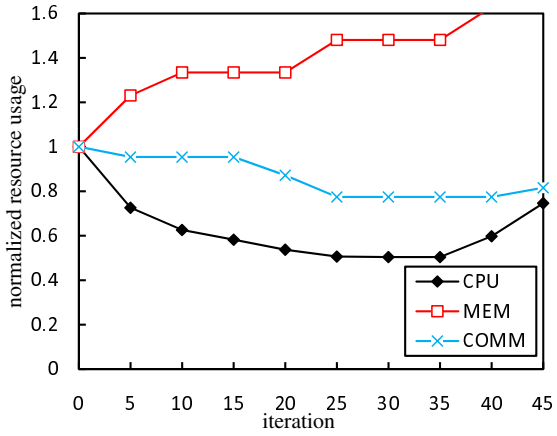

Figure 15: Performance of solutions in order of iterations.

of *CPU* is done at the cost of increasing the usage of *MEM*. The COTs that have high optimization capabilities to reduce *CPU* usage are selected first. When there is less room for reducing *CPU* usage, the COTs that reduce *COMM* usage are combined. Within the given resource usage constraints (which is 150% of the basis resource usage), the first local minimum is identified in iteration 30. The final solution in turn optimizes the usages of *CPU* and *COMM* to 50% and 77%, respectively, at a cost of increasing the *MEM* usage to 148%.

In our experience, the heuristic search algorithm has a high chance of terminating before the first $|X|$ iterations, where $|X|$ is the total number of COTs.

5.5. *Case Study*

In this section, we further evaluate our method on the same target program and simulation platform used in Section 5.

Figure 16 shows the resource optimization capabilities of the 50 randomly selected COTs for program CntToLedsAndRfm. To ease the manual process, we assume that no dependency relations exist among these COTs. Starting from a color palette that represents the optimization capabilities of the COTs, the developer may manually choose the appropriate COTs, refine the choices, and finally work out a COT combination.

The *CPU* resource is given the highest optimization priority, followed by *MEM*, and *COMM* is given the lowest optimization priority. First, we would like to pick out the COTs with a high optimization capability in reducing *CPU* usage. By examining the dark red cells in the first row, we identify eighteen COTs numbered 4, 5, 8, 10, 11, 15, 17, 20, 21, 26, 29, 34, 35, 37, 40,

	120%	115%	110%	105%	100%	95%	90%	85%	80%	75%	70%
CPU	0.97	0.85	0.96	0.81	0.93	1.00	1.00	1.00	1.00	1.00	1.00
MEM	0.94	0.98	0.76	1.00	0.95	0.93	1.00	1.00	1.00	1.00	1.00
COMM	0.69	0.76	0.71	0.76	0.67	0.62	0.71	1.00	0.75	0.72	0.61

Table 4: Resource usage of solution on different resource constraints.

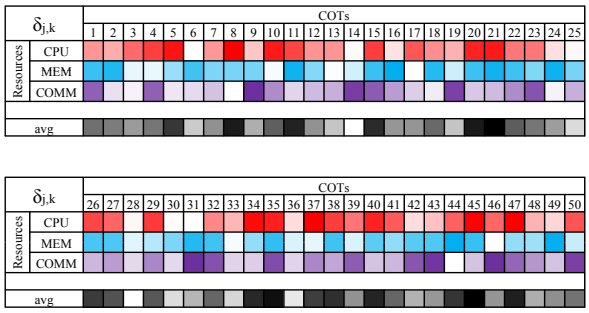

Figure 16: Color palette for program CntToLedsAndRfm.

45, 47, and 50. Then, for the remaining COTs, we pick out the dark blue cells in the second row, which stand for a relatively high optimization capability in reducing *MEM* usage. Ten COTs numbered 1, 2, 6, 16, 22, 27, 31, 32, 38, and 49 are chosen. In the same way, we choose three more COTs numbered 9, 14, and 43, whose colors in the third row are dark purple. Intuitively, they should have a high optimization capability in reducing *COMM* usage. Finally, we choose another three COTs 18, 26, and 27 which are dark gray in color in the fourth row. These COTs also have a high optimization capability in reducing resource usage. Denoting the set of the above selected COTs as Y, the resource usages are estimated using the calculation formulas in Figure 11.

Table 5 shows the basis resource usages (see the row tagged as "Unoptimized"), the resource usages of the manual process (the row tagged as "Manually chosen"), the resource usages of the solution generated by our algorithm (the row tagged as "By algorithm"), and the resource usages of the optimal COT combination (the row tagged as "Optimal", calculated by iterating all possible combinations). The usages have been normalized with respect to the unoptimized case. Let us focus on the row tagged as "Manually chosen" and use it as an example for explanation. It shows that, after applying the COTs in Y, *CPU* usage is reduced to 71% of the unoptimized case. Such optimization comes with the cost of increasing *MEM* usage to 133% of the unoptimized case. At the same time, *COMM* usage is also reduced to 91% of the unoptimized case. The other rows can be explained similarly.

Previously, to meet resource constraints, developers need to manually implement the COTs and iteratively tune the results. Such a process is time-consuming and error-prone. With the use of a color palette, developers can visually compare the optimization capabilities of different COTs and different resources. At the same time, the average optimization capability is also displayed to help select the most valuable and suitable COT. Contrasted with a comparison of numbers, the color palette mechanism is more effective. Developers have an easier time to decide on the appropriate COTs.

In this case study, our algorithm is executed in less than 0.001 second but outperforms the manual process. Although both our algorithm and the manual process cannot, of course, perform better than the optimal solution, the cost to generate the optimal is fairly high (having to iterate 2^{50} possible COT combinations). In real cases, developers may decide on a suitable strategy by weighing between quality and effort.

5.6. Threats to Validity

We summarize the threats to validity in this section.

A construct validity thread is the target program we used. The WSN application used in the experiment is a sample program from the TOSSIM platform. It is very simple and consists of only five components. Real-life applications are more complicated. Hence, we clone components and COTs to increase the complexity of the subject program. In addition, we vary the count of COTs in a controlled range to simulate other cases.

Another construct threat to validity is the use of COTs. In the experiment, we have created various COTs and specified different resource usages to them. The related resource usages are estimated according to our pilot study on monitoring the WSN application running on the TOSSIM simulator. Although they may reflect the resource usages of realistic scenarios in WSN applications, deploying the applications in real WSN hardware environments may give results different from our simulation study.

A threat to internal validity is the assumption that code optimization techniques are applied in WSN applications. We use the term COT to represent all optimization-like patterns used in the programs. To

	Normalized resource usages		
	CPU usage	*MEM* usage	*COMM* usage
Unoptimized	1.00	1.00	1.00
Manually chosen	0.71	1.33	0.91
By algorithm	0.61	1.35	0.82
Optimal	0.50	1.48	0.77

Table 5: Resource usages of various solutions.

obtain reasonable results, we monitor the resource usages of several WSN applications, and specify the raw data used in the experiment accordingly.

The next threat is the complexity of the problem. Although the only safe way to search for the optimal solution is to iterate all possible combinations, the time cost is not acceptable. We have introduced a heuristic algorithm that has a high chance of producing a suboptimal solution within a reasonable time limit. In real-life settings, developers can implement their own version of sorting algorithm and searching algorithm.

A threat to external validity may be due to the resources chosen for experimentation. Resources used in WSN applications vary widely. Our model has, therefore, been designed for the general situation and does not depend on the types of resource used. We have taken three representative kinds of resource for study in the experiment and set up the corresponding calculation formulas.

We set up our model based on the TinyOS platform since it is the most widely used platform for WSN applications.

There is, however, no guarantee that our model work for other WSN platforms. To address this validity threat, we design our model from the perspective of tasks and components. It is independent of hardware and software architectural issues (such as processes, threads, and concurrency). It should be portable to other platforms easily.

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 conducted through estimated usages of resources based on the infrastructure of an application under study. We provide developers with a color palette to help them visualize the optimization capabilities of the COTs and to manually choose a favorable combination. We also present a heuristic algorithm that automatically determines a suboptimal combination. An experimental study shows that our heuristic algorithm produces a promising solution to code optimization. A case study further demonstrates the effectiveness of our visualization mechanism. As future work, we will conduct experiment in real-life WSN hardware environments. And 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 method.

References

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

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

Chan, W.K., Chen, T.Y., Cheung, S.C., Tse, T.H., Zhang, Z., 2007. Towards the testing of power-aware software applications for wireless sensor networks. In: Reliable Software Technologies: Ada-Europe 2007, Abdennadher, N., Kordon, F. (Eds.), Lecture Notes in Computer Science, vol. 4498. Springer, Berlin, Germany, pp. 84–99.

Gay, D., Levis, P., Culler, D., 2005. 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.

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D., 2003. 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.

Healy, M., Newe, T., Lewis, E., 2007. Power management in operating systems for wireless sensor nodes. In: Proceedings of the 2007 IEEE Sensors Applications Symposium (SAS 2007). IEEE Computer Society, Los Alamitos, CA, pp. 1–6.

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

Kaspersky, K., 2003. Code Optimization: Effective Memory Usage. A-List Publishing, Wayne, PA.

Kuchcinski, K., 1997. Embedded system synthesis by timing constraints solving. In: Proceedings of the 10th International Symposium on System Synthesis (ISSS '97). IEEE Computer Society, Los Alamitos, CA, pp. 50–57.

Kulkarni, P., Hines, S., Hiser, J., Whalley, D., Davidson, J., Jones, D., 2004. Fast searches for effective optimization phase sequences. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI 2004). ACM SIGSOFT Software Engineering Notes 39 (6), 171–182.

Kuorilehto, M., Hannikainen, M., Hamalainen, T.D., 2005. A survey of application distribution in wireless sensor networks. EURASIP Journal on Wireless Communications and Networking 5 (5), 774–788.

Levis, P., Lee, N., Welsh, M., Culler, D., 2003. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In: Proceedings of the 1st ACM Conference on Embedded Networked Sensor Systems (SenSys 2003). ACM, New York, NY, pp. 126–137.

Özcan, E., Onbaşioglu, E., 2007. Memetic algorithms for parallel code optimization. International Journal of Parallel Programming 35 (1), 33–61.

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

Shin, I., Lee, I., Min, S.L., 2004. 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.

Shnayder, V., Chen, B.-R., Lorincz, K., Fulford Jones, T.R.F., Welsh, M., 2005. Sensor networks for medical care. In: Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SenSys 2005). ACM, New York, NY, p. 314.

Srivastava, M., 2006. 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.

Szewczyk, R., Mainwaring, A., Polastre, J., Anderson, J., Culler, D., 2004. An analysis of a large scale habitat monitoring application. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems. ACM, New York, NY, pp. 214–226.

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

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

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

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

Zhao, M., Childers, B.R., Soffa, M.L., 2006. An approach toward profit-driven optimization. ACM Transactions on Architecture and Code Optimization 3 (3), 231–262.