Task Construction for Model-Based Design of Embedded Control Software

Shige Wang, *Member*, *IEEE*, and Kang G. Shin, *Fellow*, *IEEE*

Abstract—Constructing runtime tasks, or operating system-level processes/threads, from the components of software design models is crucial to the model-based development of embedded control software. A better method should explore more design choices and reduce the overheads of the runtime system to meet the timing and resource constraints of embedded control software. This paper presents a novel, two-step method for systematic and automatic construction of runtime tasks from software design models. It uses graph transformation to construct a task set meeting system-level end-to-end (e2e) timing constraints. Its first step decomposes the system-level e2e timing constraints into the components' timing constraints, which form a necessary condition for any valid and feasible schedule. The second step iteratively merges the components into tasks and sequences their executions. A thus-constructed task set is proven to meet both intercomponent precedence and system-level e2e timing constraints and to minimize runtime overheads by minimizing the total number of resultant tasks. Our evaluation results based on randomly generated software models have shown that the proposed method outperforms commonly used methods and is also scalable.

Index Terms—Task construction, model transformation, model-based design, embedded software.

1 Introduction

Software for today's large embedded control systems is very complex, consisting of many intercommunicating components for various devices and control functions. These embedded control systems are usually mission and safety-critical. The control software for these systems must then meet stringent timing constraints imposed by the target applications. On the other hand, the platform—consisting of supporting software, such as operating system (OS) and middleware, and hardware for the execution of application software—is typically resource-limited. Finding feasible solutions to the problem of running such complex software with stringent timing constraints on a resource-limited platform poses a serious challenge to the designers of embedded control software.

Model-based software design, which focuses on architecture-level optimization instead of traditional code-level optimization, has been shown to be very promising [4], [10], [21]. In model-based software design, an abstract software model is first synthesized using software components to implement the designed control functions. Since the software must be organized as tasks in order to be implemented as threads/processes for scheduling and execution on a platform, the component-based software model must be transformed to a task-based model while considering all constraints as the design process progresses. This transformation step, called *task construction* in our e2e software design process, is essential for software implementation, as well as for offline analyses to ensure the system correctness. It is highly desirable and necessary to automate such a task-construction process in order to support automation of the e2e software design process and to explore better solutions quickly and effectively in the large design space.

This paper presents a novel method for automatic task construction, which can be integrated into an existing/emerging e2e software design tool chain. The method starts with a discretized control represented in a component-based software model with the precedence and e2e timing constraints specified. The components in the model are assumed to be allocated to the devices on a platform that supports multitask scheduling. For such a model, tasks can be constructed in two steps. The first step derives the timing constraints of each component, including its invocation rate, earliest-start-time (EST), and latest-completion-time (LCT), from the precedence and e2e timing constraints. The resulting components' timing constraints form a necessary condition for any valid and feasible schedule of the software and, thus, can be used to verify each choice and prune those choices leading to infeasible solutions at each step. The second step iteratively groups the components to form tasks according to *rate similarity* and *execution overlap* to reduce the number of constructed tasks, which, in turn, lowers the storage and runtime overheads of task management, without complicating task scheduling and runtime management in OS. During the task construction, the components in each task are sequenced for execution using the minimum-EST heuristic.

The task construction is, in general, a difficult problem because it may require transformation of a model in one model-of-computation (MoC) to a model in another while preserving the properties of the original model. Such a

- *S. Wang is with the Electrical and Control Integration Lab, General Motors R&D and Planning, General Motors Corp., 30500 Mound Road, Warren, MI 48090. E-mail: shige.wang@gm.com.*
- *K.G. Shin is with the Department of Electrical Engineering and Computer Science, The University of Michigan, 2260 Hayward Street, Ann Arbor, MI 48109. E-mail: kgshin@eecs.umich.edu.*

Manuscript received 6 Sept. 2005; revised 25 Jan. 2006; accepted 22 Feb. 2006; published online 27 Apr. 2006.
Recommended for acceptance by A. Wellings.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0239-0905.

model transformation between different MoCs is not always possible, nor easy. In this paper, we consider only the case where the MoC supported by the platform always comes with richer semantics than the MoC used for software design. As the platform MoC comes with richer semantics, there may exist multiple models in the platform MoC with the same properties as the original software design model after making a transformation. Although our approach is based on transformation of the models in two specific MoCs, similar algorithms can be developed to implement a model transformation between any two MoCs that are semantically transformable.[1] As it is based on graph transformation, our proposed approach can fully automate the task-construction process while still allowing the designer to interact with the e2e design process iteratively to make architecture-level trade-offs and provide criteria to evaluate the system optimality under complex, and sometimes conflicting, design constraints.

The rest of the paper is organized as follows: Section 2 states the system models and the task-construction problem. Section 3 details our task-construction method and algorithms. Section 4 presents the experimental results using randomly generated software models. Section 5 discusses related work. The paper concludes with Section 6.

2 System Model and Problem Formulation

The task construction is to generate individual schedulable OS processes/threads from a software architecture that implements the discretized control functions. Any task construction method should then be based on models for the software architecture and tasks.

2.1 Software Architecture Model

The software architecture in our task construction is modeled as a set of concurrent *transactions*, each of which describes software components and their interactions in an e2e information processing flow.

Definition 1. *A* transaction *is defined as a weighted directed acyclic graph, $G_T = (C, L, loc, F, H)$, where*

- *C is a set of port-based software components;*
- *L is a set of directed links representing synchronous data communications, each of which connects an output port O_i of a component c_i to an input port I_j of another component c_j (with $\{O_i\} \cap \{I_j\} = \emptyset$ for all components);*
- *$loc : C \rightarrow N^+$ defines a function that uniquely maps a component to an integer representing a computation device on the target platform;*
- *$F : C \bigcup L \rightarrow Q_0^+$ defines a weight function that maps a component/link to a nonnegative rational number in Q_0^+, representing the resource demand;*
- *H defines a set of system-level timing constraints of a transaction, including invocation rates, release offsets for each input, and a deadline for each output.*

1. An MoC A is said to be "semantically transformable" to another MoC B if any modeling construct of A can be expressed using some (basic or composite) modeling construct of B with the same semantics.

In Definition 1, the components in C are considered as basic building blocks, each of which is modeled as a reactive port automaton [22]. Upon invocation, a component executes a set of predefined functions in a run-to-completion manner, transforms the inputs to outputs, and delivers the result(s) to all of its output ports upon completion of its execution. This indicates that the components in our model are *process-oriented* and not *object-oriented*. For a transaction containing components whose structures are modeled in an object-oriented modeling language, such as UML [18], we can transform the model to one with only process-oriented components by tracing the interaction models (e.g., collaboration diagrams and/or sequence charts in UML) in the object-oriented model and generating *port-dependency graphs* (PDGs), as described in [10]. The transformation requires that the potential blocking caused by the mutually exclusive access of an object's internal data between its method calls be uncovered and then included in the PDG as a part of the component's resource demand.

In this work, we start with discretized control, so no cycle exists in a transaction model. A component without any incoming link, called an *input component*, models the beginning of the transaction. An input component is triggered by an external signal, such as a timer or a data-arrival event. The designer is responsible for discretizing the control with a proper transaction rate to maintain the freshness of data. Similarly, a component without any outgoing link, called an *output component*, models the completion of concurrent computations of the transaction. Synchronous, directed links specify the precedence constraints among the components, indicating a firing token sent to a downstream component. A noninput component starts only when all its inputs are available. This implies that the MoC for components' concurrency in a transaction follows synchronous data flow (SDF) [9]. An invocation of a transaction, therefore, starts upon release of its earliest input component and finishes upon completion of its last output component.

Here, we also assume that the task construction starts with a model whose components have already been allocated to the computation devices using, for example, the techniques in [14], [27]. These techniques consider only resource consumptions. For every component c, $loc(c)$ defines the device that will execute c. The resource demand $F(c)$ of a component c, in terms of uninterrupted computation time, can then be determined based on the characteristics of the device where c is located. Similarly, the resource demand $F(l)$ needed for communications of components on different devices can also be determined according to the network characteristics. The communication delays between components on the same device are assumed to be negligible ($F(l) = 0$).

Each transaction T_i is specified with a set of system-level e2e timing constraints obtained from control discretization, including invocation rate $r_i(c_{in})$ and release offset $o_i(c_{in})$ for each input component c_{in}, and relative deadline $D_i(c_{out})$ for each output component c_{out}. The invocation rate, $r_i(c_{in})$, models the frequency at which an input is updated. Under the synchronous transaction model, all the input components, and, consequently, all other components, of the

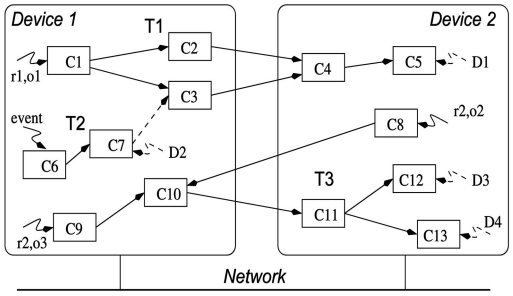

Fig. 1. An example software model with three transactions.

transaction must run at the same rate. Such a rate r_i is called the *invocation rate* of the transaction T_i and satisfies the relationship $r_i = r_i(c_{in})$. For a transaction whose inputs arrive aperiodically, we mark its rate as event-triggered. The release offset, $o_i(c_{in})$, is defined as the distance in time between the start of the transaction's invocation period and the event/data arrival at c_{in}. It is used to model the synchronization among different inputs. The deadline, $D_i(c_{out})$, specified for each output component c_{out}, bounds the duration from the start of the invocation period to the completion of the output component c_{out}. We allow the transaction T_i's period $P_i \leq D_i(c_{out})$ (where $P_i = 1/r_i$), implying that there can be multiple active instances of a transaction. However, every component c_i must satisfy $F(c_i) \leq P_i$, implying that component c_i can have at most one active instance at any given time.

The system software model can then be represented by a set of concurrent transactions in each system mode that represents the system runtime status. We allow data communications among concurrent transactions in such a software architecture model. Keeping the communications in a transaction following the SDF model, we relax the directed links of the data communications between different transactions to carry no firing tokens. Therefore, a component may start its execution upon availability of its inputs in the same transactions and without block-waiting for data from other transaction(s). In case the data among different transactions are carrying firing tokens, for example, in the multirate control scenario, signal-aggregation functions are required to satisfy the balance equations of SDF. The software architecture model may also contain multiple concurrent transaction sets, each of which contains transactions for a given system mode. Different components, communications, and e2e timing constraints are allowed to be specified for the same or different transactions running in different modes.

Fig. 1 shows an example of a software architecture model with the above definition. The model contains three transactions, T_1, T_2, and T_3, with their components allocated to two computation devices. T_1 has an input component c_1 with rate r_1 and release offset o_1 and an output component c_5 with deadline D_1. T_2 is an event-triggered (aperiodic) transaction with an input c_6 triggered by an external event and a deadline $D2$ for its output c_7. T_2 and T_1 communicate via components c_7 and c_3. The data passed from c_7 to c_3 does not carry any firing token, modeled as a dashed link. T_3 has two inputs, c_8 and c_9, with rates and offsets of (r_2, o_2) and (r_2, o_3), respectively, allocated on different devices. T_3 also has two output components, c_{12} and c_{13}, with their relative deadlines D_3 and D_4, respectively.

2.2 Task Model

A *task* in our system is the basic unit that can be scheduled directly by the supporting software, such as OS, on a computation device. In most of today's supporting software, a task is typically implemented as a process or thread. We assume that the block-waiting thread architecture in [19] is used to implement tasks. The thread execution follows run-to-completion semantics. In the task model, we make no assumptions on how a task is scheduled for its execution.

Definition 2. *A* task *is defined as* $\tau =< \Phi, P, d, o, w, loc >$, *where*

- Φ*: a sequence of components;*
- P*: task's invocation period;*
- d*: task's relative deadline;*
- o*: task's release time offset;*
- w*:* $\tau \rightarrow Q_0^+$ *is the task's resource demand with* $w(\tau) = \sum_{c \in \Phi} F(c) + \sum_{l \in \Phi} F(l)$*; and*
- *loc:* $\tau \rightarrow N^+$ *maps each task to an integer representing a computation device.*

In this model, a task may contain components from different transactions. All components in a task are executed sequentially with run-to-completion semantics. In other words, Φ defines a total order of the components in a task τ. If component c_{i-1} precedes component c_i in Φ (i.e., $c_{i-1} \prec c_i$), c_i cannot start before the completion of c_{i-1}, regardless of the relationship between c_{i-1} and c_i in the original model. $loc(\tau)$ restricts each task to not crossing the boundary of a computation device, meaning that no task contains the components allocated to different computation devices. This condition, along with the static allocation of components, binds each task statically to a computation device. The resource demand of a task includes the resource demands for the constituent components' computation, $F(c)$, and for their communications, $F(l)$. Since the components of a task reside on the same device and execute sequentially, $F(l)$ in a task can be ignored. Thus, we use $w(\tau) = \sum_{c \in \Phi} F(c)$ as τ's resource demand in the rest of this paper.

The runtime software architecture can then be modeled as a set of *task graphs*, each of which models the runtime architecture of the concurrent transaction set for a system mode. A task graph consists of the above-defined tasks as nodes and internode communications as edges. The readiness for execution of a task is controlled by 1) the task's release offset, which is defined as the duration between the start of its period and the time when the task can run, and 2) the data generated by the task's predecessors and required for the task's execution. The model of computation for a task graph with thus-defined tasks is a timed multithreading model [15] with a relaxation that the task execution is controlled by the release guard protocol [23].

Fig. 2 shows an example of task graph for the software system in Fig. 1. In this task graph, components from different concurrent transactions (c_1, c_2, c_3 from T_1 and c_9 from T_3) can be grouped and sequenced to run in the same

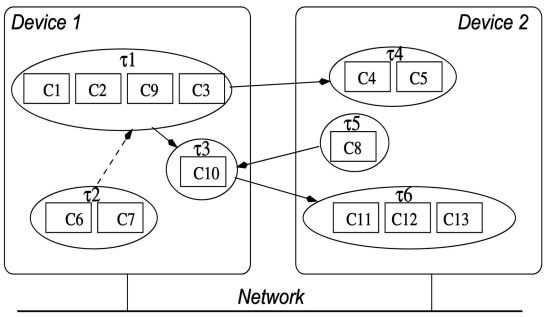

Fig. 2. An example of constructed task set for the software model in Fig. 1.

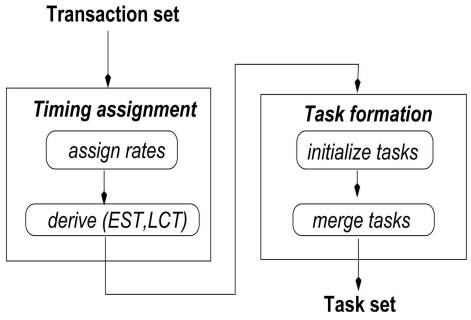

Fig. 3. Task-construction process.

task (τ_1). The dashed link between τ_2 and τ_1 implements the data communication without a firing token between c_7 and c_3 in Fig. 1. It indicates that the release of τ_1 depends on its release offset only. Other links in Fig. 2, representing task dependencies with release guard control, implement the data dependencies with firing tokens in Fig. 1.

2.3 Problem Statement

Given the above definitions and assumptions, our task-construction problem can be regarded as a model-transformation problem, determining which components should be in which task, their execution sequence, and the tasks' timing attributes. This task-construction problem is stated formally as follows:

Given a software architecture model containing concurrent transactions $T = \{T_1, T_2, \ldots, T_m\}$ and the e2e timing constraints in $\{H_i\}$, transform T into tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ such that

- S1. *$\forall c_k \in C$, there exists one and only one task $\tau_i \in \tau$ with $c_k \in \Phi_i$.*
- S2. *Meeting Φ_i, P_i, d_i, and o_i of every task $\tau_i \in \tau$ leads to satisfaction of the precedence constraints in L_i and the system-level timing constraints in H_i of every transaction $T_i \in T$.*
- S3. *The total number of the thus-constructed tasks $|\tau|$ is minimal while the Φ_i of every τ_i executes contiguously.*

We want to develop a method for automatically transforming T to τ while satisfying S1, S2, and S3. In such a transformation, S1 ensures that the constructed tasks perform the same system functionality (as before the transformation) and with the same application workloads. S2 guarantees the implementation with the thus-constructed tasks to meet the original system constraints. S3 minimizes the total runtime task-management overhead while still maintaining the unblocking nature of tasks. Note that the problem of meeting these conditions with a given set of communicating components is NP-hard [26], so we need heuristics to make the transformation scalable to large embedded systems.

During the transformation, tasks are generated for each set of concurrent transactions under the same system mode. For a multimode system, the transformation needs to be repeated for each mode. Repeating the transformation for each mode may result in the same component(s) assigned to different tasks in different modes. This is acceptable at runtime and does not violate S1 since the system modes are mutually exclusive and the system reorganization is inevitable during mode transition. After the tasks are constructed, generation of a task graph is a trivial process of identifying the communicating components in different tasks and creating links for communications among those tasks.

3 The Task-Construction Method

The task-construction problem is complex as there are a large number of ways to group the components in a software model to form a task set. Different ways of grouping components result in different resource consumptions and performances of the final system. One, therefore, has to make design trade-offs and evaluate/validate whether all constraints are met, or not, at each construction step.

Our task construction relies on a two-step process to search for a solution that maintains the schedule flexibility and minimizes the runtime overheads. This process is illustrated in Fig. 3.

3.1 Timing Assignment

The first step in the task construction process, called *timing-assignment*, determines the timing constraints of each component, which include the invocation rate, the earliest start time (EST), and the latest completion time (LCT) of the component. They are assigned conservatively so that meeting these constraints will automatically satisfy the system-level e2e timing constraints.

The rate assignment is achieved through a *rate-propagation* process. As the rates are only specified for the input components of a transaction, the process uses a breadth-first-like algorithm to assign the components' rates, starting from the inputs. It assigns the rates of reachable components, whose rates have not been assigned, to the rate of their predecessors along the directed links, until the output component(s) is reached.

Derivation of EST starts from the input component(s) and traverses the transaction T_s by following directed links toward the output component(s). During the visit to component c_i, the method determines $EST(c_i)$ along the current incoming link. For c_i with $m \geq 1$ incoming links, $EST(c_i)$ is finalized as the latest EST among all its incoming links.

$$EST(c_i) = \begin{cases} max_{1 \leq j \leq m}\{EST(c_{i-1}^j) + F(c_{i-1}^j) + F(l_{i-1,i}^j)\} \\ \qquad\qquad \text{if } c_i \text{ is not an input component} \\ o_{T_s}(c_i) \qquad \text{if } c_i \text{ is an input component,} \end{cases} \tag{1}$$

where $l^j_{i-1,i}$ is the link between c_i and its immediate predecessor c^j_{i-1} along the jth incoming link ($1 \le j \le m$), $F(c_i)$ and $F(l^j_{i-1,i})$ are the resource demands of component c_i and link $l^j_{i-1,i}$, respectively; $o_{T_s}(c_i)$ is c_i's release offset if c_i is an input component; the max operator chooses the latest EST among all c_i's incoming links.

$LCT(c_i)$ is derived similarly by traversing a transaction from each output component backward to the input component(s). Given c_i has $m \ge 1$ outgoing links, $LCT(c_i)$ is determined after all its successors' LCTs are assigned, as follows:

$$LCT(c_i) = \begin{cases} min_{1 \le j \le m}\{LCT(c^j_{i+1}) - F(c^j_{i+1}) - F(l^j_{i,i+1})\} \\ \qquad \text{if } c_i \text{ is not an output component} \\ D_{T_s}(c_i) \qquad \text{if } c_i \text{ is an output component,} \end{cases} \tag{2}$$

where $l^j_{i,i+1}$ is the link between c_i and its immediate successor c^j_{i+1} along the jth outgoing link; $D_{T_s}(c_i)$ is the deadline specified at the output component; the min operator chooses the earliest LCT among all of c_i's outgoing links.

The components' ESTs and LCTs derived using the above method form a necessary condition for a *valid* and *feasible* schedule of the transaction. A schedule is said to be *valid* if all components' precedence constraints are preserved. A schedule is said to be *feasible* if all of the e2e timing constraints are met. This is shown in Lemma 1.

Lemma 1. *For any component schedule Φ with component sequence $< c_1, c_2, \ldots, c_n >$, Φ is valid and feasible only if the start and completion times of any component c_i in Φ, denoted by $s(c_i)$ and $e(c_i)$, respectively, satisfy both $EST(c_i) \le s(c_i)$ and $e(c_i) \le LCT(c_i)$.*

Proof. The proof is straightforward by using contradiction. For feasibility, any component c_i with $s(c_i) < EST(c_i)$ in Φ implies that c_i starts before the completion of some of its predecessors, contradicting the fact that Φ is valid. Thus, $EST(c_i) \le s(c_i)$ must be true. Similarly, any component with $LCT(c_i) < e(c_i)$ makes Φ infeasible. Thus, $e(c_i) \le LCT(c_i)$ must be true. □

Lemma 1 holds for the timing assignments of both the components in a transaction and the components of concurrent transactions communicating through data with firing tokens. According to Lemma 1, the pair $(EST(c_i), LCT(c_i))$ defines an execution time window for component c_i in a valid and feasible schedule. We can, therefore, use ESTs and LCTs to verify the satisfaction of the constraints and prune those solutions that violate them. ESTs and LCTs can further be used to verify the satisfiability of e2e timing constraints before tasks are constructed and software scheduling mechanisms are determined. In case the condition $LCT(c_i) - EST(c_i) \ge F(c_i)$ is not met for any component, no feasible schedule can possibly be found and the design parameters must be changed.

3.2 Task Formation

The second step in the process, called *task formation*, constructs tasks by grouping and sequencing the components in the transactions. The objective of this step is to generate a minimum task set in which each task can run continuously if it runs alone. Tasks in such a set can be directly implemented as processes/threads on a target platform and do not require special support for task-execution control from the underlying system. To generate such tasks, the components in each task must 1) run at the same rate, 2) be located on the same device, and 3) execute sequentially in a run-to-completion manner. Further, the task should not contain any idle time inside itself. Task τ's internal idle time, denoted by $\theta(c_i)$, is the time between the completion of a component c_i, $e(c_i)$, and the start of its immediate successor c_{i+1}, $s(c_{i+1})$, in τ's component schedule Φ, i.e., $\theta(c_i) = s(c_{i+1}) - e(c_i)$, where c_i and c_{i+1} both belong to τ. If there exists an internal idle time within τ, then $max_{\forall c_i}(\theta(c_i)) > 0$. The existence of internal idle times within a task requires special mechanisms to control the component interrelease time within the task (for example, the system function call *delay()* or *sleep()* implemented in OS), which complicates task implementation, scheduling, and runtime execution management.[2] Note that the blocking and preemption times during the execution of a task are not considered as internal idle times under this definition.

Given $c_1, \ldots, c_n$ are the components of a concurrent transaction set $T = \{T_1, \ldots, T_m\}$ with the components' timing constraints assigned, our task-construction process starts with an initial set of n tasks $\tau = \{\tau_1, \ldots, \tau_n\}$, each of which contains one component, and is modeled as $\tau_i =< c_i, 1/r(c_i), LCT(c_i), EST(c_i), F(c_i), loc(c_i) >$. These tasks are then merged iteratively to minimize the total number of tasks in the final system. The components from different tasks are sequenced during the merge according to their timing constraints. In this process, we must address two issues: 1) determine which tasks are to be merged and 2) sequence the components after the merge.

Selection of tasks to be merged. Tasks are merged iteratively. At each iteration, our algorithm selects two tasks on the same computation device for merge according to *rate similarity* and *execution overlap*. *Rate similarity* requires the selected tasks to have harmonically related invocation rates. *Execution overlap* requires the selected tasks to have overlapping execution windows. These criteria ensure the nonexistence of task internal idle times, as proven in Lemma 2.

Lemma 2. *Given two tasks $\tau_i =< \Phi_i, P_i, d_i, o_i, w_i, loc_i >$ and $\tau_j =< \Phi_j, P_j, d_j, o_j, w_j, loc_j >$ $(P_i \le P_j)$ with contiguous component sequences $\Phi_i = \{a_1, a_2, \ldots, a_m\}$ and $\Phi_j = \{b_1, b_2, \ldots, b_n\}$, τ_i and τ_j are merged to form a new task τ_{ij} without any internal idle time in Φ_{ij} if and only if:*

1. *P_i is a divisor of P_j ($P_j = NP_i$, where N is an integer).*
2. *Executions of τ_i and τ_j overlap at the beginning:*

$$max(o_i, o_j) - min(o_i, o_j) \le \begin{cases} w_i & \text{if } o_i \le o_j; \\ w_j & \text{otherwise.} \end{cases} \tag{3}$$

3. *Φ_j has a sufficiently long execution time to fill idle times between the invocations of Φ_i:*

2. The duration arguments passed in the system function call depend on the execution status of a task. They must be computed dynamically and, thus, are inaccurate and introduce overhead.

$$w_j + (N-1)w_i \geq \begin{cases} (N-1)P_i & \text{if } o_i \leq o_j; \\ (N-1)P_i + o_i - o_j & \text{otherwise.} \end{cases} \tag{4}$$

Proof. *Sufficiency:* According to Condition 1, $P_j = NP_i$, we need to consider only one cycle of P_j. Denote the start and completion times of component $a_k \in \Phi_i$ by $s_i(a_k)$ and $e_i(a_k)$, those of component $b_l \in \Phi_j$ by $s_j(b_l)$ and $e_j(b_l)$, and those of component c_i ($c_i = a_k$ or b_l) in merged Φ_{ij} by $s_{ij}(c_i)$ and $e_{ij}(c_i)$.

Suppose $o_i \leq o_j$. We have

$$\begin{aligned} & o_j - o_i \leq w_i, o_j = s_j(b_1), o_i = s_i(a_1) \\ \Rightarrow\ & s_j(b_1) - s_i(a_1) \leq w_i \\ \Rightarrow\ & s_i(a_1) \leq s_j(b_1) \leq s_i(a_1) + w_i = e_i(a_m). \end{aligned}$$

So, the executions of Φ_i and Φ_j overlap. According to Condition 3, we have

$$\begin{aligned} & w_j + (N-1)w_i \geq (N-1)P_i \\ \Rightarrow\ & w_j \geq (N-1)(P_i - w_i). \end{aligned}$$

Since the maximum idle time during P_i is $P_i - w_i$ when τ_i runs alone, the maximum idle time during $P_j - P_i$ when running τ_i alone is $(N-1)(P_i - w_i)$. So, $w_j \geq (N-1)(P_i - w_i)$ implies that the idle time between the completion of the current Φ_i and the start of the next Φ_i during P_j can be filled by Φ_j.

Similarly, for the case of $o_i > o_j$, $o_i - o_j \geq w_j$ ensures the overlapping execution of τ_i and τ_j and w_j is long enough to fill not only all $(N-1)(P_i - w_i)$, but also between o_j and o_i.

According to the above discussion, for any pair of components c_i and c_{i+1} in Φ_{ij} after the merge, if $c_i = a_k, c_{i+1} = a_{k+1}$, $\theta(c_i) = \theta(a_k) = 0$ since Φ_i has no internal idle time. Similarly, if $c_i = b_l, c_{i+1} = b_{l+1}$, $\theta(c_i) = \theta(b_l) = 0$ since Φ_j has no internal idle time. If $c_i = a_k, c_{i+1} = b_l$, according to Conditions 2 and 3, $s_{ij}(c_i) < s_j(b_l) < e_{ij}(c_i)$. This results in $s_{ij}(b_l) = e_{ij}(c_i)$. Thus, $\theta(c_i) = 0$. Similarly, we can prove $\theta(c_i) = 0$ for $c_i = b_l, c_{i+1} = a_k$. Therefore, satisfying the three conditions can lead to $\theta(c_i) = 0$ for all components in the merged component sequence Φ_{ij}.

Necessity: Suppose P_i is for Φ_i to start with a_1 and P_j for Φ_j to start with b_1 ($o_i \leq o_j$), the difference of the release offsets between Φ_i and Φ_j at the kth cycle of P_j, denoted by $\Delta^k(b_1)$, can be computed as

$$\begin{aligned} & \Delta^k(b_1) = (kP_j + o_j) - min\Big(kP_j + o_j, \Big\lfloor \frac{kP_j}{P_i} \Big\rfloor P_i + o_i\Big) \\ & k = 0, 1, 2, \ldots, \tfrac{LCM(P_i,P_j)}{P_j} - 1. \end{aligned} \tag{5}$$

Suppose a_i in Φ_i is the immediate predecessor of b_1 in Φ_{ij}, then $\theta_{ij}(b_1) = s_{ij}(b_1) - e_{ij}(a_i) \leq \Delta^k(b_1)$. To guarantee $\theta_{ij}(b_1) = 0$, $\Delta^k(b_1) = 0$ must hold for any k. According to (5), $\Delta^k(b_1) = 0$ holds when P_i is a divisor of P_j, i.e., $P_j = NP_i$. The case of $o_i > o_j$ can be proven similarly.

For Condition 2, let us assume $o_{ij} = o_i$ (a_1 is the first component in Φ_{ij}). This implies $o_i \leq o_j$. Since $\theta_{ij}(c_i) = 0$ for any c_i in Φ_{ij}, b_1 must satisfy $\theta_{ij}(b_1) = 0$. This implies the existence of a_i in Φ_i such that $e_{ij}(a_i) = s_{ij}(b_1)$. Therefore,

$$\begin{aligned} & e_{ij}(a_i) = s_{ij}(b_1) \\ \Rightarrow\ & s_{ij}(a_i) < s_j(b_1) < e_{ij}(a_i) \\ \Rightarrow\ & s_i(a_1) < s_j(b_1) \leq e_i(a_m) \\ \Rightarrow\ & s_j(b_1) \leq s_i(a_1) + w_i, s_i(a_1) = o_i, s_j(b_1) = o_j \\ \Rightarrow\ & o_i + w_i \geq o_j \\ \Rightarrow\ & o_j - o_i \leq w_i \\ \Rightarrow\ & max(o_i, o_j) - min(o_i, o_j) \leq w_i. \end{aligned}$$

Likewise, we can prove that the condition is also true for the case of $o_i > o_j$.

For Condition 3, let us assume $o_{ij} = o_i$ first. Φ_{ij} has $\theta(c_i) = 0$ for any component c_i. Given $P_j = NP_i$, the completion time of Φ_{ij} can be computed as

$$\begin{aligned} & o_{ij} + w_{ij} + \textstyle\sum_k \theta_{ij}(c_k) = o_{ij} + w_{ij} \\ \Rightarrow\ & o_i + w_{ij} = o_i + w_j + Nw_i \\ \Rightarrow\ & o_i + w_j + Nw_i = o_i + w_j^1 + (N-1)w_i + w_i + w_j^2 \\ \Rightarrow\ & o_i + w_j + Nw_i \geq o_i + (N-1)P_i + w_i \\ \Rightarrow\ & w_j + (N-1)w_i \geq (N-1)P_i. \end{aligned}$$

In the above derivation, w_j^1 is the resource demands by the components in Φ_j that are executed during $(o_i, (N-1)P_i + o_i)$, while w_j^2 is the resource demands by the remaining components in Φ_j. Similarly, for the case of $o_{ij} = o_j$, the above derivation will become

$$\begin{aligned} & o_{ij} + w_{ij} + \textstyle\sum_k \theta_{ij}(c_k) = o_{ij} + w_{ij} \\ \Rightarrow\ & o_j + w_{ij} = o_j + w_j + Nw_i \\ \Rightarrow\ & o_j + w_j + Nw_i \geq o_i + (N-1)P_i + w_i \\ \Rightarrow\ & w_j + (N-1)w_i \geq o_i - o_j + (N-1)P_i. \end{aligned}$$

□

According to Lemma 2, we can first merge τ_i and τ_j on the same computation device with $P_i = P_j$, then with $P_j = N \cdot P_i$. For the latter, we can roll up τ_i to create N tasks $\tau_i^k =< \Phi_i, P_j, d_i^k, o_i^k, w_i, loc_i)$ $(1 \leq k \leq N)$ during each cycle of P_j with the parameters modified as:

- $d_i^k = (k-1) \cdot P_i + d_i$;
- $o_i^k = (k-1) \cdot P_i + o_i$.

These N tasks $\tau_i^1, \ldots, \tau_i^N$ can then be merged with τ_j as individual tasks of the same rate. Note that only those tasks that meet all conditions in Lemma 2—not tasks that only meet $P_i = N \cdot P_j$—will be rolled up. This ensures that the tasks are either merged into one (if it passes the timing check) or are kept separate (if it fails the timing check) when the process completes. Thus, the size of a task set after each merge is always nonincreasing. We further constrain the merge to start with the tasks with the minimum computation resource demands $w(\tau_i)$ when there are more than two eligible tasks, thus making task selection deterministic.

For the components in aperiodic transactions, our process merges those of the same transaction with their executions overlapping into one task. The resulting tasks are aperiodic.

Sequencing components. The components within a task must execute sequentially. Sequencing components subject to their timing constraints is NP-hard [7]. Both EST and LCT have been used as heuristics in real-time systems to sequence tasks in a schedule to meet precedence constraints. Since our objective is to generate a near-minimum task set without introducing internal idle times, heuristics

used for scheduling, such as latest-start-time and deadline monotonic that can yield near-optimal schedule, may not be optimal for generation of the minimum task set without internal idle times. Thus, we choose minimum-*EST*-first as our heuristic to execute the components as early as possible in the merged task.

We adopt a branch-and-bound process to determine the sequence of components. EST is used to expand a branch and the component's execution time window (EST, LCT) is used to bound the search. Specifically, given $\Phi_i = < a_1, \ldots, a_n >$ and $\Phi_j =< b_1, \ldots, b_m >$ are the component sequences of two tasks, τ_i and τ_j, with $o_i < o_j$, we insert $b_1, \ldots, b_m$ into Φ_i in their execution order in Φ_j to form Φ_{ij} of the merged task τ_{ij}. Let the start time of component a_l in Φ_i be $s_i(a_l)$ and its completion time be $e_i(a_l)$. An eligible position for b_j (assuming after a_i) in Φ_{ij} should satisfy: 1) $EST(a_i) < EST(b_j) \leq EST(a_{i+1})$, 2) $max\{e_{ij}(a_i), e_{ij}(b_{j-1})\} \leq s_j(b_j)$, and 3) no timing constraints of $b_{j+1}, \ldots, b_m$ and $a_{i+1}, \ldots, a_n$ are violated after inserting b_j. The start time of $b_{j+1}, \ldots, b_m$ in Φ_j and $a_{i+1}, \ldots, a_n$ in Φ_i should then be updated using

$$s_i(a_{i+1}) = e_{ij}(b_j), s_i(a_k) = s_i(a_{k-1}) + F(a_k), \tag{5}$$

$$s_j(b_{j+1}) = e_{ij}(b_j), s_j(b_l) = s_j(b_{l-1}) + F(b_l). \tag{6}$$

If there does not exist any position for b_j that satisfies all of the above conditions, we consider the process failed and, hence, keep the two tasks separate. The algorithm for sequencing components in a task merge is detailed in Algorithm 1.

Algorithm 1 Component sequencing.
begin
1 initialize the search with start-point $p \leftarrow 1$ and end-point $q \leftarrow n$;
2 **for** $j = 1$ to m **do**
3 find i between $[p, q]$ in Φ_i with $EST(a_i) < EST(b_j) \leq EST(a_{i+1})$;
4 **if** no such i **then exit-loop**;
5 find k between $[i, q]$ in Φ_i with $s_i(a_k) \leq LCT(b_j) - F(b_j) < s_i(a_{k+1})$;
6 **if** no such k **then exit-loop**;
7 find a position l between $[i, k]$ in Φ_i to insert b_j;
8 **if** no such l can be found **then return** $fail$;
9 **else**
10 insert b_j to Φ_i before l;
11 update $s_j(b_k)$ for $b_{j+1}, \ldots, b_m$;
12 update $s_i(a_k)$ for $a_{l+1}, \ldots, a_n$;
13 **if** any $a_{l+1}, \ldots, a_n$ with $e_i(a_p) > LCT(a_p)$ or any $b_{j+1}, \ldots, b_m$ with $e_j(b_q) > LCT(b_q)$ **then return** $fail$;
14 $p \leftarrow l$;
15 **end-if-else**
16 **end-for**
17 **if** $j < m$ **then**
18 add the rest components $b_j \in \Phi_j$ to the end of Φ_i;
19 revise $s_{ij}(b_j), e_{ij}(b_j)$ for these components;
20 **end-if**
21 **return** $succ$, $\Phi_{ij}(\Phi_i)$;
end.

Lemma 3 shows that the task merging with the above algorithm does not introduce any internal idle time.

Lemma 3. *Given two tasks τ_i and τ_j with the same rate, if both component sequences Φ_i and Φ_j are with $\theta_i(a_i) = 0$ and $\theta_j(b_j) = 0$ and $o_i < o_j < o_i + w_i$ or $o_j < o_i < o_j + w_j$, then the merged sequence Φ_{ij} that is obtained from using Algorithm 1 has $\theta_{ij}(c_i) = 0$ for every c_i.*

Proof. We prove this lemma by contradiction. Suppose there exists a c_i in Φ_{ij} such that $\theta_{ij}(c_i) > 0$ in Φ_{ij} of the merged task τ_{ij}, then

$$\theta_{ij}(c_i) = s_{ij}(c_{i+1}) - e_{ij}(c_i) > 0.$$

We examine the following two cases:

Case 1: c_i, c_{i+1} are from the same task τ_i. Since c_i and c_{i+1} are immediately next to each other in Φ_{ij}, c_i, c_{i+1} must be immediately next to each other in Φ_i according to the algorithm. $\theta_i(c_i) = s_i(c_{i+1}) - e_i(c_i) > 0$. This contradicts $\theta_i(a_i) = 0$ for Φ_i.

Case 2: c_i, c_{i+1} are from different tasks. Without loss of generality, we can assume $c_i \in \Phi_i$ and $c_{i+1} \in \Phi_j$. According to the algorithm, c_i and c_{i+1} are next to each other in Φ_{ij} after merging only if there exists $< c_i, c_j > \in \Phi_i$ between which c_{i+1} is inserted. Suppose Φ_{ij}^k and Φ_{ij}^{k+1} are the sequences before and after inserting c_{i+1} and there are k components in Φ_j that have been merged with Φ_i.

$$\Phi_{ij}^k = c_1, \ldots, c_i, c_j, \ldots, c_{n+k}$$
$$\Phi_{ij}^{k+1} = c_1, \ldots, c_i, c_{i+1}, c_j, \ldots, c_{n+k+1}.$$

After inserting c_{i+1}, we have $s_{ij}^k(c_j) = s_{ij}^{k+1}(c_{i+1})$ and $e_{ij}^k(c_i) = e_{ij}^{k+1}(c_i)$. If $\theta_{ij}(c_i) > 0$, then

$$\begin{aligned} & s_{ij}(c_{i+1}) - e_{ij}(c_i) > 0 \\ \Rightarrow\ & s_{ij}^{k+1}(c_{i+1}) - e^{k+1}(c_i) > 0 \\ \Rightarrow\ & s_{ij}^k(c_j) - e_{ij}^k(c_i) > 0 \\ \Rightarrow\ & s^{k-1}(c_j) - e^{k-1}(c_i) > 0 \\ \Rightarrow\ & s_{ij}^0(c_j) - e_{ij}^0(c_i) > 0. \end{aligned}$$

Φ_{ij}^0 is the sequence of Φ_i without any component in Φ_j inserted. So, $\Phi_{ij}^0 = \Phi_i$. According to the above derivation, we must have $\theta_i(c_i) = s_i(c_j) - e_i(c_i) > 0$. This contradicts the fact that Φ_i is contiguous.

Since only the above cases are possible for immediate adjacent components in the merged sequence, we can conclude that $\theta_{ij}(c_i) = 0$ for Φ_{ij}. □

After the selected tasks are merged and their components are sequenced, we can finalize the merged task's properties of

$$\tau_{ij} = (\Phi_{ij}, P_{ij}, d_{ij}, o_{ij}, w_{ij}, loc_{ij})$$

with

$$\begin{aligned} & P_{ij} = P_i = P_j, \\ & d_{ij} = \begin{cases} d_i + w_j & \text{if } d_i \leq d_j \\ d_j + w_i & \text{if } d_i > d_j \end{cases} \\ & o_{ij} = min(o_i, o_j), \\ & w_{ij} = w_i + w_j, \\ & loc_{ij} = loc_i = loc_j. \end{aligned}$$

TABLE 1
Parameters for Scalability Experiments

parameter	value in random generation
# of transactions	$5 \sim 20$
in/out links/component	$1 \sim 5$
# of input rates	$5 \sim 10$
component utilization	$0.005 \sim 0.05$
communication/link	$10 \sim 100$

Theorem 1. *The tasks constructed above satisfy the three conditions in the problem statement.*

Proof. For S1, since the task construction starts with each component in a task and no component is duplicated in the merge, the constructed task set satisfies S1. For S2, since the components in the constructed tasks satisfy the derived components' timing constraints, all output components will meet their deadlines (according to the derivation of EST and LCT). According to Lemma 1 and minimum-EST-first heuristic, the constructed task set meet both e2e system-level timing constraints and components' precedence constraitns. For S3, according to Lemmas 2 and 3, the constructed tasks are all with contiguous executions. The set of tasks is minimal as the tasks cannot be merged any further without introducing internal idle times. □

Note that we make no assumption on the implemented scheduling algorithm on the final target. This allows our task-construction process to be used in the software design independently of the platform design. After the target platform is decided, the schedulability analysis is required to ensure that the generated task set is indeed schedulable on the platform with the given scheduling algorithm. In case the generated task set is not schedulable on a platform when all information becomes available, some iterative refinement, as described in [17], is necessary to improve the design. Optimization can also be applied to the obtained task set to improve the performance and resource usage after the details of the platform are decided.

4 Evaluation

We now evaluate the task-construction method described thus far using a set of randomly generated models, focusing on its scalability and effectiveness. A baseline construction method was implemented and used as a reference for comparison. It adopted an algorithm that synthesizes tasks using components of the same transaction running at the same rate on the same processor. This is similar to the specialized architecture presented in [19] and is based on the fact that such a strategy is commonly seen in current industrial practices [6], [24].

We first evaluated the scalability of our task-construction method. The metric used in this evaluation was the number of algorithm steps taken to generate a task set. The number of steps reflects the size of the design space explored, thus indicating the method's scalability. A more scalable method explores fewer nodes and executes fewer steps. In each experiment, we first generated a system model and applied both the baseline and proposed methods to the model. Each experiment used a model with a fixed number of components ranging from 100 to 1,000 at an increment of 100. Other parameters of the random generation of the system models are given in Table 1. To simplify the experiments, we fixed the number of computation devices in the platform at five and allocated the components to these computation devices using the allocation algorithm in [27] with a load-balancing policy.

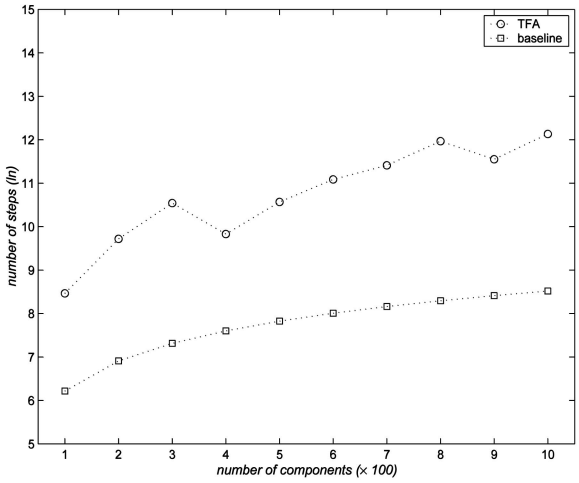

Fig. 4. Number of steps taken to form tasks.

Fig. 4 shows the number of steps taken to construct the task set using our algorithm (TFA) and using the baseline algorithm (baseline). To fit the results in the diagram, we used the natural logarithm of steps, $ln(s)$. It showed that the number of steps required to form the task set increased with the number of components for both the baseline method and TFA. TFA always took more steps than the baseline method because it requires merging and sequencing components from different transactions and with different rates. Despite the additional steps TFA took, it demonstrated the same trend, with more variation, of the required number of steps. The same trend implies that TFA is scalable as the baseline method. The variation indicates that TFA is more sensible to the model properties (connectivity, resource demands, and rates) than the baseline as these properties have a significant impact on the selection and sequencing of merged tasks.

We also investigated the effectiveness of TFA for a given system model. The total number of resultant tasks was chosen as the metric. A method is considered more effective if it generates a smaller task set while meeting all system constraints. Unlike the evaluation of scalability, the experiments for effectiveness evaluation used randomly generated models with a fixed number of rates while varying from 5 to 30 at an increment of 5. The rates used in the experiments were predefined and organized in two different groups with rates harmonically related within each group but not across groups to reflect the existence of multiple rate groups in real-world applications. The number of transactions in the system model was randomly generated and multiple transactions were allowed to run at

TABLE 2
Parameters for Effectiveness Experiments

parameter	value in random generation
# of transactions	$10 \sim 100$
# of component/transaction	$20 \sim 50$
# of transactions/rate	$1 \sim 5$
# of outgoing link	$1 \sim 3$ (low concurrency)
	$3 \sim 5$ (high concurrency)

one rate. Further, we considered the factors of system workload and component concurrency in this evaluation. The system workload may result in overlapping executions of different components. We chose the system workload to be light (utilization = 0.1), medium (utilization = 0.3), or heavy (utilization = 0.6). To minimize the noises introduced by different generations of the system model in different experiments, we generated a model with the heavy workload first, then scaled the workload of every component equally to obtain the medium and light workloads without changing the structure of the system model. Component concurrency may also affect the resultant number of tasks since fewer component dependencies may result in more components merged into a single task. We controlled the model concurrency in each experiment by constraining the number of outgoing links allowed for each component during the random generation of the model. A larger number of components' outgoing links yields a model with higher concurrency. For the same reason of minimizing experiment noises, we first generated the system model with high component concurrency, then adjusted the links among the components to reduce the concurrency without altering the other properties of the model (i.e., the number of components, components' workloads, and system rates). As in the scalability evaluation, we fixed the computation devices in the platform at five and allocated the components to these devices using a load-balancing policy. Other parameters used in the random generation of the system model for the experiments are given in Table 2.

The results of these experiments are shown in Fig. 5 and Fig. 6. For the baseline, we only showed the results under heavy workload since the results under medium and light workloads were the same. This was because we chose to keep the same model structure and scale the components' workload equally. As the results of the baseline depended only on the locations, rates, and links of the components, changing the components' workloads had no effect on the results given that the allocation strategy (algorithm and policy) and the model structure did not change.

TFA was shown to generate fewer tasks than the baseline method in all experiments. For all cases, the number of tasks generated by both the baseline and TFA increased as the number of system rates increased, but at a slower speed for TFA. Hence, TFA is more effective than the baseline for a system model with more rates. This increasing effectiveness with the number of rates is a direct result of considering the components with harmonically related rates in TFA. As the workload increased, the difference in the results between TFA and the baseline also increased, meaning that the difference between TFA and the baseline was less

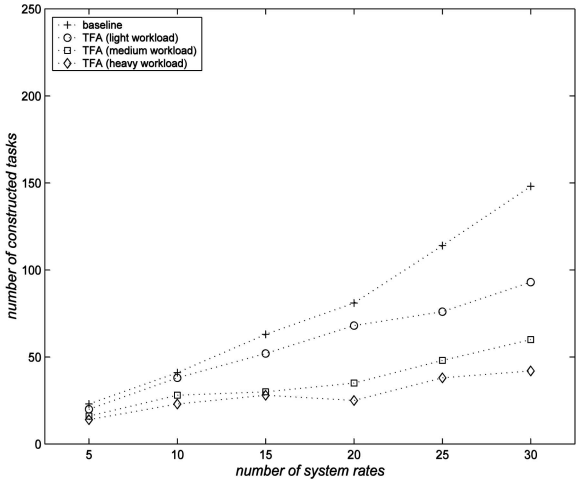

Fig. 5. Number of resultant tasks (high concurrency).

significant for light workloads than for heavy workloads, regardless of component concurrency. This indicates that the TFA is more effective for heavily loaded systems. In such a case, more components may have overlapping executions and, therefore, can be merged. A method that becomes more effective under heavy workloads is desirable because resources must be managed better when the system deals with heavy workloads than with light workloads. Both methods were shown to construct fewer tasks for higher concurrency. The difference between different concurrency levels was more pronounced for medium and light workloads than for heavy workloads. Better results under high concurrency come from fewer dependencies, thus merging more components in one task. A low concurrency model, on the other hand, contains more dependent components on different computation devices, which cannot be grouped into one task. The effect of model concurrency can be reduced significantly for a system with heavy workloads due to more overlapping components' executions. Note that a smaller task set for high concurrency may also be the result of the load-balancing allocation policy and may be reduced by using a different policy that

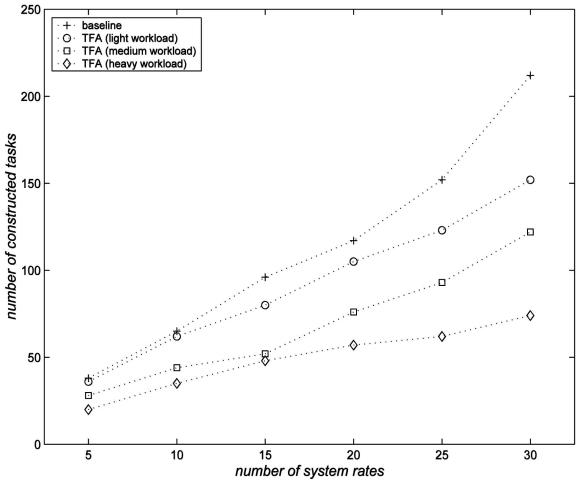

Fig. 6. Number of resultant tasks (low concurrency).

allocates more dependent components to the same device. From the above evaluation results, we can conclude that our task-construction method is both scalable and effective.

5 Related Work

Several solutions to automatic task construction using software design models based on the object-oriented modeling paradigm have been proposed. Burns and Wellings [2] and Cornwell and Wellings [3] proposed a task construction method for the HRT-HOOD model which is an object-based modeling paradigm. In this method, tasks are constructed through transaction specification and timing assignments. Since the final implementation targets using Ada programming language, some modeling constructs corresponding to Ada implementations are introduced to facilitate target generation. Saksena et al. [19] developed a method for automatic task construction based on the software design models in ROOM [20], which is an object-oriented real-time modeling language. It models the computation as event streams, with each event triggering an object action. During the task construction, the priority of each event is determined according to the schedulability analysis. Then, the actions of the components are assigned to the threads using a branch-and-bound technique, although the details of the algorithm are not available. The method is simplified in implementation by imposing architectural restrictions, such as mapping all events for the same component (or in the same transaction or with the same priority) to the same thread. A similar scenario-based task construction approach was developed in [12]. In this method, the thread is constructed based on capsule instances. However, it is up to the designer to determine how many threads in the system and which capsule instances are merged with this method. Both the above two methods need to deal with mutually exclusive data access in a shared object, which is difficult to automate. There are some task-construction methods based on process-oriented component models, such as timed multitasking system [16], Matlab/RTW [25], and ETAS ASCET [5]. Among these methods, timed multitasking can construct task automatically with requirements that all tasks must be synchronized by delivering all outputs only at the end of each timing frame. The task constructions in RTW and ASCET are still manual processes. There also exist task-construction methods using the time-triggered model [13], but they are limited to systems with periodic activities and harmonic rates.

Our approach, on the other hand, uses a process-oriented model, which naturally fits many modeling environments for control design and, hence, facilitates the integration with modeling environment. Compared to other task-construction methods, our method constructs minimal task sets with smaller overheads while meeting the system-level timing constraints. Our timing assignment is different from traditional timing derivation [8] and deadline distribution [11] in that our derived constraints form a necessary condition for a valid feasible schedule and, thus, can be used to verify the task construction during the process. The assignment imposes no requirement on the scheduling policy and, thus, the platform design can choose the one that best suits the target application.

6 Conclusions

Task construction using software design models is essential to model-based embedded control software development. It is also desirable to be able to automate to support e2e design automation using integrated design tool chains. In this paper, we presented a novel, two-step approach to the automatic task-construction problem. Taking a software model—which contains transactions with process-oriented communicating components and system-level e2e timing constraints—as input, our task construction first assigns the components' timing constraints according to the precedence and e2e timing constraints. Such timing constraints, including invocation rates, ESTs, and LCTs, form a necessary condition for building any valid feasible schedule. Then, the tasks are iteratively merged according to rate similarity and execution overlap to minimize the total number of tasks, which, in turn, minimizes the runtime overheads. While components are merged, the components from different tasks are sequenced to form a new component sequence that meets their timing constraints. Our evaluation results have shown that the proposed task-construction approach is scalable and effective. The developed methods, together with modeling methods, have been implemented and integrated in an embedded control software design tool, called *AIRES* [1] and have been applied to applications in both avionics and automotive domains.

Our future work includes in-depth evaluation and improvement of the proposed task-construction approach. First, we would like to improve our solution to support iterative design refinements according to the analysis results at a later design phase. The method will be more effective and integrable in an e2e design process if it provides options for task-construction strategies according to the later-phase analysis and allows the designer to choose from these options. Further, we would like to investigate the runtime performance of the constructed task set. This will help us determine the efficiency of the task-construction approach. We would also like to improve the approach by "optimizing" the task construction using techniques such as simulated annealing. Finally, applying the proposed approach to a wider range of applications with different types of design models and design objectives would also be worthwhile.

Acknowledgments

The work reported in this paper was supported in part by the Escher Consortium and Ford Motor Company.

References

[1] AIRES Group, http://kabru.eecs.umich.edu/aires, 2001.
[2] A. Burns and A.J. Wellings, "A Structured Design Method for Hard Real-Time Systems," Technical Report YCS-93-199, Univ. of York, Heslington, York, U.K., http://www.cs.york.ac.uk/ftpdir/reports/YCS-93-199.pdf, 1993.

[3] P. Cornwell and A. Wellings, "Transaction Specification for Object-Oriented Real-Time Systems in HRT-HOOD," *Lecture Notes in Computer Science (LNCS 1031),* pp. 365-378, 1996.
[4] D. de Niz and R. Rajkumar, "Time Weaver: A Software-through-Models Framework for Embedded Real-Time Systems," *Proc. 2003 ACM SIGPLAN Conf. Language, Compiler, and Tool for Embedded Systems,* pp. 144-152, June 2003.
[5] ETAS, Inc., "ETAS Ascet Manual," 2001.
[6] Ford Motor Company, General Motors Corporation, and Motorola Automotive and Industrial Electronics Group, "SmartVehicle Challenge Problems," http://vehicle.me.berkeley.edu/mobies/, 2000.
[7] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York: W.H. Freeman and Company, 1979.
[8] R. Gerber, S. Hong, and M. Saksena, "Guaranteeing Real-Time Requirements with Resource-Based Calibration of Periodic Processes," *IEEE Trans. Software Eng.,* vol. 21, no. 7, pp. 107-131, July 1995.
[9] A. Girault, B. Lee, and E.A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 18, no. 6, pp. 742-760, 1999.
[10] Z. Gu, S. Kodase, S. Wang, and K.G. Shin, "A Model-Based Approach to System-Level Dependency and Real-Time Analysis of Embedded Software," *Proc. Ninth IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS 2003),* pp. 78-87, May 2003.
[11] J. Jonsson and K.G. Shin, "Deadline Assignments in Distributed Hard Real-Time Systems with Relaxed Locality Constraints," *Proc. IEEE Int'l Conf. Distributed Computing Systems,* pp. 432-440, May 1997.
[12] S. Kim, S. Cho, and S. Hong, "Schedulability-Aware Mapping of Real-Time Object-Oriented Models to Multi-Threaded Implementations," *Proc. IEEE Real-Time Computing Systems and Applications Symp.,* pp. 7-14, 2000.
[13] H. Kopetz, "The Time-Triggered Model of Computation," *Proc. IEEE Real-Time Systems Symp.,* pp. 168-177, Dec. 1998.
[14] C.M. Krishna and K.G. Shin, *Real-Time Systems.* The McGraw-Hill Companies, 1997.
[15] J. Liu, "Responsible Frameworks for Heterogeneous Modeling and Design of Embedded Systems," PhD dissertation, Dept. of Electrical Eng. and Computer Science, Univ. of California at Berkeley, 2001.
[16] J. Liu and E.A. Lee, "Timed Multitasking for Real-Time Embedded Software," *IEEE Control Systems Magazine,* vol. 23, no. 1, pp. 65-75, Feb. 2003.
[17] J.R. Merrick, S. Wang, K.G. Shin, J. Song, and W. Milam, "Priority Refinement for Dependent Tasks in Large Real-Time Software," *Proc. 11th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS 2005),* pp. 365-374, Mar. 2005.
[18] T. Quatrani, *Visual Modeling with Rational Rose and UML.* Addison-Wesley, 1998.
[19] M. Saksena, P. Karvelas, and Y. Wang, "Automatic Synthesis of Multi-Tasking Implementations from Real-Time Object-Oriented Models," *Proc. IEEE Symp. Object-Oriented Real-Time Distributed Computing,* pp. 360-367, Mar. 2000.
[20] B. Selic, G. Gullekson, and P.T. Ward, *Real-Time Object-Oriented Modeling.* John Wiley & Sons, 1994.
[21] J. Stankovic, "VEST: A Toolset for Constructing and Analyzing Component Based Operating Systems for Embedded and Real-Time Systems," technical report, Univ. of Virginia, http://www.cs.virginia.edu/~techrep/CS-2000-19.pdf, 2000.
[22] D.B. Stewart, R.A. Volpe, and P.K. Khosla, "Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects," *IEEE Trans. Software Eng.,* vol. 23, no. 12, pp. 759-775, Dec. 1997.
[23] J. Sun, "Fixed-Priority End-to-End Scheduling in Distributed Real-Time Systems," PhD dissertation, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, http://junsun.net/publications/Sun97.ps.Z, 1997.
[24] The Boeing Company, "Challenge Problems for Model-Based Integration of Embedded Software: Weapon System Open Experimental Platform," 2001.
[25] The MathWorks, Inc., "Real-Time Workshop User's Guide," http://www.mathworks.com/access/helpdesk/help/toolbox/rtw, 2004.
[26] S. Wang, "Performance Modeling and Analysis Techniques and Its Application for Embedded Control Software Design," PhD dissertation, Dept. of Electrical Eng. and Computer Science, Univ. of Michigan, Ann Arbor, Sept. 2004.
[27] S. Wang, J.R. Merrick, and K.G. Shin, "Component Allocation with Multiple Resource Constraints for Large Embedded Real-Time System Design," *Proc. 10th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS 2004),* pp. 219-226, May 2004.

Shige Wang received the BE degree in computer engineering and the MS degree in computer science from Northeaster University, Shenyang, China, and the PhD degree in computer science and engineering from The University of Michigan, Ann Arbor. He is currently a senior research scientist in the Electrical and Control Integration Lab at General Motors R&D and Planning. His research focuses on embedded and control software modeling and architecture, system integration and analysis, and performance-aware model transformation. He is a member of the IEEE.

Kang G. Shin is the Kevin and Nancy O'Connor Professor of Computer Science and founding director of the Real-Time Computing Laboratory in the Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor. His current research focuses on QoS-sensitive networking and computing as well as on embedded real-time OS, all with emphasis on timeliness, dependability, and security. He has supervised the completion of 54 PhD theses and authored/coauthored more than 630 technical papers and numerous book chapters in the areas of distributed real-time computing and control, computer networking, fault-tolerant computing, and intelligent manufacturing. He has coauthored (jointly with C.M. Krishna) a textbook *Real-Time Systems* (McGraw Hill, 1997). He has received a number of best paper awards, including the IEEE Communications Society William R. Bennett Prize Paper Award in 2003 and an Outstanding *IEEE Transactions of Automatic Control* Paper Award in 1987. He has also received several institutional awards, including the Research Excellence Award in 1989, Outstanding Achievement Award in 1999, Distinguished Faculty Achievement Award in 2001, and Stephen Attwood Award in 2004 from The University of Michigan; a Distinguished Alumni Award of the College of Engineering, Seoul National University in 2002; 2003 IEEE RTC Technical Achievement Award; and the 2006 Ho-Am Prize in Engineering. He is a fellow of the IEEE.