Implementation Synthesis of Embedded Software under Operating Systems Supporting the Hybrid Scheduling Model

Zhigang Gao[1], Zhaohui Wu[1], and Hong Li[1]

[1] College of Computer Science, Zhejiang University
Hangzhou 310027, Zhejiang, China
{gaozhigang, wzh, lihong}@zju.edu.cn

Abstract. Implementation synthesis of embedded software has great influence on implementing embedded software's non-functional requirements, such as real-time, memory consumption, and low power, etc. In this paper, we focus on the implementation synthesis problem under a class of operating systems that supports the hybrid-scheduling model, that is, task sets have preemptable tasks and non-preemptable tasks. We propose a time analysis technology and an implementation synthesis method with the ability of design space exploration and optimization. Experimental evaluation shows our implementation synthesis method yields real-time embedded software with lower system overheads.

1 Introduction

Implementation synthesis of embedded software, as a part of embedded software integration, refers to the process from logical software design models (abbr. design models) to implementation models on specific platforms. Most non-functional requirements, such as real-time, memory consumption, and low power, are implemented and optimized during the implementation synthesis of embedded software, which makes it an important stage in the design of embedded systems.

Currently, almost all the research work [4, 7, 8] is based on the assumption that the underlying operating system (OS) uses the priority-based fully preemptive scheduling strategy. Although this kind of scheduling strategy is widely used in currently commercial real-time OS, there are also other scheduling strategies that can achieve better effects in some specific domains. Hybrid scheduling is such an example. It mixes preemptable tasks with non-preemptable ones. This kind of scheduling strategy makes sense when the execution time of a task is in the same magnitude of the time of context switches, RAM is required to use economically or the execution of a task must not be interrupted. It is one of the scheduling modes supported by the specification of OSEK/VXD [9], a wildly accepted specification in automotive electronic industries.

Under the hybrid scheduling strategy, the implementation synthesis involves not only priority assignments to tasks, but also scheduling property (preemptable or non-preemptable) assignments to tasks. In this paper, we focus on the implementation synthesis of real-time embedded software running on uniprocessor with the goal of satisfying real-time and reducing system overheads. We propose the time analysis technology and a new implementation synthesis method, which is an extension of Gu et al.'s work [7], to address the implementation synthesis problem under OSs supporting the hybrid-scheduling model.

The rest of this paper is organized as follows. Section 2 presents the software models and implementation strategy. Section 3 describes the time analysis technology under the hybrid-scheduling model. We describe the process of implementation synthesis in section 4. The experimental evaluation results are given in section 5. Finally, we give conclusions and future work with section 6.

2 Software Models and Implementation Strategy

In terms of the model presented by Wang et al. [1, 2], a component is a logical software entity that can carry out certain functions triggered by events (we do not differentiate between the term "event" and the term "message" in the following sections, and use them interchangeably). An action is defined as the computation performed by a component when receiving an event. A transaction is a sequence of actions that are triggered by an external input event, possibly cut through one or more components. The components in a transaction communicate through buffered asynchronous messages. For simplicity, we only consider the *or* relation when more than one input event trigger the same output event, that is, a component can issue the output event once it receives any event from its input ports.

In design models, one action of a component has the worst-case execution time (WCET), and a transaction has a fixed period and a fixed deadline. In this paper, we assume the deadline of a transaction is no more than its period.

We use transaction-based runtime models. It is the counterpart of transactions in design modes, which consisting of a sequence of related tasks (the transactions in design modes and the transactions in runtime models can be differentiated according to their context). Each task has a period and an end-to-end (e2e) deadline. After being created and initialized, a task waits for events. When an event arrives, the task does the corresponding computation and sends one or more messages to other tasks. And then it goes back to wait for another event. Since tasks may use some shareable resources, it is necessary to synchronize the access to mutually exclusive resources.

During implementation synthesis, we chose *Component-Based Multi-Threading* (CBMT) strategy, where a thread consists of one or more components. It has the benefits of reasonable context-switching overheads, sufficient parallelism, optimal memory consumption, and better support for software engineering [7].

3 Time Analysis for CBMT under the Hybrid Scheduling Model

For the time analysis of CBMT, Gu et al. [5, 7] used the modified form of the time analysis algorithm presented by Harbour, Klein, Lehoczky [10] (They call it the HKL algorithm.). The method presented by Gu et al. is suitable for the OSs supporting the priority-based preemptive scheduling model, but not suitable for the OSs supporting the hybrid scheduling model.

The task model used in HKL algorithm assumes that a task consists of one or more subtasks. For example, a task τ_i consists of n subtasks, $(\tau_{(i,1)},\tau_{(i,2)},\ldots,\tau_{(i,n)})$. $P_{(i,j)}$ refers to the priority of the subtask $\tau_{(i,j)}$. $P_{\min(i)}$ refers to the minimum priority of all the subtasks of τ_i. When $P_{\min(m)}>P_{(i,j)}$, τ_m has *multiply preemptive effect* on $\tau_{(i,j)}$. If $\exists k,(P_{(m,1)},\cdots,P_{(m,k)}>P_{(i,j)})\wedge(P_{(m,k+1)}<P_{(i,j)})$, τ_m has *singly preemptive effect* on $\tau_{(i,j)}$. τ_m has *blocking effect* on $\tau_{(i,j)}$ if $\exists k,l,(P_{(m,l)}<P_{(i,j)})\wedge((P_{(m,l+1)},\cdots,P_{(m,k)})>P_{(i,j)})\wedge(P_{(m,k+1)}<P_{(i,j)})$. If the priority of several continuous subtasks of τ_m is higher than $P_{\min(i)}$, they are called an H segment; If the priority of several continuous subtasks of τ_m is lower than $P_{\min(i)}$, they are called an L segment.

The canonical form of a task τ_i is a task τ_i' whose subtasks maintain the same order, but with priority levels that do not decrease. Harbour et al. proved that the completion time of τ_i was equal to that of τ_i'. When we calculate the worst case response time (WCRT) of τ_i, first, transform task τ_i into its canonical form, τ_i', denoted as $(\tau_{(i,1)}', \tau_{(i,2)}', \cdots,\tau_{(i,m)}')$, then analyze each subtask's completion time in τ_i' one by one. The completion time of the last subtask of τ_i' is equal to the completion time of τ_i'.

In the HKL algorithm, $\mathbf{C_{(i,k)}}$ is the WCET of $\tau_{(i,k)}'$; $\mathbf{MP_{(i,k)}}$ is the tasks that have multiply preemptive effect (type-1 tasks) on $\tau_{(i,k)}'$; $\mathbf{SP_{(i,k)}}$ is the tasks that has singly preemptive effect (type-2 tasks) on $\tau_{(i,k)}'$; and $\mathbf{B_{(i,k)}}$ is the blocking time suffered by $\tau_{(i,k)}'$.

The HKL algorithm works well under the four assumptions given by Harbour et al. But the HKL algorithm does not consider context-switching overheads and blocking time caused by resource sharing and other factors. In the design models of this paper, the total blocking time comes from three aspects: blocking time caused by high priority task, which has been discussed in the HKL algorithm, blocking time caused by sharing resources that must be accessed serially, and blocking time caused by sharing components among multiple transactions. Gu et al. considered the influence caused by sharing components. However, the blocking time they discussed is only suitable for the situation that different input messages trigger different output messages. Moreover, context-switching overheads in Gu et al.'s work does not consider an H/L segment may include more than one tasks. In the following, we extend the HKL algorithm for performing time analysis on CBMT.

In design models, there are three typical relations: (1) One input message triggers an action sequence. The action sequence outputs messages to multiple components, as shown in Fig. 1 (a). We call this kind of sharing relation *1-M sharing*; (2) Any of multiple messages can trigger a sequence of actions, as shown in Fig. 1 (b). We call this kind of sharing relation *M-1 sharing;* and (3) Multiple different messages trigger multiple different action sequences on multiple sharing components, as shown in Fig. 1 (c). We call this kind of sharing relation *M-M sharing.* In comparison to the task model used in HKL algorithm, the time analysis for CBMT is more complex due to the above three cases. The *M-M sharing* has the same influence on the time analysis with *M-1 sharing*, thus we only research the other two component sharing relations.

In order to perform time analysis for design modes, we regard components as scheduling entities. We still use the notion of subtask to denote the tasks in transactions. We use τ_{Ci} to denote the subtask that a component C_i belongs to, and use $CO_{\tau(i,j)}$ to denote the component that a subtask $\tau_{(i,j)}$ consists of. In Fig. 1 (a), assuming that Tr_1-Tr_q are transactions that M_{o1}-M_{oq} belong to.

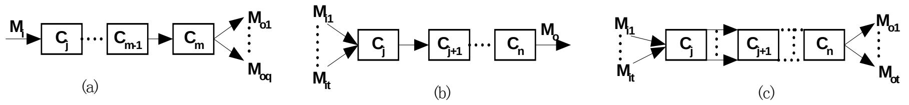

Fig. 1. Component-sharing relations (a) *1-M* sharing ($m \geq j$) (b) *M-1* sharing ($n \geq j$) (c) *M-M* sharing ($n \geq j$)

1-M sharing has the following influence on the execution time of subtasks:

- Among the transactions that share multiple common components, one transaction does not preempt or block the other transactions.
- The preemption time (including multiple preemption time and single preemption time) caused by the subtasks that consist of sharing components should only be calculated in one transaction.

In the following discussion of *M-1 sharing*, we use the example shown in Fig. 1 (b), and assume Tr_1-Tr_t are transactions that M_{i1}-M_{it} belong to. For an randomly selected transaction Tr_i from Tr_1-Tr_t, its canonical form is a transaction with subtask sequence $(\tau_{(i,1)}',\cdots,\tau_{Cj}',\cdots,\tau_{Ck}'\cdots,\tau_{Cn}'\cdots)$. τ_{Ck}' is a subtask among τ_{Cj}'-τ_{Cn}'.

Under the relation of *M-1 sharing*, a subtask consisting of a shared component cannot be preempted or blocked by succeeding subtasks. For example, in Fig. 1 (b), τ_{Cj} cannot be preempted or blocked by τ_{Ck}, where $\tau_{Ck} \in \{\tau_{Cr} \mid j < r \leq n\}$. There are two cases in the calculation of the WCRT of subtasks:

- For subtasks before τ_{Cj}, their WCRT can be calculated by using the HKL algorithm.
- For subtasks τ_{Cj}'-τ_{Cn}', the preemption time and the blocking time caused by other transactions in Tr1-Trt can be calculated by using the following equation (1).

The preemption effect of a transaction only occurs once under the condition of *M-1 sharing* because multiple transactions use the same components. So we regard it as blocking effects. The blocking time suffered by $\tau_{(i,j)}'$ because of *M-1 sharing* is denoted as:

$$B_{CM1}^{(i,j)} = \sum_{Tr_p \in PE(i,j)) \wedge (CO_{\tau(p,q)} = CO_{\tau(i,j)'})} CET(PC_{(p,i,j)}) \qquad (1)$$

Where $PC_{(p,i,j)}$ denotes the components before $CO_{\tau(i,j)'}$ in transaction Tr_p that exhibits preemption effects on $\tau_{(i,j)}'$, $CET(PC_{(p,i,j)})$ denotes the sum of the WCET of all components in $PC_{(p,i,j)}$. $PE_{(i,j)}$ denotes the transactions that have preemption effects (including multiple preemption effects and single preemption effects) on $\tau_{(i,j)}'$ in Tr_1-Tr_k except Tr.

Under the hybrid scheduling, if a component appears in an L segment, the L segment should be divided into three segments: an L segment, an H segments, and an L segment. Harbour et al. have discussed this problem in [10].

Considering the blocking effects caused by component sharing, we modify the $MP_{(i,j)}$ to $MP_{(i,j)}'$, $SP_{(i,j)}$ to $SP_{(i,j)}'$. They are denoted as:

$$MP^{'}_{(i,j)} = MP_{(i,j)} - \{Tr_u \mid \exists m, CO_{\tau_{(i,j)'}} = CO_{\tau_{(u,m)}}\}$$
$$SP^{'}_{(i,j)} = SP_{(i,j)} - \{Tr_u \mid \exists m, CO_{\tau_{(i,j)'}} = CO_{\tau_{(u,m)}}\}$$

Except the blocking time $B_{(i,j)}$ which is caused by inner H segments, the blocking time suffered by the subtask $\tau_{(i,j)}'$ consists of three aspects: the blocking time $B^{(i,j)}_{CM1}$, which is caused by *M-1 sharing*; the blocking time $B^{(i,j)}_{CMM}$, which is caused by *M-M sharing*, and can be calculated using similarly method in equation (1); and the blocking time $B^{(i,j)}_{O}$, which is caused by sharing resources and has been discussed by Gu et al. in [7]. It is denoted as:

$$B^{*}_{(i,j)} = B^{(i,j)}_{O} + B^{(i,j)}_{CM1} + B^{(i,j)}_{CMM}$$

The context-switching time is mainly caused by multiply preemptive tasks, so we only consider this kind of time overheads. The context-switching time suffered by $\tau_{(i,j)}'$ because of $Tr_p \in MP_{(i,j)}$ is:

$$CS_{(i,j)} = 2N_p \cdot CS_{sys}$$

Where N_p is the subtask number in Tr_p, CS_{sys} is the context-switching overheads.

If $\tau_{(i,j)}'$ is a preemptable subtask, its completion time is:

$$t_{(i,1)} = B^{*}_{(i,1)} + B_{(i,1)} + \sum_{Tr_p \in MP^{'}_{(i,1)}} \left\lceil \frac{t_{(i,1)}}{T_p} \right\rceil (C_p + CS_{(i,1)}) + \sum_{Tr_p \in SP^{'}_{(i,1)}} C^h_p + C_{(i,1)}$$
$$t_{(i,j)} = t_{(i,j-1)} + B^{*}_{(i,j)} + \sum_{Tr_p \in MP^{'}_{(i,j)}} \left[\left\lceil t_{(i,j)} / \mathrm{T}_p \right\rceil - \left\lceil t_{(i,j-1)} / \mathrm{T}_p \right\rceil \right] (C_p + CS_{(i,j)})$$
$$+ \sum_{Tr_p \in SP^{'}_{(i,j)}} \min(1, \left[\left\lceil t_{(i,j)} / \mathrm{T}_p \right\rceil - \left\lceil t_{(i,j-1)} / \mathrm{T}_p \right\rceil \right]) C^h_p + C_{(i,j)} \quad when \quad j > 1 \quad (2)$$

Where $C_{(i,j)}$ is the WCET of $\tau_{(i,j)}'$, T_p is the period of Tr_p, C_p is the WCET of Tr_p, and C^h_p is the WCET of the first H segment of Tr_p. Under the condition of *1-M sharing*, the WCET of sharing components only be calculated in one transaction.

According to algorithm proposed by Wang et al. [6], the response time of a non-preemptable task consists of waiting time $W_{(i,j)}$ and the execution time $C_{(i,j)}$. However, the algorithm reported by Wang et al is based on independent tasks. For a non-preemptable subtask $\tau_{(i,j)}'$, its completion time is:

$$t_{(i,j)} = t_{(i,j-1)} + W_{(i,j)} + C_{(i,j)}$$
$$W_{(i,1)} = B^{*}_{(i,1)} + B_{(i,j)} + \sum_{Tr_p \in MP^{'}_{(i,1)}} (1 + \left\lceil \frac{W_{(i,1)}}{T_p} \right\rceil)(C_p + CS_{(i,1)}) + \sum_{Tr_p \in SP^{'}_{(i,1)}} C^h_p$$
$$W_{(i,j)} = B^{*}_{(i,j)} + \sum_{Tr_p \in MP^{'}_{(i,j)}} \left[\left\lceil (W_{(i,j)} + WT_{(i,j-1)}) / \mathrm{T}_p \right\rceil - \left\lceil WT_{(i,j-1)} / \mathrm{T}_p \right\rceil \right] \cdot (C_p + CS_{(i,j)})$$
$$+ \sum_{Tr_p \in SP^{'}_{(i,j)}} \min(1, \left[\left\lceil (W_{(i,j)} + WT_{(i,j-1)}) / \mathrm{T}_p \right\rceil - \left\lceil WT_{(i,j-1)} / \mathrm{T}_p \right\rceil \right]) C^h_p$$
$$where \quad WT_{(i,j)} = \sum_{k=1}^{j} W_{(i,k)}. \quad when \quad j > 1 \qquad (3)$$

4 The Process of Implementation Synthesis

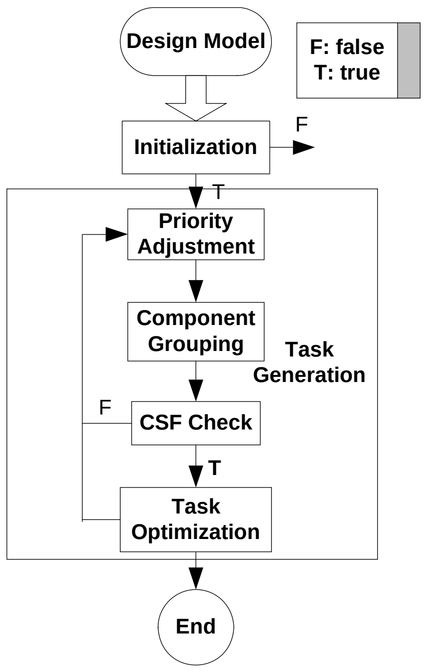

Fig. 2. The process of implementation synthesis

Implementation synthesis includes initialization and task generation, as shown in Fig. 2. In this paper, we choose SA to perform task generation because of its effectiveness in dealing with discrete optimization problem. Our optimization objective is to increase Critical Scaling Factor (CSF) [3] and reduce system overheads. CSF is an index that is used to evaluate time sensitivity of a task set. The bigger the CSF of a system is, the more stable a system is. The energy function is defined as:

$$E_{(S)} = -CSF + \frac{1}{10 \cdot N_c} \cdot N_{sys}$$

Where N_c denotes the number of components in the design model, N_{sys} is the total times of context switches when each task completes its once execution. When the generated task set has the expected $E_{(s)}$ and no change occurs during some steps (we set it 1000), the SA ends successfully.

4.1 Initialization

The purpose of initialization is to abstract all transactions in a design model and assign components' initial priorities. It is include two steps:

First, transaction set generation. We use the method proposed in [4] to find transactions in the design model.

Second, assign initial priorities of components. We follow the rules: (a) In a transaction, the nearer a component is from the event source, the higher its priority is; and (b) The longer the period of a transaction is, the higher the priority levels of the components that it includes is. If two transactions have the same period, the transaction with longer execution time (the sum of WCET of all its components) is assigned the higher priorities. It should be pointed out that a component shared by multiple transactions is assigned the same priority as the shortest-period transaction using this component.

4.2 TASK Generation

The purpose of task generation is to find a task set with acceptable CSF. It includes three steps.

Priority Adjustment. Randomly select two components and exchange their priorities to find other priority assignment schemes in the search space of SA.

Component Grouping. The objective of component grouping is to merge adjacent components with consecutive priorities and organize them into tasks. It works as follows: Following through the component sequence of a transaction, if a component with multiple output messages or its next component has multiple input messages, the components with adjacent priorities are merged into tasks. Then continue to find other component in this transaction. For example, there are eight components with priority sequence (3, 5, 4, 6, 8, 7, 10, 11), where the first six components are preemptable and the last two components are non-preemptable. They can be merged into two tasks. The first task consists of the first six components with priorities 7 and it is preemptable. The second task consists of the last two components with priorities 11 and it is non-preemptable.

CSF check. The completion time of a transaction can be obtained using the equation (2) and equation (3). From the completion time and deadlines of all transactions, the CSF of the system can be obtained. If CSF is acceptable, we continue the next step. Otherwise, go to the step of priority adjustment.

Task optimization. Its purpose is to reduce the number of tasks and the context-switching time by merging adjacent tasks and adjusting their preemption properties.

We use APT_{sys} denote the average times that all tasks are preempted by other tasks in transaction set, APT_i denoting the average times that a task τ_i is preempted by other tasks in transaction set.

The algorithm of task optimization is shown in **Algorithm TP.**

Algorithm TP(TS)

```
  /* TS is the set of tasks. $E_{(s)}$ denotes the energy of
the transaction set. $CSF_{acc}$ is the threshold that judges
whether CSF is acceptable. */
m = The transaction number in TS;
```

```
for ( j=0; j<m; j++){
  Tr = The j^th transaction in TS;
  n= The number of tasks in Tr;
  for (i=0; i<n; i++) {
    Ti = The i^th task in Tr;
    if ($APT_i$ > $APT_{sys}$ && Ti has preemptable property){
     Make Ti a non-preemptable task;
     Calculate the current $E_{(s)}$;
    if ($E_{(s)}$ increase || CSF < $CSF_{acc}$)
      Make Ti a preemptable task;
    }
  }
  for ( j=0; j<m; j++){
     Tr = The j^th transaction in TS;
     n= The number of tasks in Tr;
    for (i=0; i<n; i++) {
      Ti= The i^th task in Ts;
      if (Ti and T(i-1) have different preemptive
          properties) continue;
      if ($APT_i$ < $APT_{sys}$ && $APT_{i-1}$ < $APT_{sys}$) {
        Merge Ti and T(i-1) into a tasks Ti', and use the
        higher priority between Pi and P(i-1) as the
        merged task's priority;
       Calculate the current $E_{(s)}$;
       }
       if ($E_{(s)}$ increase)
         Break Ti' up into the original Ti and T(i-1);
      }
    }
 }
```

Algorithm TP searches all tasks in transactions, and try to assign preemptable property to tasks that have more preempted times in order to reduce context-switching overheads. In this process, CSF should be no less than the expected value and E(s) does not increase. After that, merge adjacent tasks to reduce the number of tasks.

5 Experimental Evaluation

In order to evaluate the performance of the algorithm presented in this paper, we constructed the design model with the following parameters: transaction number: 40; the number of components in each transaction: 5-10; period of transactions: 10-2000ms; the WCET of component: 1-25ms; context-switching time: 8 microseconds. There were *1-M*, *M-M* and *M-1* sharing among components. There were sharing resources among components. All components in the design model were preemptable and left a large freedom degree for priority assignment. A fixed CSF (1.15) was chosen in the experiments. The experiments were performed on a PC running Windows XP, with 2500MHz CPU speed and 512Mb memory.

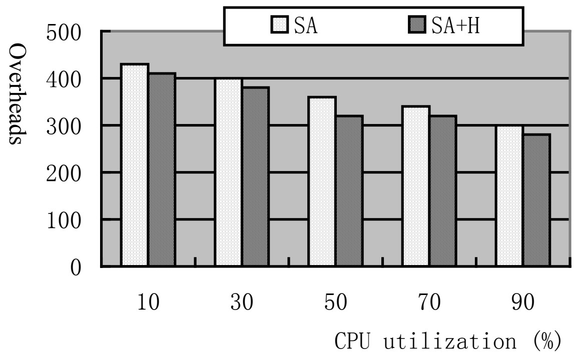

Fig. 3. The comparison of system overheads

In order to obtain the reduction of context-switching time, we compared the method proposed in this paper, named SA+H, to the method only using SA, named SA, under the same condition, but it did not assign preemption properties to tasks in SA. We counted the context-switching times when every task completed once execution. The experimental result is shown in Fig. 3. In comparison to SA, the context-switching times of SA+H are less than those of SA at different CPU utilization. The difference of context-switching times between SA and SA+H is maximal when the CPU utilization is 50 percent. This is because they are both effective when CPU utilization is low. But SA+H shows its advantage when the CPU utilization is moderate because the more accurate time analysis algorithm in section 3.1 makes non-preemptable tasks generated effectively. The difference of context-switching times between SA and SA+H becomes little as the increase of CPU utilization. It is because the non-preemptable properties of some tasks influence task merging. Non-preemptable tasks lead to a reduction of 10 percent in context-switching time in SA+H (not shown in Fig. 3). Although the non-preemptable tasks contribute a small part in total context-switching time, it is significant to assign non-preemptable properties to a task to make its execution uninterrupted in some safe-critical applications.

We investigated the distribution of the non-preemptable tasks in a task set generated with the proposed method in this paper. The experiment was performed with CPU utilization of 70%. The experimental result shows the non-preemptable tasks occur more frequently in the tasks with shorter WCET. It is because tasks with larger WCET have greater possibility to destroy the shedulability of system than tasks with smaller WCET. It also confirms the fact that hybrid-scheduling property is more suitable for tasks with smaller WCET and tasks that cannot interrupt for special reasons.

6 Conclusions and Future Work

This paper proposes a new method for the implementation synthesis of embedded software under OS that supports the hybrid-scheduling model. It is an extension of implementation synthesis method that is based on priority-based fully preemptive scheduling and enlarges the application range of implementation synthesis technology. We propose a time analysis method for CBMT and an implementation synthesis method with the design space exploration and optimization ability. This method can yield real-time implementation and has lower system overheads. The work focusing on the influence of other resource constraints, such as memory, energy, on implementation synthesis, is on the way.

Acknowledgments

This research was supported by 863 National High Technology Program under Grant No. 2005AA1Z2020 and Key Program of Science and Technology Department of Zhejiang Province under Grant No. 2004C21057.

References

1 Wang, S., Merrick, J. R., and Shin, K. G.: Component allocation with multiple resource constraints for embedded real-time software design. In proc. IEEE Real-Time and Embedded Technology and Applications Symposium. (2004) 219-226
2. Wang, S., and Shin, K. G.: An architecture for embedded software integration using reusable components. In proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems. (2000) 110-118
3. Vestal, S.: Fixed-priority sensitivity analysis for linear compute time models. IEEE Trans. Software Eng., vol. 20. (1994) 308-317
4. Kodase, S., Wang, S., and Shin, K. G.: Transforming structural model to runtime model of embedded software with real-time constraints. In Proc. Design, Automation and Test in Europe Conference . (2003) 20170-20175
5. Gu, Z., Wang, S., and. Shin, K. G.: Synthesis of real-time implementation from UML-RT models. In Proc. IEEE RTAS Workshop on Model-Driven Embedded Systems. (2004)
6. Wang, L., and Wu, Z.: Schedulability Test for Fault-Tolerant Hybrid Real-time Systems with Preemptive and Non-preemptive tasks. In Proc. the Fourth International Conference on Computer and Information Technology. (2004) 1169-1174
7. Gu, Z., and Shin, K. G.: Synthesis of Real-Time Implementations from Component-Based Software Models. In Proc. IEEE Real-Time Systems Symposium. (2005)
8. Bartolini, C., Lipari, G., and Natale, M. D.: From functional blocks to the synthesis of the architectural model in embedded real-time applications. In Proc. IEEE Real Time and Embedded Technology and Applications Symposium. (2005) 458-467
9. OSEK/VDX Operating System, Version 2.2.1, Jan. 16, 2003. [Online]. Available: http://www.osek-vdx.org/mirror/os221.pdf
10. Harbour, M., Klein, M. H., and Lehoczky, J.: Timing analysis for fixed-priority scheduling of hard real-time systems. IEEE Trans. Software Eng., vol. 20, no. 2. (1994) 13-28