A Design Framework for Real-Time Embedded Systems with Code Size and Energy Constraints

SHEAYUN LEE
Samsung Electronics Co., Ltd.
INSIK SHIN
University of Pennsylvania
WOONSEOK KIM
Samsung Electronics Co., Ltd.
INSUP LEE
University of Pennsylvania
and
SANG LYUL MIN
Seoul National University

Real-time embedded systems are typically constrained in terms of three system performance criteria: space, time, and energy. The performance requirements are directly translated into constraints imposed on the system's resources, such as code size, execution time, and energy consumption. These resource constraints often interact or even conflict with each other in a complex manner, making it difficult for a system developer to apply a well-defined design methodology in developing a real-time embedded system. Motivated by this observation, we propose a design framework that can flexibly balance the tradeoff involving the system's code size, execution time, and energy

This research was supported in part by NSF CNS-0410662, NSF CNS-0509143, ARO W911NF-05-1-0182, MIC & IITA through the IT Leading R & D Support Project, MIC under the IT-SoC Project, the Ministry of Education under the Brain Korea 21 Project, and the Korean Ministry of Science and Technology under the National Research Laboratory program. ITC at Seoul National University provided research facilities for this study.
The work was done while S. Lee and W. Kim were students at Seoul National University.
Authors' addresses: Sheayun Lee, Memory Division, Samsung Electronics Co., Ltd., Hwasung City, Gyeonggi-Do 445-701, Korea; email: sheayun.lee@samsung.com. Insik Shin and Insup Lee, University of Pennsylvania, 3330 Walnut St., Philadelphia, Pennsylvania 19104; email: {ishin,lee}@cis.upenn.edu. Woonseok Kim, Digital Media R&D Center, Samsung Electronics Co., Ltd. Suwon City, Gyeonggi-Do 443-742, Korea; email: woonseok.kim@samsung.com. Sang Lyul Min, Seoul National University, Seoul 151-742, Korea; email: symin@dandelion.snu.ac.kr.

consumption. Given a system specification and an optimization criteria, the proposed technique generates a set of design parameters in such a way that a system cost function is minimized while the given resource constraints are satisfied. Specifically, the technique derives code generation decision for each task so that a specific version of code is selected among a number of different ones that have distinct characteristics in terms of code size and execution time. In addition, the design framework determines the voltage/frequency setting for a variable voltage processor whose supply voltage can be adjusted at runtime in order to minimize the energy consumption while execution performance is degraded accordingly. The proposed technique formulates this design process as a constrained optimization problem. We show that this optimization problem is NP-hard and then provide a heuristic solution to it. We show that these seemingly conflicting design goals can be pursued by using a simple optimization algorithm that works with a single optimization criteria. Moreover, the optimization is driven by an abstract system specification given by the system developer, so that the system development process can be automated. The results from our simulation show that the proposed algorithm finds a solution that is close to the optimal one with the average error smaller than 1.0%.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

General Terms: Algorithm, Design, Performance

Additional Key Words and Phrases: Embedded, code size, energy, real-time, scheduling

ACM Reference Format:

Lee, S., Shin, I., Kim, W., Lee, I., and Min, S. L. 2008. A design framework for real-time embedded systems with code size and energy constraints. ACM Trans. Embedd. Comput. Syst. 7, 2, Article 18 (February 2008), 27 pages. DOI = 10.1145/1331331.1331342 http://doi.acm.org/10.1145/1331331.1331342

1. INTRODUCTION

Embedded systems are characterized by the diversity of application areas, ranging from simple nodes in a sensor network to large-scale complex systems, such as flight control and factory automation. Because of this diversity, embedded systems typically have requirements in different aspects of system performance, which are, in turn, expressed as constraints on system resources, such as space, time and energy. For example, in a cost-sensitive system, it may be desirable to generate as small code as possible, since the amount of memory used by application programs affect the component cost for memory modules. On the other hand, many real-time systems have timing requirements that are expressed in terms of deadlines that tasks must meet even in the worst-case. For this type of system, the designer must guarantee the worst-case timing performance since the system may function incorrectly without such a guarantee. Guaranteeing the worst-case performance, however, could result in larger code. Another constraint commonly found in mobile embedded systems is on the energy consumption of the system, since an increasing number of them are powered by batteries. Moreover, in general, most systems have a mixture of these different requirements that interact or even conflict with each other in a complex manner.

These different performance requirements, combined with other types of constraints, such as implementation cost and time-to-market requirement, make it difficult for a system designer to apply a well-defined methodology in developing

embedded systems. Motivated by this observation, we propose an embedded system design framework that can be used to balance the tradeoff relationship among code size, execution time, and energy consumption. The proposed framework aims at providing a system designer with a parameterized view of the system development process, where the system can be flexibly fine-tuned with regard to the different performance criteria.

The tradeoff between code size and execution time is assumed to be enabled by providing different versions of code for the same application program, which have different characteristics in terms of code size and execution time. Specifically, we incorporate a code generation technique for a dual instruction set processor that exploits the tradeoff between a program's code size and its WCET (worst-case execution time) [Lee et al. 2004]. A dual instruction set processor [Lee et al. 2003] supports a reduced (compressed) instruction set in addition to a full (normal) instruction set. By using the reduced instruction set in code generation, an application program's code size can be significantly reduced, while its execution time is increased. Such dual instruction set processors provide a mechanism to dynamically switch the instruction set mode in which the processor executes [Furber 1996]. Therefore, using different instruction sets selectively for different program sections, we are able to flexibly balance the tradeoff between a program's code size and execution time.

On the other hand, we assume that the tradeoff between execution time and energy consumption is enabled by using a variable voltage processor. Such a processor provides a mechanism to reduce the energy consumption by lowering the processor's supply voltage at runtime. However, when the supply voltage is lowered, the operating clock frequency must be lowered accordingly, which leads to an increased execution time for programs. Therefore, by setting an appropriate voltage/frequency level, we can exploit the tradeoff relationship between the processor's execution speed and energy consumption. The technique for dynamically adjusting the supply voltage and the clock frequency is called a DVS (dynamic voltage scaling) technique.

Figure 1 shows the overall structure of the proposed design framework. The design framework receives two sets of inputs: a system specification and an optimization criteria. In the system specification, the task set defines the functionality of the system, along with the tradeoff data between the code size and execution time of each program. Besides, the system specification includes the scheduling policy used to schedule the tasks, which must meet the timing requirements associated with each of them. Finally, the hardware model describes the hardware platform on which the applications are executed. This includes the tradeoff data between the execution speed, i.e., the clock frequency, and the energy consumption of the processor.

The optimization criteria are assumed to be specified by the system developer, which describe the resource constraints and the goal of the design optimization. The resource constraints are specified in terms of space, time, and energy. In other words, they consist of the code size, the timing and the energy constraints imposed on the system. On the other hand, the goal of the design optimization is given in the form of an objective function that captures the system cost that is desired to be minimized.

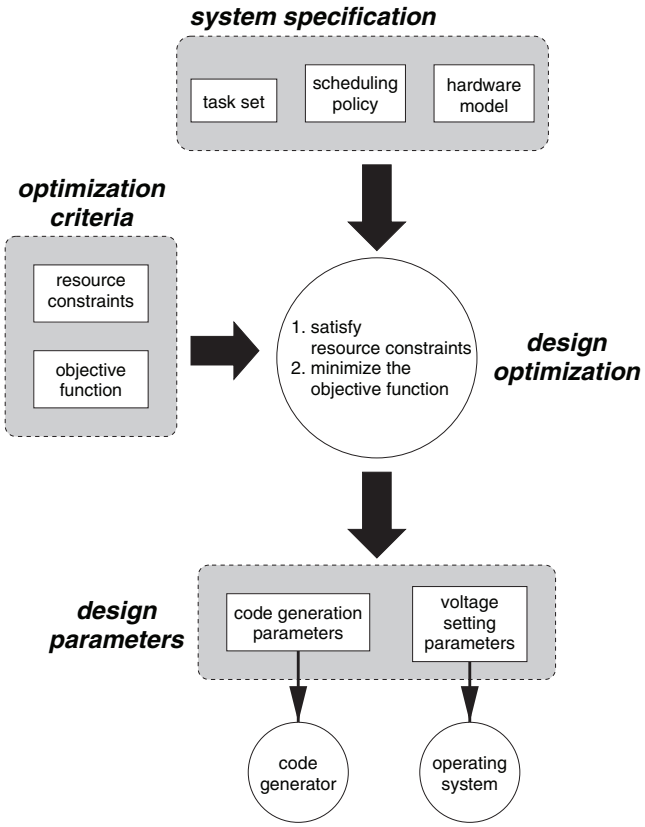

Fig. 1. Overall structure of the proposed design framework.

Given these inputs, the design framework generates a set of design parameters in such a way that the resulting system satisfies the constraints imposed on different aspects of the system resources and, at the same time, minimizes the system cost function. The design parameters being derived are: (1) the code generation parameters that are given to the code generator to select a version of code for each application program and (2) the voltage setting parameters that are given to the operating system so that it can adjust the voltage/frequency of the processor at runtime. By deriving these design parameters that are used at different levels of system development at the same time, the proposed design framework can effectively exploit the tradeoff involving space, time, and energy.

The main contribution of the paper can be summarized as follows. First, we identify the tradeoff relationship involving code size, execution time, and energy consumption, and propose a system optimization framework that flexibly balances this tradeoff. Second, we formulate a multidirectional optimization problem, show the NP-hardness of the problem, and provide a heuristic solution to it. Finally, we develop a framework where system design parameters can be systematically drived from an abstract specification of system requirements, which are given by the system developer.

The rest of the paper is organized as follows. In Section 2, we describe the system model and assumptions, and formulate an optimization problem that we address. Section 3 details the algorithm for deriving the system design parameters. We present the results from simulations in Section 4. In Section 5, we discuss the issues of extending our proposed framework by relaxing some assumptions. We present a survey of related work in Section 6 and finally conclude the paper in Section 7.

2. SYSTEM MODEL AND PROBLEM DEFINITION

We describe the system model and assumptions of our proposed framework in Section 2.1. Based on this, we formally define our problem of design parameter derivation in Section 2.2.

2.1 System Model and Assumptions

We assume that the system consists of a set of tasks $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$, where n denotes the number of tasks. Each task τ_i is characterized by $\langle p_i, X_i \rangle$ as follows:

—*Period p_i:* the fixed time interval between the arrival times of two consecutive requests of τ_i. We assume each task has a relative deadline equal to its period. We further assume that the tasks are scheduled under the EDF (earliest-deadline-first) scheduling policy, which has been shown to be an optimal dynamic-priority scheduling algorithm [Liu and Layland 1973].

—*Execution Descriptor List X_i:* the list that enumerates the possible pairs of code size s_i and WCEC (worst-case execution cycles)[1] c_i of τ_i, under the assumption that τ_i has multiple versions of executable code. That is,

$$X_i = \{x_{i,j} = \langle s_{i,j}, c_{i,j} \rangle | j = 1, 2, \ldots, K_i\},$$

where $x_{i,j}$ denotes the jth element of X_i, called an execution descriptor, $s_{i,j}$ and $c_{i,j}$ give the code size and WCEC of the jth version of executable code of τ_i, and K_i denotes the number of elements in X_i. In other words, the execution descriptor list for a task essentially summarizes the tradeoff relationship between the task's code size and worst-case execution time.

We assume that each task's execution descriptor list is derived by the selective code transformation technique [Lee et al. 2004], which utilizes a dual-instruction set processor. The greedy nature of this technique ensures that the execution descriptor list X_i satisfies the following two properties:

—The code size $s_{i,j}$ increases while the WCEC $c_{i,j}$ decreases as the index j increases. That is, $\Delta^s_{i,j} = s_{i,j} - s_{i,j-1} > 0$ and $\Delta^c_{i,j} = c_{i,j} - c_{i,j-1} < 0$, $\forall i \in [1, n]$ and $\forall j \in [2, K_i]$.

[1]Rather than the worst-case execution time (WCET), we use the notion of the worst-case execution cycles (WCEC), that is the number of execution cycles in the worst case to execute for the completion of τ_i, since the execution time of programs can vary depending on runtime settings of the voltage/frequency of the processor.

—The marginal gain in the WCEC reduction for the unit increase in the code size is monotonically nonincreasing, i.e.,

$$\frac{|\Delta^c_{i,j+1}|}{\Delta^s_{i,j+1}} \leq \frac{|\Delta^c_{i,j}|}{\Delta^s_{i,j}}, \quad \forall i \in [1, n], \forall j \in [2, K_i - 1]$$

Note that the minimum number of execution descriptors for a task is 2, because $x_{i,1}$ corresponds to the program compiled entirely into the reduced instruction set (minimum code size and maximum execution cycles), and x_{i,K_i} to the program compiled entirely into the full instruction set (maximum code size and minimum execution cycles).

We assume that the processor supports variable voltage and clock frequency settings, where the operating frequency of the clock is proportional to the supply voltage. By lowering the supply voltage and thus the clock frequency, the processor's energy consumption can be reduced while execution performance is degraded to a certain extent. We assume that the operating frequency can be set at a continuous level. That is, if we let f_i denote the CPU frequency at which τ_i executes, $f_i \in (0, f_{max}]$, where f_{max} denotes the maximum clock frequency at which the processor can run. We assume that the execution time of a program is inversely proportional to the frequency setting. That is, if we denote the execution time of a task τ_i by t_i and the clock frequency by f, it holds that $t_i \propto 1/f$. In addition, since we assume that the energy consumption of the processor is proportional to the supply voltage squared [Chandrakasan et al. 1992] and that the supply voltage and the clock frequency are in a direct proportional relationship. That is, if we let E denote the processor's energy consumption, it holds that $E \propto f^2$.

For each task τ_i, we now define its energy consumption e_i as proportional to its WCEC and its CPU frequency squared [Chandrakasan et al. 1992] as follows:

$$e_i = \kappa \frac{P}{p_i} c_i f_i^2 \tag{1}$$

where κ gives the energy consumption constant and $P = LCM^n_{i=1}(p_i)$ denotes the hyperperiod, i.e., the least common multiple of periods of all the tasks. The equation implicitly assumes that the energy consumed in a cycle is independent of whether the processor is executing the reduced instruction or the full instruction set. This assumption holds for the target processor (ARM7TDMI) considered in this paper [Chang et al. 2002], but may not hold for other dual-instruction set processors. Then, we define the system energy consumption E as the total energy consumption of all tasks in a hyperperiod, i.e.,

$$E = \sum_{i=1}^{n} e_i \tag{2}$$

Finally, we define the system code size S as the sum of code sizes of all the tasks, i.e.,

$$S = \sum_{i=1}^{n} s_i \tag{3}$$

2.2 Problem Definition

With the assumptions described in the previous section, we define our design optimization problem, called the SETO (size-energy-time optimization) problem. Given the timing constraint of each task and a list of possible pairs of code size and WCEC of each task, the SETO problem is to determine the code size, WCEC, and CPU frequency of each task such that the system cost function is minimized subject to the system's resource constraints in terms of space, time, and energy.

The inputs and outputs of the SETO problem are as follows:

—**Input: task set T.** The inputs of the SETO problem are (1) the timing constraint and (2) the code generation information of individual tasks. The timing constraint of task τ_i is given by its period p_i. The code generation information of task τ_i is given by its execution descriptor list X_i.

—**Output: $\langle V, F \rangle$.** The outputs of the SETO problem are (1) an execution descriptor that guides code generation decision for a task and (2) voltage (frequency) setting that determines the DVS setting for that task. We define an execution descriptor assignment vector $V = \langle v_1, v_2, \ldots, v_n \rangle$, where $v_i = j$ indicates that the execution descriptor $x_{i,j}$ is assigned to task τ_i, and that the task's code size is $s_i = s_{i,j}$ while its WCEC is $c_i = c_{i,j}$. We also define a frequency assignment vector $F = \langle f_1, f_2, \ldots, f_n \rangle$, where f_i denotes the frequency setting for task τ_i.[2]

As a constrained optimization problem, the SETO problem has the following objective function and constraints:

—**Optimization:** To capture the resource costs in terms of memory space and energy consumption considering their relative importance, we define the system cost function as a linear combination of the system code size and the system energy consumption, normalized to the upper bound imposed on each. That is,

$$f(S, E) = \alpha \frac{S}{\bar{S}} + \beta \frac{E}{\bar{E}} \tag{4}$$

where S denotes the system code size, E denotes the system energy consumption, and coefficients α and β are constants, which we assume the system designer provides to indicate the relative importance of code size and energy consumption constraints, respectively. We also assume that the upper bound on the system code size ($\bar{S}$) and that on the system energy consumption ($\bar{E}$) are given by the system designer in the form of the design constraints, as will be shortly discussed.

Note that the system cost function does not take the timing behavior into account. Since in a hard real-time system, the timing requirement only serves as a constraint and does not affect the value or cost of the system, the objective function does not include the notion of time.

[2]In the problem formulation under discussion, $f_{sys} = f_1 = f_2 = \cdots = f_n$ for all the tasks in the task set, since setting them all equal provides the optimal solution in terms of energy consumption under the EDF scheduling algorithm. However, this does not hold for a different scheduling algorithm, for example, the rate-monotonic (RM) algorithm [Shin et al. 2000; Saewong and Rajkumar 2003].

—**Constraints:** The system's resource requirements are defined in terms of constraints imposed on space, time, and energy, as follows:
- Code size constraint. The system code size S must not exceed a given upper bound $\bar{S}$.
- Timing constraint. The task set must be schedulable under the given scheduling policy, i.e., all the task instances must finish execution before their respective relative deadlines, which are equal to their periods, respectively. We can check whether the timing constraint on the task set is satisfied using a simple schedulability test, since the tasks are scheduled by the EDF scheduling algorithm. The schedulability test calculates the system utilization and determines that the whole task set is schedulable if the utilization is below 1.0 [Liu and Layland 1973].
- Energy constraint. The system energy consumption E must not exceed a given upper bound $\bar{E}$.

In real-world applications, reducing code size may not linearly increase the benefit (or interchangeably, reduce the cost) of the system. That is, provided that the total code size of the system fits in the total available memory capacity, further reducing the code size may not affect the system's value. In such a situation, a more viable approach will be one that models the code size constraint and/or the objective function using a discrete step function. The model described in this paper in its current form does not allow the use of nonlinear function in the system value function. However, the problem with a discrete code size cost function can be formulated as a set of smaller subproblems with different (distinct) code size constraints and then a best solution can be selected among the solutions to these smaller subproblems.

The SETO problem is, given a task set, to find the execution descriptor assignment vector V and the frequency assignment vector F such that the following system cost function is minimized

$$f(S, E) = \alpha \frac{S}{\bar{S}} + \beta \frac{E}{\bar{E}} \tag{5}$$

subject to

$$S = \sum_{i=1}^{n} s_i \leq \bar{S} \tag{6}$$

$$U = \sum_{i=1}^{n} \frac{t_i}{p_i} \leq 1.0 \tag{7}$$

$$E = \kappa \sum_{i=1}^{n} \frac{P}{p_i} c_i f_i^2 \leq \bar{E} \tag{8}$$

where $t_i = c_i / f_i$ denotes the (worst case) execution time of task τ_i. Inequalities 6, 7, and 8 guarantee the size, timing, and energy constraints, respectively.

We present the following theorem to show that the SETO problem is intractable.

THEOREM 2.1. *The SETO problem is NP-hard.*

PROOF. The proof is via a polynomial-time reduction from the subset sum problem that is known to be NP-complete [Garey and Johnson 1979]. Let a set of positive integers $A = \{a_1, \ldots, a_n\}$ and k represent an instance of the subset sum problem. Assume that for each i, $1 \leq i \leq n$, $a_i \geq 1$ and $\sum_{i=1}^{n} a_i = M$.

For reduction, for each i, $1 \leq i \leq n$, we first construct the execution descriptor list X_i of τ_i such that $X_i = \{\langle \epsilon, a_i - \epsilon \rangle, \langle a_i - \epsilon, \epsilon \rangle\}$, where $\epsilon < 1/n$. We pick the coefficients of the system cost function, α and β, such that $\alpha = 1$ and $\beta = 0$. We also pick the system code size upper bound $\bar{S}$ and the system energy consumption upper bound $\bar{E}$ such that $\bar{S} = M$ and $\bar{E} = \kappa M$. We pick the system-level CPU frequency F_{sys} such that $F_{sys} = 1.0$, and thus the CPU frquency f_i of each τ_i is constructed as $f_i = 1.0$. We finally construct the period p_i of τ_i such that $p_i = k$. The energy consumption e_i of τ_i is κc_i.

In any schedule, $\sum_{i=1}^{n}(c_i + s_i) = M$. The system code size and energy consumption constraints are then always met; $\sum_{i=1}^{n} s_i < \bar{S}$ and $\sum_{i=1}^{n} e_i < \bar{E}$. In any feasible schedule, the timing constraint should be met; $\sum_{i=1}^{n} c_i \leq k$. Then, in any feasible schedule, $\sum_{i=1}^{n} s_i \geq M - k$. Considering $\epsilon < 1/n$, we know that the minimum system code size is between $M - k$ and $M - k + 1$. Considering $\alpha = 1$ and $\beta = 0$, the system cost function is minimized if, and only if, the system code size is minimized. The minimum system code size is achieved if, and only if, there is a subset A' of A such that the sum of the elements of A' is k. □

Given the difficulty of the SETO problem, a natural approach is to develop a heuristic algorithm that can effectively address the SETO problem.

3. OPTIMIZATION ALGORITHM

We propose an algorithm that assigns to each task an execution descriptor and the processor's operating frequency in such a way that the assignment meets the design goals described in the previous section. The algorithm consists of two distinct phases. In the first phase, it tries to satisfy the constraints imposed on the system's code size, timing behavior, and energy consumption, while the second phase is aimed at minimizing the system cost function. The first phase begins with the smallest code size possible for each task by using an initial execution descriptor assignment vector $V = \langle 1, 1, \ldots, 1 \rangle$. Then, aiming at meeting the timing and the energy constraints, the algorithm iteratively transforms the assignment vector. In each iteration of the algorithm, it selects a task whose code size is to be increased while the system code size does not exceed the given upper bound. In return for the increased code size of the selected task, the workload generated by the task is reduced, which not only makes the task set more likely to be schedulable, but also reduces the system's energy consumption. Specifically, assuming the current execution descriptor assignment vector is $V = \langle v_1, v_2, \ldots, v_n \rangle$, each iteration of the algorithm selects an execution descriptor from the candidate set $\{x_{i,j} | i \in [1, n] \text{ and } j \in [v_i + 1, K_i]\}$[3] and assigns

[3]If $v_i = K_i$ for a task τ_i, then the candidate set does not contain any execution descriptor for that task.

$v_i = j$. The selection is made in such a way that the system workload is reduced as much as possible for the unit increase in the code size.

Once both the timing and the energy constraints are met, the first phase is terminated and the second phase begins. In the second phase, the algorithm continues transforming the execution descriptor assignment vector by selecting an execution descriptor and assigning it to the corresponding task, where the selection is geared toward minimizing the system cost function. Note that the resource constraints remain satisfied during the second phase. Since the transformation reduces the system workload, the timing and energy constraints cannot be violated, while the code size constraint is checked in every iteration of the transformation. The algorithm terminates when (1) selecting any of the execution descriptors remaining in the candidate set would violate the system code size constraint, or (2) no further reduction of the system cost function can be made by the transformation, i.e., the system cost function has been minimized.

After the algorithm finishes, the clock frequencies for tasks $F = \langle f_1, f_2, \ldots, f_n \rangle$ are assigned with a single value for all the tasks, i.e., $f_i = f$, $\forall i \in [1, n]$, which is shown to be optimal in terms of energy consumption under the EDF scheduling policy [Saewong and Rajkumar 2003]. Calculating the optimal setting for the clock frequency f will be discussed later in Section 3.1.

In Sections 3.1 and 3.2, we show that throughout the transformation process, a single selection criteria can be used that favors the execution descriptor with the maximum ratio of reduction in the system workload to the increase in the code size. In Section 3.3, we will show that a simple greedy heuristic can be employed for this purpose, which has a low complexity but yet produces near-optimal results.

3.1 Phase 1: Satisfying the Timing and Energy Constraints

The first phase of the assignment algorithm repeats selecting a task and increasing its code size until the timing and energy constraints are satisfied. The selection is made in such a way that both constraints are met with the increase in the system code size as small as possible. Therefore, in selecting an execution descriptor, the algorithm favors the one with the maximum reduction of the system workload for the unit increase in code size, as will be described in the following.

As explained earlier in Section 2.2, we can check whether the timing constraint is met by using a simple schedulability test that compares the system utilization against the upper bound of 1.0. To be more specific, the deadlines of all the tasks are met if, and only if, the processor utilization is less than or equal to 1.0, assuming that all the task instances are executed using the maximum frequency of the processor. If we use the same operating frequency f for all the task instances, the processor utilization can be represented as a function of the frequency. That is, $U(f) = \sum_{i=1}^{n} t_i/p_i = (1/f) \sum_{i=1}^{n} c_i/p_i$, and the schedulability condition states that $U(f_{max}) \leq 1.0$. Furthermore, under the EDF scheduling algorithm, the optimal clock frequency setting is to use the same clock frequency for all the task instances in such a way that makes the processor utilization

equal to 1.0 [Saewong and Rajkumar 2003]. That is, if $U(f_{max})$ is less than or equal to 1.0, setting $f_i = f_{max} \times U(f_{max})$, $\forall i \in [1, n]$ guarantees that the timing constraints of all the tasks are met, and at the same time minimizes the energy consumption of the task set.

Based on this observation, if the task set is not schedulable with the initial execution descriptor assignment vector $V = \langle 1, 1, \ldots, 1 \rangle$, the algorithm tries to lower the task set's processor utilization by selecting a task and reduce its WCEC while increasing a certain amount of code size. For this purpose, the algorithm selects an execution descriptor from the candidate set that has the largest ratio of the reduction of utilization to the increase in the code size. In order to estimate the reduction of the processor utilization by selecting an execution descriptor, we rewrite the equation for the utilization as

$$U(f) = \frac{1}{f} \sum_{i=1}^{n} \frac{c_i}{p_i} = \frac{1}{fP} \sum_{i=1}^{n} \left(\frac{P}{p_i} \right) c_i = \frac{1}{fP} \sum_{i=1}^{n} w_i \tag{9}$$

where $w_i = (P/p_i)c_i$ gives the number of clock cycles required by all the invocations of task τ_i during a hyperperiod. Based on this, the algorithm selects the execution descriptor $x_{i,j}$ with the maximum value of $|\Delta w_{i,j}|/\Delta s_{i,j}$, where $\Delta w_{i,j} = w_{i,j} - w_i$ and $\Delta s_{i,j} = s_{i,j} - s_i$. We call this ratio $|\Delta w_{i,j}|/\Delta s_{i,j}$ the *workload reduction factor* of the execution descriptor $x_{i,j}$ and it is used throughout the entire transformation process.

On the other hand, the energy consumption by the processor in a hyperperiod is calculated by $E = \kappa \sum_{i=1}^{n} (P/p_i) c_i f_i^2$, as in Eq. (8) presented earlier in Section 2.2. If we set the operating frequency of all the tasks to be $f_i = f$, the energy consumption becomes $E = \kappa f^2 \sum_{i=1}^{n} w_i$. Letting $f = f_{max} \times U(f_{max}) = \frac{1}{P} \sum_{i=1}^{n} w_i$ gives

$$E = \frac{\kappa}{P^2} \left(\sum_{i=1}^{n} w_i \right)^3 \tag{10}$$

Therefore, in order to satisfy the energy constraint with the smallest increase in the total code size, the algorithm selects the execution descriptor $x_{i,j}$ with the maximum workload reduction factor, which is the same selection criteria used in satisfying the timing constraint.

This means that the first phase of the algorithm can be driven by a single selection criteria for the execution descriptor to achieve its goal of satisfying both the timing and energy constraints with the smallest increase in the code size. That is, the algorithm incrementally transforms the execution descriptor assignment vector by selecting in each iteration an execution descriptor $x_{i,j}$ with the maximum workload reduction factor among the ones in the candidate set. This selection is repeated until both the constraints are met, after which the algorithm moves on to its second phase where it tries to minimize the system cost function. In the case where either of the two constraints cannot be satisfied by the incremental transformation, the algorithm terminates and determines that there is no assignment of execution descriptors that makes the task set feasible under the given set of constraints.

3.2 Phase 2: Minimizing the System Cost Function

Once the timing and the energy constraints are met, the algorithm tries to minimize the system cost function by balancing the tradeoff between the system code size and the energy consumption. As mentioned earlier in Section 2.2, the system cost function is represented as a linear combination of the system code size and the system energy consumption, which is given by

$$f(S, E) = \alpha \frac{S}{\bar{S}} + \beta \frac{E}{\bar{E}} = \frac{\alpha}{\bar{S}} \sum_{i=1}^{n} s_i + \frac{\beta \kappa}{P^2 \bar{E}} \left(\sum_{i=1}^{n} w_i \right)^3 \tag{11}$$

where nonnegative constants α and β reflect the relative importance of the system code size and the system energy consumption constraints, respectively. The second phase of the algorithm uses the same strategy in transforming the execution descriptor assignment vector by selecting a task and increasing its code size in exchange for reduced energy consumption. In doing this, the value of the system cost function may increase or decrease depending on the amount of code size increase and the energy consumption reduction. Therefore, the algorithm should pay attention to the change in the system cost function, as explained in the following.

Suppose that the algorithm has selected an execution descriptor $x_{i,j}$ to be used in transforming the execution descriptor assignment vector. The system code size then increases, whereas the total energy consumption is reduced, because (1) the total workload generated by the task set is reduced and (2) the operating frequency can be lowered accordingly. That is, the value of the system cost function after the transformation is $f' = \alpha(S'/\bar{S}) + \beta(E'/\bar{E})$, where

$$\begin{cases} S' = S + \Delta S & (\Delta S > 0) \\ E' = E - \Delta E & (\Delta E > 0) \end{cases} \tag{12}$$

The reduction of the system cost function can be calculated by $\Delta f = f - f' = \beta(\Delta E/\bar{E}) - \alpha(\Delta S/\bar{S})$. If we let $\Delta E = \gamma \Delta S$, we have $\Delta f = (\gamma\beta/\bar{E} - \alpha/\bar{S})\Delta S$. Therefore, for the reduction Δf to be positive, the selection by the algorithm should have $\gamma > \alpha\bar{E}/\beta\bar{S}$, since we assume $\alpha > 0$ and $\beta > 0$. Furthermore, in order to achieve the maximum reduction of the cost function, the algorithm should select the execution descriptor that corresponds to the largest value of γ, i.e., the largest ratio of ΔE to ΔS. Since the largest reduction of the energy consumption can be achieved by the largest reduction of the system workload, the above mentioned selection is equivalent to selecting the one with the maximum workload reduction factor, which is the same selection criteria used in the first phase.

Therefore, the second phase of the algorithm selects the execution descriptor $x_{i,j}$ with the maximum workload reduction factor, provided that $\Delta E/\Delta S > \alpha\bar{E}/\beta\bar{S}$ and $\Delta s_{i,j} \leq \bar{S} - S$. If assigning the execution descriptor with the maximum workload reduction factor would increase the system cost function, i.e., if $\Delta E/\Delta S \leq \alpha\bar{E}/\beta\bar{S}$, selecting any other execution descriptor would degrade the system cost function as well. In such a case, the optimization process is terminated and the final solution is generated. On the other hand, if assigning the execution descriptor with the maximum workload reduction factor would

violate the code size constraint, i.e., if $\Delta s_{i,j} > \bar{S} - S$, the algorithm checks the one with the next largest workload reduction factor by resuming the selection procedure after eliminating from the candidate set the execution descriptor with the maximum workload reduction factor. If there remains no execution descriptor in the candidate set, this means that no task's workload can be further reduced, in which case the algorithm finishes with the current assignment. The next section describes the assignment algorithm and discusses its properties and complexity.

3.3 The Assignment Algorithm

Figure 2 shows the ALG algorithm that assigns execution descriptors to tasks. The ALG algorithm, in both the first and the second phases, selects the execution descriptor with the maximum workload reduction factor in a greedy fashion. Also, note that in the case where either of the timing and energy constraints cannot be satisfied within the code size limit, the first phase of the algorithm determines that no assignment is possible such that the task set meets those requirements. On the other hand, in the second phase, the algorithm continues the transformation of the execution descriptor assignment vector, either (1) until the system cost function is minimized, or (2) until the transformation has consumed all the remaining code space. After the algorithm has finished, V will contain the assignment of execution descriptors for all the tasks, from which the code generation decisions can be made.

The algorithm does not examine all the execution descriptors of tasks in each iteration. Instead, it checks only one execution descriptor for each task in selecting the one with the maximum workload reduction factor. Specifically, since each task's execution descriptor list has the property that the marginal gain in the ratio of reduction of the WCEC to the increase of the code size ($|\Delta c_{i,j}|/\Delta s_{i,j}$) is monotonically nonincreasing, as previously mentioned in Section 2.1, the workload reduction factors ($|\Delta w_{i,j}|/\Delta s_{i,j} = (P/p_i)|\Delta c_{i,j}|/\Delta s_{i,j}$) are also in a nonincreasing order. That is, the execution descriptors $x_{i,j+1}, x_{i,j+2}, \ldots, x_{i,K_i}$ for a task τ_i cannot have a greater workload reduction factor than the execution descriptor $x_{i,j}$ for the same task τ_i. Therefore, assuming that the current execution descriptor assignment vector is $\{v_1, v_2, \ldots, v_n\}$, the algorithm only needs to examine $X = \{x_{i,j} | i \in [1, n], j = v_i + 1\}$[4] for the purpose of selecting an execution descriptor among the remaining ones for all the tasks.

The algorithm has a substantially lower time complexity than an exhaustive search algorithm. That is, the number of iterations of the proposed algorithm is $\sum_{i=1}^{n}(K_i - 1)$ in the worst case, which is linear to the number of tasks. On the other hand, an exhaustive search would require the complexity of $\Pi_{i=1}^{n} K_i$, which is exponential to the number of tasks and thus is considered impractical.

Finally, the algorithm can be further improved by maintaining an ordered list for the candidate set X, with the workload reduction factor as the key. Having such an ordered list, selecting an execution descriptor with the maximum workload reduction factor takes constant time and adding an execution

[4]Again, if $v_i = K_i$ for a task τ_i, this candidate set does not contain any execution descriptor for that task.

Algorithm ALG: Assign an execution descriptor $x_i = \langle s_i, c_i \rangle$ for each task τ_i, $i = 1, 2, \ldots, n$.

Input: $X_i = \{x_i | i = 1, 2, \ldots, K_i\},\ i = 1, 2, \ldots, n$
Output: $V = \{v_i | i = 1, 2, \ldots, n\}$

Initialization:

$V = \langle v_1, v_2, \ldots, v_n \rangle \leftarrow \langle 1, 1, \ldots, 1 \rangle$
$X \leftarrow \{x_{1,2}, x_{2,2}, \ldots, x_{n,2}\}$

Phase 1:

```
while ((1/(f_max P) Σ_{i=1}^{n} w_i > 1.0 or κ/P^2 (Σ_{i=1}^{n} w_i)^3 > Ē) and X ≠ φ ) {
        select x_{i,j} ∈ X with maximum |Δw_{i,j}|/Δs_{i,j}
        if (Δs_{i,j} ≤ S̄ − Σ_{i=1}^{n} s_i) {
                s_i ← s_{i,j} ; c_i ← c_{i,j}
                v_i ← j
        }
        X ← X − {x_{i,j}}
        if (j < K_i)
                X ← X ⋃ {x_{i,j+1}}
}

if (X = φ)
        terminate (fail)
```

Phase 2:

```
while (X ≠ φ) {
        select x_{i,j} ∈ X with maximum |Δw_{i,j}|/Δs_{i,j}
        if (ΔE/ΔS > αĒ/βS̄) {
                if (Δs_{i,j} ≤ S̄ − Σ_{i=1}^{n} s_i) {
                        s_i ← s_{i,j} ; c_i ← c_{i,j}
                        v_i ← j
                }
                X ← X − {x_{i,j}}
                if (j < K_i)
                        X ← X ⋃ {x_{i,j+1}}
        }
        else
                X ← φ
}
```

Fig. 2. The algorithm ALG for assigning an execution descriptor to each task.

descriptor to the set requires $O(\log(|X|))$ time, which would otherwise require time complexity of $O(|X|)$ and constant time, respectively.

3.4 Reverse-Direction Version

Thus far, we have described the proposed ALG algorithm and now we present its reverse-direction version (ALG-R).[5] While the ALG algorithm works from

[5]This algorithm was suggested by an anonymous reviewer.

Table I. Benchmark Programs Used in the Experiments

Name	Source	Description	Number of exec. desc.
`crc`	SNU-RT	32-bit CRC checksum computation	7
`fir`	SNU-RT	Finite impulse response filter	10
`jfdctint`	SNU-RT	Integer discrete cosine transform for the JPEG image compression algorithm	5
`ludcmp`	SNU-RT	Solution to 10 simultaneous linear equations by the LU decomposition method	5
`matmul`	SNU-RT	multiplication of two 5×5 integer matrices	7
`adpcm.rawcaudio`	MiBench	Adaptive differential pulse code modulation speech encoding	3
`G.721.encode`	MediaBench	CCITT G.721 voice compression	7
`blowfish`	public domain	Symmetric block cipher used for data encryption	8

the smallest possible code size of each task toward the largest possible code size, the ALG-R algorithm works the other way around, i.e., from the largest possible code size of each task toward the smallest possible code size. The ALG-R algorithm works as follows:

—*Phase 1*. The first phase begins with the largest code size possible for each task by using the last execution descriptor for each task, i.e., $V = \langle K_1, K_2, \ldots, K_n \rangle$. In this phase, the algorithm tries to reach a point where all constraints are met. It selects the execution descriptor $x_{i,j}$ with the minimum value of $\Delta w_{i,j}/|\Delta s_{i,j}|$ (workload increase factor), where $\Delta w_{i,j} = w_{i,j} - w_i$ and $\Delta s_{i,j} = s_{i,j} - s_i$. We note that the ALG algorithm selects a task with the maximum workload reduction factor.

—*Phase 2*. Given that all constraints are met, we try to decrease the system cost function as much as possible, by choosing a task with the minimum workload increase factor as long as the chosen task does not increase the system cost function.

—*Assignment*. The assignment step is the same as the ALG algorithm.

4. RESULTS

To assess the effectiveness of the proposed algorithm (ALG) and its variation (ALG-R), we simulated them with benchmark programs. Section 4.1 describes the simulation setup used for the performance analysis; Section 4.2 presents the results from simulations.

4.1 Simulation Setup

For the simulations, we obtained a set of benchmark programs from three different benchmark suites: SNU-RT real-time benchmark suite [SNU], MiBench [Guthaus et al. 2001], and MediaBench [Lee et al. 1997]. One exception is the `blowfish` program, which has been obtained from the public domain sources. Table I lists the programs used in the simulations. In the table, the first column shows the name of each benchmark, while the second one denotes the source of each program. The third column gives a brief description of

each benchmark program, and the final column shows the number of elements in each benchmark program's execution descriptor list (K_i).[6] Note that the number of execution descriptors differs from one program to another, as it depends on the characteristics of the specific benchmark program.

For simulations, we have five simulation parameters as follows:

—The number of tasks (n). We set the variable n equal to $2, 3, \ldots,$ and 8.

—The tightness of the code size constraint (r_S). We set the variable r_S equal to 0.2, 0.4, ..., and 1.0 to determine the upper bound on the system code size $\bar{S}$ as follows:

$$\bar{S} = r_S \cdot (S_{max} - S_{min}) + S_{min},$$

where S_{max} and S_{min} represent the maximum and minimum possible system code sizes, respectively. In other words, a smaller value of r_S means a tighter system code size constraint, whereas a larger value means a looser constraint, with an extreme case being when $r_S = 1.0$, where the code size is not constrained at all.

—The tightness of the energy constraint (r_E). Similar to r_S, we set the variable r_E equal to 0.2, 0.4, ..., and 1.0 to determine the upper bound on the system energy consumption $\bar{E}$ as follows:

$$\bar{E} = r_E \cdot (E_{max} - E_{min}) + E_{min}$$

where E_{max} and E_{min} denote the maximum and minimum possible system energy consumption, respectively.

—The initial system utilization (r_U). We define the initial system utilization r_U as the utilization of the task set when the execution descriptor assignment vector is $V = \langle 1, 1, \ldots, 1 \rangle$ and the frequency assignment vector is $F = \langle f_{max}, f_{max}, \ldots, f_{max} \rangle$. The initial system utilization r_U is adjusted by determining the period p_i of each task τ_i by $p_i = c_{i,1} \cdot n / r_U$, while we set the variable r_U equal to 0.2, 0.4, ..., and 1.0.

—The relative importance of code size constraint and energy consumption constraint in the system cost function (α and β). To vary the relative importance of the system code size constraint and the system energy consumption constraint, we vary the relative importance of the two optimization criteria by letting $\alpha = 0.0, 0.2, \ldots,$ and 1.0 and $\beta = 1.0 - \alpha$.

For each value of the number of tasks, $n = 2, 3, \ldots,$ and 8, we generated 30 different task sets in a random manner from a set of benchmark programs.[7]

[6]Specifically, the number of execution descriptors for a program is derived by using a selective code transformation technique described in Lee et al. [2004]. The technique begins with the smallest code size possible for a given program, and then gradually increases the code size while satisfying the constraint given on the upper bound on the total code size. In return for the increased code size, the WCET of the program is reduced. The technique proposed in Lee et al. [2004] generates code for the program so that the WCET is minimized within the code size constraint. By applying the technique, while adjusting the code size constraint, we can derive the possible set of execution descriptors for the given program.

[7]To generate 30 different task sets, we added a modified `crc` program when $n = 2$, 6, and 7, and modified `crc` and `matmul` programs when $n = 8$.

Table II. Impact of the Number of Tasks on the Solution Quality

Number of Tasks	Closeness				Solution Percentage (%)			
	Mean (std. dev.)		Worst-case		Optimal		Feasible	
	ALG	ALG-R	ALG	ALG-R	ALG	ALG-R	ALG	ALG-R
2	1.000 (0.004)	1.000 (0.003)	1.236	1.185	90.9	88.0	95.8	95.7
3	1.001 (0.005)	1.001 (0.003)	1.156	1.122	88.1	84.2	98.2	97.9
4	1.001 (0.004)	1.001 (0.003)	1.089	1.057	87.8	83.0	99.5	99.1
5	1.001 (0.003)	1.000 (0.002)	1.056	1.032	87.4	81.1	99.7	99.5
6	1.000 (0.002)	1.000 (0.001)	1.029	1.015	87.4	81.9	99.9	99.9
7	1.000 (0.001)	1.000 (0.001)	1.017	1.011	87.3	83.0	100.0	100.0
8	1.000 (0.001)	1.000 (0.001)	1.014	1.009	86.8	81.5	100.0	100.0

For each task set, the other four simulation parameters, combined together, give us a total of 750 different simulation cases. That is, since r_S, r_E, and r_U have five different values, respectively, and the combination of α and β has six distinct values, the combination of all these values results in $5 \times 5 \times 5 \times 6 = 750$ distinct cases.

Three performance metrics are used to measure the effectiveness of our proposed algorithm. One is the *feasible solution percentage*, which is the percentage of simulation cases, where an algorithm finds a feasible solution. Another is the *optimal solution percentage*, which is the percentage of simulation cases, where an algorithm finds an optimal solution. Here, the optimal solution is defined to be an assignment that gives the minimum value of the objective function, which is found by an exhaustive search over all the possible combinations of the assignment. The final metric is the *closeness*, which represents the closeness of the solution of an algorithm to an optimal solution. The closeness of a solution z to the optimal solution z^* is defined as follows:

$$\text{closeness}(z) = \frac{z}{z^*} \tag{13}$$

We computed the mean of closeness of an algorithm's solutions only for its feasible solutions.

We simulated two heuristic algorithms over all the simulation cases for each number of tasks and also performed an exhaustive search to find an optimal solution.

4.2 Simulation Results

Table II presents the results to show the impact of the number of tasks (n) on the effectiveness of the two algorithms, which are our proposed algorithm (ALG) and its reverse-direction version (ALG-R). It is shown that these two algorithms generate near-optimal solutions, with the average closeness no greater than 1.001, while ALG-R performs slightly better than ALG in terms of generating the worst-case closeness. The worst-case closeness of ALG ranges from 1.014 to 1.236 and that of ALG-R ranges from 1.009 to 1.185. Two algorithms perform similarly in finding feasible solutions, but ALG performs slightly better than ALG-R in finding optimal solutions. The optimal solution percentage of ALG ranges from 86 to 91%, while that of ALG-R ranges from 81 to 88%.

Table III. Impact of the Tightness of the Code Size Constraint on the Solution Quality

	Closeness				Solution percentage (%)			
	Mean (std. dev.)		Worst-case		Optimal		Feasible	
r_S	ALG	ALG-R	ALG	ALG-R	ALG	ALG-R	ALG	ALG-R
0.20	1.000 (0.002)	1.001 (0.002)	1.022	1.016	83.7	73.8	98.7	97.3
0.40	1.001 (0.003)	1.001 (0.002)	1.056	1.032	85.2	77.5	100.0	100.0
0.60	1.001 (0.003)	1.000 (0.002)	1.056	1.032	86.7	80.1	100.0	100.0
0.80	1.001 (0.003)	1.000 (0.002)	1.056	1.032	89.1	81.1	100.0	100.0
1.00	1.001 (0.003)	1.000 (0.002)	1.056	1.032	91.5	93.1	100.0	100.0

As the number of tasks increases, the algorithms generate a worst-case solution closer to the optimal one. With a large number of tasks, the algorithms have a number of different possible selections for the execution descriptor assignment. Therefore, even if the algorithms make suboptimal decisions, the solutions are more likely to be close to the optimal one, since the algorithm have a number of choices of tasks whose code size is increased in return for reduced workload.

On the other hand, the optimal solution percentage of the two algorithms decreases as the number of tasks increases. This can be explained as follows. When there are a large number of tasks, the solution space is large, i.e., there exist a number of different feasible solutions. Therefore, a number of execution descriptor assignments are possible that are different from the optimal one, but yet very close to it, which leads to the situations that both of the two algorithms generate solutions different from the optimal one. However, even in such cases, the magnitude of the closeness for the algorithms is negligibly small, which indicates that the algorithms can always derive a near-optimal solution.

To assess the impact of the other parameters, we summarize the simulation results in a number of different ways. In doing this, we fixed the number of tasks to $n = 5$, while all the parameters other than the one being considered are varied in the manner explained in Section 4.1. The simulation results are averaged over all the combinations of the varying parameters.

First, we analyzed the impact of the tightness of the system code size constraint on the solution quality by varying the variable r_S. The results are shown in Table III. When the constraint on the system code size is tight, i.e., when the value of r_S is small, the algorithms should efficiently reduce the workload of the task set by distributing the small additional code space to appropriate tasks. On the other hand, when the constraint is loose, i.e., when the value of r_S is large, the algorithms are allowed to more freely select a task whose code size is to be increased for reduced workload. Therefore, the optimal solution percentage of the proposed algorithm becomes higher as the value of r_S increases.

Table IV shows the simulation results according to the variation in the tightness of the system energy consumption constraint. The average closeness of the algorithms is shown to be immune to the tightness of the energy consumption constraint. On the other hand, as the value of r_E increases, the worst-case closeness of the algorithms gets smaller and their optimal solution percentage becomes higher.

Table III and IV also show that ALG generally performs better than ALG-R in terms of finding optimal solutions, except when there is no code size

Table IV. Impact of the Tightness of the Energy Constraint on the Solution Quality

	Closeness				Solution percentage (%)			
	Mean (Std. Dev.)		Worst-Case		Optimal		Feasible	
r_E	ALG	ALG-R	ALG	ALG-R	ALG	ALG-R	ALG	ALG-R
0.2	1.002 (0.005)	1.001 (0.004)	1.056	1.032	75.3	73.2	99.0	97.7
0.4	1.001 (0.002)	1.000 (0.001)	1.023	1.013	86.1	79.5	99.7	99.7
0.6	1.001 (0.003)	1.000 (0.002)	1.021	1.016	89.9	83.1	100.0	100.0
0.8	1.000 (0.001)	1.000 (0.001)	1.010	1.012	95.1	87.7	100.0	100.0
1.0	1.000 (0.001)	1.000 (0.001)	1.010	1.012	89.8	82.2	100.0	100.0

Table V. Impact of the System Utilization on the Solution Quality

	Closeness				Solution Percentage (%)			
	Mean (std. dev.)		Worst-case		Optimal		Feasible	
r_U	ALG	ALG-R	ALG	ALG-R	ALG	ALG-R	ALG	ALG-R
0.2	1.001 (0.003)	1.000 (0.002)	1.056	1.032	87.3	81.2	99.7	99.5
0.4	1.001 (0.003)	1.000 (0.002)	1.056	1.032	87.2	81.1	99.7	99.5
0.6	1.001 (0.003)	1.000 (0.002)	1.056	1.032	87.3	81.1	99.7	99.5
0.8	1.001 (0.003)	1.000 (0.002)	1.056	1.032	87.2	81.1	99.7	99.5
1.0	1.001 (0.003)	1.000 (0.002)	1.056	1.032	87.2	81.1	99.7	99.5

Table VI. Impact of the Relative Importance of the System Code Size and the System Energy Consumption on the Solution Quality

	Closeness				Solution Percentage (%)			
	Mean (Std. Dev.)		Worst-Case		Optimal		Feasible	
α	ALG	ALG-R	ALG	ALG-R	ALG	ALG-R	ALG	ALG-R
0.0	1.000 (0.001)	1.001 (0.002)	1.007	1.012	80.5	37.5	99.7	99.5
0.2	1.000 (0.000)	1.000 (0.001)	1.005	1.006	97.7	94.6	99.7	99.5
0.4	1.000 (0.001)	1.000 (0.001)	1.015	1.011	97.3	98.1	99.7	99.5
0.6	1.000 (0.002)	1.000 (0.001)	1.028	1.018	92.7	95.2	99.7	99.5
0.8	1.001 (0.003)	1.000 (0.002)	1.042	1.025	83.6	87.0	99.7	99.5
1.0	1.002 (0.006)	1.001 (0.004)	1.056	1.032	71.6	74.5	99.7	99.5

constraint, i.e., when $r_S = 1.0$. With no code size constraint, but energy and time constraints, ALG-R can perform better than ALG, since ALG starts from the largest energy consumption (smallest code size) of each task while ALG-R starts from the smallest energy consumption (largest code size).

Table V summarizes the results according to the varying initial system utilization. The system utilization constrains the optimization problem in the form of tightness of timing constraint for each task. Since the EDF scheduling algorithm is able to meet the deadlines of all the tasks provided that the system utilization (under the maximum frequency settings) is under 1.0, the system utilization has a direct relationship with the schedulability of the task set and thus the timing constraints. The results do not differ significantly, depending on the initial system utilization. That is, the timing constraint does not have a great influence on the solution quality, since it does not affect the procedure of minimizing the objective function once all the tasks are guaranteed to meet their respective deadlines.

Finally, Table VI shows the results according to the relative importance of the system code size and the system energy consumption.When $\alpha = 0.0$ and $\beta = 1.0 - \alpha = 1.0$, the objective function consists only of the system energy

consumption, where the optimization goal is to minimize the system energy consumption. On the other hand, when $\alpha = 1.0$ and $\beta = 0.0$, the optimization goal is to solely minimize the system code size, indicated by the objective function simply denoting the system code size. As shown in the table, the relative importance given by the system developer does not have a great impact on the solution quality, in terms of both the worst-case closeness and the optimal solution percentage.

Two notable cases from the others are when $\alpha = 1.0$ and $\alpha = 0.0$, where we have a smaller optimal solution percentage for both algorithms. These correspond to the cases where the optimization goal is merely the system code size or the system energy consumption only. Since the algorithms use the workload reduction (increase) factor as the criteria in selecting the task whose code size is increased (reduced) for a reduced (increased) workload in order to reduce a combination of the system code size and energy consumption, such performance degradation is possible when the objective function captures only the system code size or the system energy consumption. That is, in selecting a task and increasing its code size, the algorithms give priority to the one with the maximum workload reduction factor (minimum workload increase factor), instead of the one with the smallest increase in code size or in energy consumption. Even in such cases, both algorithm's average closeness was no greater than 1.002.

5. DISCUSSION

Thus far, we have addressed the proposed optimization problem under the assumptions that (1) the operating frequency of the processor can be continuously scaled and (2) the execution descriptor list of each task has the property of monotonically nonincreasing workload reduction factors. This section discusses the issues of extending our design framework in relaxing these two assumptions.

5.1 Discrete-Level CPU Frequency Settings

In Section 2.1, we assumed that the clock frequency of the processor can be set at a continuous level. However, using continuously variable clock frequencies is infeasible, because it requires significant power and hardware cost [Ishihara and Yasuura 1998]. In other words, real processors usually have a finite number of operating frequencies and clock frequencies cannot be continuously scaled. To sustain acceptable performance and timeliness guarantee, these processors have to operate at the next higher energy-efficient operating frequency(f') if a desired frequency is not available [Saewong and Rajkumar 2003]. That is, where the processor has Q levels of clock frequency (i.e., $f' \in \{F_1, F_2, \ldots, F_Q\}$ and $f_{max} = F_Q$), the next higher clock frequency(f') can be determined as

$$f' = MIN\ \{\ F_i \mid U(F_Q) \cdot F_Q \leq F_i\ ,\ 1 \leq i \leq Q\} \tag{14}$$

Although the schedule of the task set is feasible under f', it may not result optimal energy efficiency. When the resulting processor utilization is less than 1.0 (i.e., $U(f') < 1.0$), the processor is not fully utilized and there are idle times in the task schedule. In this case, some tasks may have a chance to lower

their clock frequencies one more step, so that tasks may have different clock frequencies in order to improve the energy efficiency.

For this problem, several DVS algorithms were proposed in [Ishihara and Yasuura 1998; Kwon and Kim 2003]. However, these algorithms cannot be directly used for the periodic task model. In this paper, we formulate this problem as a well known *0/1 Knapsack problem*, and it can be solved using a traditional search algorithm.

Where the clock frequency for τ_i is

$$f_i = \{\ F_{q_i} \mid F_{q_i} \in \{F_1, F_2, \ldots, F_Q\}\ ,\ 1 \le q_i \le Q\}$$

the set of current clock frequencies for tasks is

$$F\ =\ \{f_1, \ldots, f_n\}\ = \{F_{q_1}, \ldots, F_{q_n}\}$$

Our goal is to find a set of tasks which can be scheduled with the next lower clock frequency(F_{q_i-1}) while the feasible schedule of tasks is still guaranteed ($U(F') \le 1$, where $U(F')$ is the processor utilization when tasks are scheduled with F'). Let the solution be $L = \{l_i | l_i \in [0, 1], 1 \le i \le n\}$. If τ_i can be scheduled with a next lower clock frequency, then l_i is set to 1; otherwise to 0. Then the problem can be formulated as follows.

—Find $L = \{l_1, \ldots, l_n\}$

- (Objective) which maximizes

$$U' = \sum_{i=1}^{n} \frac{t_i}{p_i} = \sum_{i=1}^{n} \frac{c_i}{p_i \cdot F_{q_i - l_i}}$$

- (Constraints) while satisfying

$$U' \le 1 \ \text{ and } \ \forall i \in [1, n]\ ,\ q_i - l_i > 0 \ \text{ where } \ l_i \in \{0, 1\}$$

- (Assignment) When the solution(L) is found, the clock frequency for each task can be adjusted as

$$\forall i \in [1, n]\ ,\ q_i = q_i - l_i \ \text{ and } \ f_i = F_{q_i}.$$

Even if the tasks are scheduled with these lowered clock frequencies, the task set may not fully utilize the processor (i.e., $U(F') < 1$). In this case, we may iterate the above procedure until no task can be scheduled with a further lowered clock frequency.

5.2 Nonconvex Property of Execution Descriptor List

So far, we have assumed that the execution descriptor list of each task has the property of monotonically nonincreasing workload reduction factors. The proposed algorithm presented in Section 3.3 needs to examine only one execution descriptor for each task because of this property. That is, the candidate set $X = \{x_{i,j} | i \in [1, n], j = v_i + 1\}$ only contains, at most, one execution descriptor for a task, which is immediately following the one x_{i,v_i} that is currently assigned during the iteration of the proposed algorithm. However, when we relax this assumption on the execution descriptor list property, the algorithm should explore an extended candidate set $X^* = \{x_{i,j} | i \in [1, n], j \in [v_i + 1, K_i]\}$ that lists

all the execution descriptors for a task that have larger code sizes than the one currently assigned for that task (i.e., x_{i,v_i}). That is, by substituting X with X^* in the proposed algorithm, we can solve the same problem of finding the execution descriptor assignment vector, without assuming any property for the execution descriptor list for each task.

This modified algorithm still has a significantly lower time complexity than an exhaustive search algorithm. If we let K denote the sum of number of execution descriptors for all the tasks, i.e., $K = \sum_{i=1}^{n} K_i$, then the worst-case number of evaluations of the execution descriptors is given by $\sum_{i=1}^{K}(K-i) = (K^2-K)/2$. Since K is proportional to the number of tasks n, this indicates that the time complexity of our modified algorithm is $O(n^2)$, while that of the algorithm presented in Section 3.3 is linear to the number of tasks. Note that this complexity is still polynomial to the number of tasks, while that of the exhaustive search is $\Pi_{i=1}^{n} K_i$, which is exponential to the number of tasks.

6. RELATED WORK

In general, a program's code size and its execution time have a tradeoff relationship with each other. When main memory was a scarce and costly resource, much effort was made to keep the code size as small as possible. Microprogramming approaches and CISC (complex instruction set computing) architectures are typical examples of techniques for code size reduction. However, with the increase of memory capacity, the emphasis has been shifted onto enhancing the execution time. Along with the evolution of RISC (reduced instruction set computing) and VLIW (very long instruction word) architectures, a number of aggressive compiler techniques have been developed aiming at improving a program's execution speed in exchange for an increase in code size. Examples include loop unrolling, software pipelining, and procedure inlining [Muchnick 1997].

Recently, especially in the embedded domain, software techniques for code size reduction have been widely developed, since the cost of main memory is often critical in these systems [Sutter and Furhere 2003]. Techniques based on code compression [Cooper and McIntosh 1999] and procedural abstraction [Fraser et al. 1984] attempt to reduce the amount of memory occupied by program code, despite the fact that they may degrade the execution speed. Another approach for code size reduction is by using mixed-width instruction sets [Krishnaswamy and Gupta 2002], where the processor supports a reduced bit-width in addition to a normal instruction set.

While the above-mentioned techniques generally aim at addressing the code size constraint often imposed on embedded systems, challenges in compiler techniques for embedded real-time systems also lie in handling the WCEC instead of average-case performance. For example, Zhao et al. [2004] proposed a method to tune the worst-case performance of an application program via an interactive compiler optimization framework. The approach aims at finding a program transformation sequence that yields good worst-case performance, by allowing the user to gauge the progress of reducing the WCEC during the compilation process. In addition, they proposed a technique to automatically search for an effective optimization sequence using a genetic algorithm.

In addition to the worst-case performance, an important issue in embedded system design is the energy consumption. One of the most acknowledged techniques for reducing the energy consumed by the system is the DVS (dynamic voltage scaling). This technique adjusts the processor's operating clock frequency and supply voltage at runtime, in order to reduce the energy consumption of the processor while providing adequate execution performance. Especially in real-time systems, much effort has been made to develop a method to derive frequency and voltage settings for tasks in the system so that the overall energy consumption is minimized while the tasks meet the timing requirements. The DVS techniques for real-time systems are to effectively control the tradeoff relationship between the execution time and energy consumption of real-time tasks.

A seminal work in the area of DVS algorithms for hard real-time systems has been reported by Yao et al. [1995]. The authors proposed an optimal voltage setting algorithm for aperiodic tasks under the EDF scheduling algorithm. The algorithm estimates the workload density for each execution interval based on tasks' arrival times, deadlines, and execution cycles, according to which the frequency/voltage pair is assigned to each task instance. Quan and Hu [2001] indicate that this algorithm is not directly applicable to other scheduling algorithms because the schedulability of a task set is largely dependent on the underlying scheduling policy. Based on this observation, they proposed an extension to Yao et al.'s algorithm that can be applied to aperiodic task sets scheduled by a fixed-priority scheduling scheme, which is optimal in terms of energy consumption.

For a periodic task set, Shin et al. [2000] proposed a power optimization method for computing the lowest possible processor speed that guarantees the schedulability of the task set under EDF and RM scheduling. Pillai and Shin [2001] described heuristics for computing the processor speed dynamically to deal with situations where a task uses less than its WCEC. They also presented the implementation of their heuristics for EDF scheduling, which has been known as the first implementation that supports DVS for real-time systems. Gruian [2001] developed a stochastic DVS approach based on execution time distributions of tasks. This algorithm assigns several different voltage levels to each task based on the given distribution function so that a task starts execution at a low speed and later accelerates the execution. The author also proved that his stochastic DVS algorithm is probabilistically optimal. While most studies have focused on the problem of minimizing the energy consumption of the processor subject to the schedulability of a task set, Rusu et al. [2002, 2003] proposed a framework for the problem of maximizing the system value (reward) subject to timing and energy constraints, where the system value is the sum of task values for all tasks and the value of a task is determined by the task's CPU frequency.

In our earlier work [Shin et al. 2004], we have developed a design framework for addressing optimization problems for embedded real-time systems with timing, code size, and energy constraints. We first introduced a triple-tradeoff relationship among code size, execution time, and energy consumption. We then formulated an optimization problem that is to minimize the system

code size subject to timing and energy constraints. In this paper, we extend this work (1) by considering another optimization problem, which is to minimize the weighted combination of code size and energy, (2) by showing the problem as NP-hard and presenting an algorithm to it, and (3) by evaluating our algorithm using benchmark programs through simulations.

7. CONCLUSIONS AND FUTURE WORK

We have proposed a design framework for real-time embedded systems that balances the tradeoff relationship involving code size, execution time, and energy consumption. The tradeoff between code size and execution time is enabled by using a code-generation technique that can provide different versions of code for a given program that have distinguished characteristics in terms of code size and execution time. On the other hand, the tradeoff between execution time and energy consumption is enabled by using a variable voltage processor, which provides a mechanism to reduce the energy consumption by lowering the processor's supply voltage while the execution speed is degraded accordingly.

The proposed design framework formulates the multidirectional optimization as a single constrained optimization problem. The constraints are given as upper bounds on the system's total code size and the energy consumed by execution of the tasks, along with the task set's timing requirements. Within these resource constraints, the proposed technique optimizes the system by minimizing the objective function, which captures the system cost in terms of code size and energy consumption and their relative importance. The technique derives a set of design parameters as a result: (1) the code generation parameters used by the code generator to select a specific version of code for each task and (2) the voltage setting parameters used by the operating system to adjust the processor's supply voltage and the operating clock frequency. The design parameters are derived in such a way that the objective function is minimized while they collectively satisfy the given resource constraints.

We showed that the design goals seemingly conflicting with one another can be pursued by using a single selection criteria and provided an algorithm to derive the design parameters. The algorithm begins with the smallest code size possible for all the tasks and iteratively selects a task to increase the code size in return for reduced workload. By reducing the total workload generated by the task set as much as possible within the code size limit, the algorithm first tries to satisfy the timing and energy constraints, and then seeks to minimize the objective function. The performance of the proposed algorithm and its variation has been evaluated using a set of simulations, where we compared the solutions provided by the proposed algorithm with the optimal ones found by an exhaustive search. The results show that our proposed algorithm provides near-optimal solutions, with the average error of approximately 0.6%.

The future direction of our research will focus on the following. First, the applicability of the proposed framework can be significantly enhanced by incorporating more accurate timing analysis techniques that take the variation of the number of cycles that a program executes according to DVS settings. For

example, the memory access cycle can vary according to the clock frequency. That is, lowering the clock frequency may reduce the number of cycles required for a memory access, since the processor cycle time is stretched while the memory cycle time remains the same. Therefore, a task's WCEC, which has been assumed to be known at compile time, should be estimated while the potential variation in memory access cycles are taken into account. One novel example of such timing analysis technique has been proposed by Seth et al. [2003] with frequency-sensitive parameters.

Another direction for future research is to incorporate on-line optimization techniques that operate while the system is running. For example, we can further optimize the system by using on-line DVS algorithms and/or adaptive scheduling algorithms. For example, the energy consumed by the processor can be efficiently reduced by exploiting slack time. When a task is executed, it is highly likely to finish before its estimated WCET for a number of reasons. First, the WCET bound for a task used in schedulability analysis is not the actual WCET, but a safe upper bound to its execution time. Second, regardless of how accurate and tight the WCET bound may be, tasks can be completed before their estimated WCET, since execution times depend upon the input data and the actual execution paths taken according to them. Efforts have been made to develop efficient and effective on-line DVS algorithms [Kim et al. 2002, 2003]. Using such techniques, by accurately estimating the slack times and putting them into use for lowering other tasks' voltage and frequency settings, the energy consumption of the system can be substantially reduced. In addition, exploring runtime variations by incorporating some statistical information for energy optimization is another interesting open problem.

In this paper, we proposed an optimization algorithm (ALG) assuming that the task set is scheduled by the EDF scheduling algorithm. However, a different scheduling algorithm, e.g., a fixed-priority scheduling policy, such as the RM (Rate Monotonic) scheduling, can be used to schedule the tasks. In such a case, a simple schedulability analysis based on utilization bound cannot be directly used. In addition, it is no longer guaranteed that the energy consumption can be minimized by assigning the same frequency and the same supply voltage to all the tasks or even to all the instances of the same task [Shin et al. 2000; Saewong and Rajkumar 2003]. Therefore, it is necessary to develop a different optimization algorithm based on an appropriate schedulability anlaysis for the RM scheduling algorithm. We plan to extend our framework in this direction.

ACKNOWLEDGMENTS

We thank the reviewers for the insightful and constructive comments and suggestions that are essential to improve this paper.

REFERENCES

Chandrakasan, A. P., Sheng, S., and Brodersen, R. W. 1992. Low-power digital CMOS design. *IEEE Journal of Solid State Circuits 27*, 4 (Apr.), 473–484.

Chang, N., Kim, K., and Lee, H. G. 2002. Cycle-accurate energy measurement and characterization with a case study of the ARM7TDMI. *IEEE Transactions on VLSI Systems 10*, 146–154.

COOPER, K. D. AND MCINTOSH, N. 1999. Enhanced code compression for embedded RISC processors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Atlanta, GA. 139–149.

FRASER, C. W., MYERS, E. U., AND WENDT, A. L. 1984. Analyzing and compressing assembly code. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*. Montreal, Canada. 117–121.

FURBER, S. 1996. *ARM System Architecture*. Addison-Wesley. Reading, PA.

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and Interactability: A Guide to Theory of NP-Completeness*. Freeman, San Francisco, CA.

GRUIAN, F. 2001. Hard real-time scheduling using stochastic data and DVS processors. In *Proceedings of the Internatinoal Symposium on Low-Power Electronics and Design (ISLPED)*, Huntington Beach, CA. 46–51.

GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th IEEE Annual Workshop on Workload Characterization*. Austin, TX.

ISHIHARA, T. AND YASUURA, H. 1998. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 197–202.

KIM, W., KIM, J., AND MIN, S. L. 2002. A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis. In *Proceedings of the Design, Automation and Test in Europe (DATE)*, Paris, France, 788–794.

KIM, W., KIM, J., AND MIN, S. L. 2003. Dynamic voltage scaling algorithm for fixed-priority real-time systems using work-demand analysis. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, Seoul, Korea, 396–401.

KRISHNASWAMY, A. AND GUPTA, R. 2002. Profile guided selection of ARM and Thumb instructions. In *Proceedings of LCTES/SCOPES*, Berlin, Germany.

KWON, W. AND KIM, T. 2003. Optimal voltage allocation techniques for dynamically variable voltage processors. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*. 125–130.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*. 330–335.

LEE, S., LEE, J., MIN, S. L., HISER, J., AND DAVIDSON, J. W. 2003. Code generation for a dual instruction set processor based on selective code transformation. In *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, Vienna, Austria, 33–48.

LEE, S., LEE, J., PARK, C. Y., AND MIN, S. L. 2004. A flexible tradeoff between code size and WCET using a dual instruction set processor. In *Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. Amsterdam, The Netherlands. 244–258.

LIU, C. L. AND LAYLAND, J. W. 1973. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM 20*, 1 (Jan.), 46–61.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA.

PILLAI, P. AND SHIN, K. G. 2001. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of 18th ACM Symposium on Operating Systems Principles (SOSP'01)*.

QUAN, G. AND HU, X. S. 2001. Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *Proceedings of the Design Automation Conference (DAC)*, Las Vegas, NV. 828–833.

RUSU, C., MELHEM, R., AND MOSSE, D. 2002. Maximizing the system value while satisfying time and energy constraints. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, Austin, TX. 246–255.

RUSU, C., MELHEM, R., AND MOSSE, D. 2003. Multi-version scheduling in rechargeable energy-aware real-time systems. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS)*, Porto, Portugal. 95–104.

SAEWONG, S. AND RAJKUMAR, R. 2003. Pracitcal voltage-scaling for fixed-priority RT-systems. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Toronto, Canada. 106–115.

SETH, K., ANANTARAMAN, A., MUELLER, F., AND ROTENBERG, E. 2003. FAST: Frequency-aware static timing analysis. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS)*, Cancun, Mexico. 40–51.

SHIN, I., LEE, I., AND 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)*, Gothenburg, Sweden.

SHIN, Y., CHOI, K., AND SAKURAI, T. 2000. Power optimization of real-time embedded systems on variable speed processors. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. 365–368.

SNU Real-Time Benchmark Suite. http://archi.snu.ac.kr/realtime/benchmark.

SUTTER, B. D. AND FURHERE, K. D. 2003. Software techniques for program compaction. *Communications of the ACM 46*, 8 (Aug.), 33–34.

YAO, F., DEMERS, A., AND SHENKER, A. 1995. A scheduling model for reduced CPU energy. In *Proceedings of the IEEE Foundations of Computer Science*. 374–382.

ZHAO, W., KULKARNI, P., WHALLEY, D., HEALY, C., MUELLER, F., AND UH, G.-R. 2004. Tuning the WCET of embedded applications. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Toronto, Canada. 472–481.

Received May 2005; revised March/September 2006; accepted October 2006