Synthesis of Real-Time Implementations from Component-Based Software Models

Zonghua Gu
Hong Kong University of Science and Technology
Kowloon, Hong Kong, China
zgu@cs.ust.hk

Kang G. Shin
The University of Michigan
Ann Arbor, MI 48109-2122, U.S.A.
kgshin@umich.edu

Abstract

Component-based software development is an effective technique for tackling the increasing complexity of large-scale embedded software systems. After building a logical software model, the designer must make design decisions, including choosing a multi-threading strategy and assigning priorities to threads, to ensure that the final implementation on the target execution platform satisfies non-functional requirements. Code generators for software design tools produce functional code, but typically ignore concurrency and timing issues. In this paper, we describe techniques for real-time scheduling and design-space exploration and optimization, with the goal of helping the designer synthesize efficient real-time implementations from component-based software models. Experimental evaluation shows that our techniques yield high-quality implementations with reasonable running time of the optimization algorithm.

1 Introduction

Component-Based Software Development (CBSD) is gaining popularity as an effective technique for tackling the increasing complexity of large-scale *Real-Time Embedded* (RTE) software systems. Most companies have some type of in-house software component libraries. The CBSD process can be divided into two stages: *component development*, where a domain expert develops software components, and *component integration*, where a system integrator picks components from the component library and assembles them into a complete application. Each component should be developed once and reused many times. Our research focuses on the component integration stage, since it is the stage that presents the most number of issues and challenges in the software development lifecycle.

Even though CBSD has been around for a long time and widely applied in the enterprise computing arena, it has met with limited success in the RTE domain due to unique challenges caused by resource constraints and non-functional requirements such as real-time, low-power, fault-tolerance, etc. In this paper, we focus on a class of *Component-Based Event-Driven* (CBED) software models with interaction style of buffered asynchronous message passing between components with ports. This programming style is well-suited for large-scale distributed RTE systems. It offers a number of benefits from a software engineering perspective, such as modularity, encapsulation and decoupling of interactions, and hence, it is scalable to larger systems than the time-triggered, synchronous approach, which often have much smaller size due to strict requirements for clock synchronization. One representative example that follows this interaction style is UML-RT, a UML Profile based on Real-Time Object-Oriented Modeling [1] and supported by CASE Tools from IBM Rational. The main concepts of UML-RT have been incorporated into the new UML 2.0 standard. Other examples include the Specification and Description Language (SDL) [2], a popular design language in the telecommunications domain, and the Quantum Framework [3], a programming discipline that achieves the same goals as UML-RT without expensive CASE tools.

This class of software systems typically has enormous size and complexity. They often adopt object-oriented design patterns and abstractions that make code-based analysis difficult, and necessitates building and analyzing higher-level models at early design stages. The asynchronous, event-driven interaction style makes it difficult to guarantee system timing properties. In this paper, we address the *implementation synthesis* problem: given a logical software design model, how do we synthesize a multi-threaded implementation that runs on a target hardware platform and satisfies timing constraints? Commercial code generators typically yield functional code, but largely ignore *para-functional requirements* [4] such as real-time, low-power, fault-tolerance, etc. It is up to the designer to make design decisions such as allocating components to processors, choosing a multi-threading strategy, and assigning priorities to threads, etc., to ensure that the final implementation on the target platform satisfies these para-functional requirements. We focus on *real-time requirements* in this paper, and leave the other requirements to future work. We first compare alternative multi-threading strategies from CBED software models, and then describe real-time scheduling and design-space exploration techniques for implementation synthesis. We use the *Critical Scaling Factor* (CSF) [5, 6] as our optimization objective, which is defined as the largest coefficient by which Worst-Case Execution Time (WCET) of all threads can be simultaneously multiplied while preserving feasibility. For example, if a system has CSF of 1.17, then if we multiply the WCET of all threads by a number $n \leq 1.17$, the system would still be schedulable. However, any $n > 1.17$ would render the system unschedulable. It is desirable to adopt scheduling algorithms and attribute assignments to maximize the CSF. A system with larger CSF is more robust to timing faults caused by inaccuracies in WCET estimation or transient runtime overload. It also has more room to grow from the perspective of system evolution and upgrading. A system with CSF not much larger than 1 is "barely feasible", and a minor perturbation can cause it to miss deadlines [5]. A system with CSF less than 1 is not schedulable, and WCET of all tasks needs to be scaled down by a factor of CSF in order to make it schedulable.

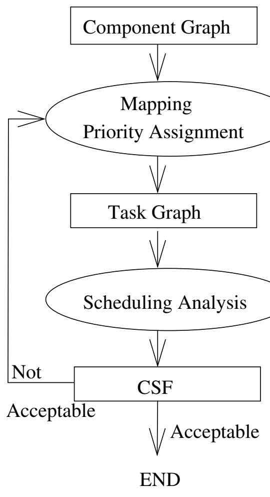

Figure 1. The workflow for implementation synthesis.

As shown in Figure 1, we start from the component dependency graph, and perform component-to-thread mapping and priority assignment to get a task graph. We then perform scheduling analysis on the task graph to get the CSF. If it is acceptable, then the workflow ends; otherwise, this process is repeated. The iterative optimization algorithm is based on Simulated Annealing (SA), and uses the real-time scheduling algorithm as a subroutine.

The rest of this paper is structured as follows. Section 2 discusses and compares different multi-threading strategies. Section 3 describes real-time scheduling analysis techniques for the component-based multi-threading strategy. Section 4 presents application of simulated annealing for design-space exploration and optimization. Section 5 experimentally evaluates our techniques. Section 6 discusses related work, and Section 7 draws conclusions and discusses future research directions.

2 Multi-Threading Strategies

We first introduce some definitions.

- A *component* is a logical entity that provides one or more *event handlers* that can be triggered by external events.
- An *end-to-end scenario* is an ordered sequence of event handlers with an end-to-end (e2e) deadline. A scenario typically spans multiple components, and may traverse one component multiple times, each time invoking a possibly different event handler.

It is important to differentiate between the concept of design-level concurrency and that of implementation-level concurrency [7]. At the design level, each component conceptually contains its own logical thread that handles incoming events, but each logical thread is not necessarily mapped into an OS thread at the implementation level. Although it is possible for each component to have its own OS thread, it may incur too much context-switching overhead if there is a large number of components. There are a number of possible multi-threading strategies at the implementation-level, as discussed below.

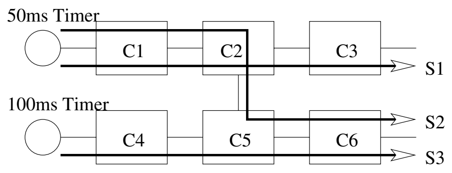

Figure 2. An example application with 6 components C1 to C6, and three e2e application scenarios, denoted by solid arrows.

Suppose we have an application as shown in Figure 2, consisting of 6 components and 3 e2e scenarios. Each scenario consists of multiple *subtasks*, which are triggered actions executed by the components. Given this logical model, how do we implement it on a multi-threaded real-time OS? We consider two alternatives: *Scenario-Based Multi-Threading* (SBMT), where each application scenario is mapped into a separate thread with uniform priority, and *Component-Based Multi-threading* (CBMT), where one or more components are grouped into a thread with uniform priority.

Saksena *et al.* [7] and Kim *et al.* [8] advocated an approach that we call *Component-Based Multi-threading, Scenario-Based Priority-Assignment*, where one or more components are grouped into the same thread, but priorities are associated with the e2e scenarios, and the thread priorities are adjusted dynamically to reflect the priority of the currently-executing e2e scenario, in order to maintain a uniform priority across each e2e scenario. This approach involves runtime system-call overheads that may or may not be acceptable to certain resource-constrained embedded systems. Certain small real-time operating systems may not even provide APIs to dynamically change thread priorities. From a real-time scheduling perspective, this approach is similar to SBMT except for overhead associated with thread context switching. Therefore, we classify this approach as the same category as SBMT, and hence do not consider it separately.

For the CBMT approach, the number of threads needs to be managed carefully. If there are too few threads, the blocking time may be too much due to insufficient parallelism; on the other hand, if there are too many threads, the context-switching overheads may be excessive. Two extreme cases are *thread-per-system* and *thread-per-component*. The thread-per-system approach is to have a single thread of execution for the entire system. It is generally not suitable for real-time systems, since it eliminates all concurrency and turns the system into a sequential program. The thread-per-component approach assigns each component its own thread, which results in high CPU and memory runtime overheads if there is a large number of components. In most situations, we would like to adopt an intermediate configuration that achieves sufficient parallelism without incurring too much runtime overhead.

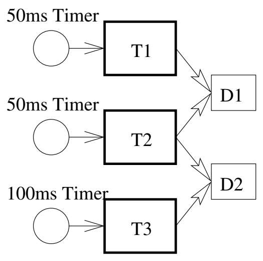

Figure 3. Map each e2e scenario into its own thread.

Figures 4, 5 and 6 show some of the many possibilities for grouping components into threads. If we adopt SBMT, then we get a runtime model as shown in Figure 3, with three periodically-triggered threads accessing shared data. Since multiple scenarios cut through the same component, it is necessary to use mutual exclusion mechanisms such as mutex and monitor to protect shared data. In this case, the designer does not have any choice for component-to-thread mapping, which is deterministic. However, he does have some freedom for priority assignment. Priorities are usually, but not necessarily, assigned rate-monotonically, i.e., a thread with faster execution rate will be assigned a higher priority. But for CBMT, there is a large design space for assigning priorities to threads, since there is a large number of design choices in terms of component-to-thread mapping and priority assignment.

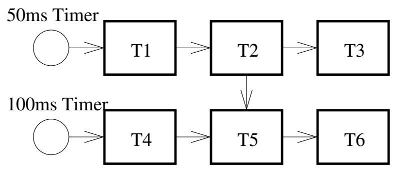

Figure 4. Map each software component into its own thread.

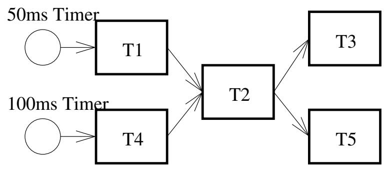

Figure 5. Map C2 and C5 into the same thread T2.

Even though it is possible to adopt a dynamic priority scheduling algorithm such as Earliest Deadline First (EDF), we focus on static priority scheduling, since it is the most widely adopted scheduling algorithm in commercial real-time operating systems, and has a number of advantages over EDF, including low runtime overhead, overload protection of high-priority tasks, etc. For fixed priority scheduling, *Rate Monotonic Analysis* (RMA) [9] is a well-known technique for analyzing the schedulability of a set of real-time tasks or threads (we use the words task and thread interchangeably in this paper). One of the assumptions of RMA is that each task must have a static, uniform priority value. For SBMT, this assumption is satisfied, since each e2e scenario can be viewed as a task with uniform priority. But it is not satisfied

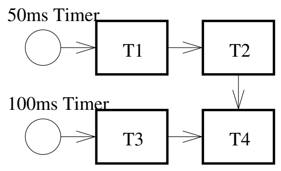

Figure 6. Map C2 and C3 into the same thread T2, and C5 and C6 into the same thread T4.

for CBMT, since each e2e task consists of multiple subtasks of varying priority. We present real-time scheduling analysis techniques for CBMT in Section 3.

Aside from the real-time scheduling perspective, CBMT has certain advantages over SBMT from a software engineering perspective, such as modularity, encapsulation, decoupling of interactions, mature tool support, etc. SBMT creates shared data and necessitates error-prone concurrency control mechanisms. This destroys a key advantage of the component-based, event-driven interaction style, which is to use buffered asynchronous message passing as the primary communication mechanism among components instead of shared data in order to minimize the need for concurrency control. CBMT is the default runtime model implemented in CASE tools, e.g., that from IBM Rational, which provides options for creating multiple threads, each containing one or more components and assigned a fixed priority. SBMT is generally not directly supported by commercial CASE tools, but requires manual customizations [8]. Table 1 summarizes key differences between SBMT and CBMT.

	SBMT	CBMT
RMA Applicable	yes	no
Direct CASE Tool Support	no	yes
Design Space	small	large

Table 1. Comparison between SBMT and CBMT.

3 Scheduling Analysis for CBMT

Consider a software model consisting of m components $O_1, O_2, \ldots, O_m$, and n e2e scenarios, where each scenario is mapped into an *e2e virtual thread*, forming the task set $\tau_1, \tau_2, \ldots, \tau_n$. Here we use the word *virtual* to denote the fact that each e2e scenario may consist of multiple subtasks distributed over different OS threads. Each e2e scenario $\tau_i, i = 1, \ldots, n$ cuts through one or more components, and triggers an action within each component, forming a chain of subtasks $\tau_{i1}, \ldots, \tau_{im(i)}$. We use $O(\tau_{ij})$ to denote the component that the subtask τ_{ij} belongs to, and $PO(\tau_{ij})$ to denote the (possibly multiple) passive objects that τ_{ij} accesses. Each subtask τ_{ij} is actually an event-triggered action within a component $O(\tau_{ij})$, and is characterized by parameters (C_{ij}, P_{ij}), where C_{ij} is its worst-case execution time, and P_{ij} is its priority. Each e2e thread τ_i has an e2e deadline D_i.

The task model is similar to the task model of e2e threads with subtasks with varying priority, as described by Harbour, Klein, Lehoczky in [10]. We call the scheduling analysis algorithm introduced in [10] the *HKL algorithm.* However, in order to be applicable to CMBT, the HKL algorithm needs to be adapted to consider the blocking time caused by multiple subtasks sharing common components and the *Run-To-Completion* (RTC) semantics, which states that, once triggered by a message at its input port, the component must execute the triggered action to completion before processing the next message. RTC is useful for reducing the number of concurrency bugs when a component can take part in multiple e2e scenarios. Messages can be assigned priorities and queued in priority order instead of FIFO order. Each OS thread processes incoming messages for the components assigned to it in a priority-based, non-preemptive manner, consistent with the RTC semantics. However, there can be preemptions between different threads in a multi-threaded system, that is, a component executing in the context of a higher-priority thread can preempt another component executing in a lower-priority thread.

A component may be involved in multiple subtasks within one e2e thread, or in multiple e2e threads. Due to RTC, a subtask may suffer a blocking time equal to the largest execution time of other subtasks sharing the same component. Blocking time can also be caused by sharing of passive objects among multiple e2e threads. We do not model method invocations to passive objects as separate subtasks, since a passive object can be viewed as an extension of the invoking component, and inherits the thread and priority from it. But we do need to account for the blocking time caused by sharing of passive objects.

We first briefly describe the HKL algorithm[1]. The *canonical form* of a task τ_i is a new task τ_i' with the same sequence of subtasks as τ_i, but with strictly increasing priorities. Now, we define $P_{min}(i)$ to be the minimum priority of all subtasks of τ_i. The next step is to classify all tasks $\tau_j, j \neq i$ according to their relative priority levels with respect to $P_{min}(i)$. For example, if the canonical form of τ_i consists of a single segment of priority 18, and τ_j consists of priority sequence (19, 10, 19, 10, 25, 10), then τ_j is classified as (H, L, H, L, H, L), where H stands for *higher* and L stands for *lower*.

There are five types of tasks:

- Type 1, or H^+, tasks, with all subtask priorities higher than or equal to τ_i. These tasks can preempt task τ_i multiple times.
- Type 2, or $(H^+L^+)^+$, tasks. The first subtask has higher priority than τ_i, but it can only preempt τ_i once, since it is followed by subtasks of lower priority. Multiple tasks of this type may preempt τ_i, but only for the first segment. The non-first high-priority segments cause a blocking effect.
- Type 3, or $((HL)^+H)$, tasks. They differ from type-2 tasks since they end with a high priority segment.
- Type 4, or $(L^+H^+)^+L^+$, tasks. The first subtask has lower priority than τ_i. Any one of the following

[1] One limitation of the HKL algorithm is that it can only handle linear task-chains, but not more general task trees or graphs. It is an open research issue as to how to extend the HKL algorithm to deal with task-trees or graphs.

subtask segments can have a blocking effect on τ_i, but only one such segment among all tasks of type 4 can have such a blocking effect.

- Type 5, or L^+, tasks. They have no effect on completion time of τ_i, and can be ignored for response time calculation.

Suppose we need to calculate response time of task t_i. To simplify the discussion, let's assume the canonical form of t_i consists of subtasks of uniform priority P_i. Define $H_1(i), H_2(i), H_4(i)$ to be the indices of all tasks of type 1, 2, and 4, respectively.

For each $j \in H_2(i)$, let $B_2(i,j)$ be the execution time of the first H^+ segment of task τ_j. $B_2(i,j)$ denotes the preemption time caused by τ_j to τ_i. Then, the total preemption time suffered by τ_i is:

$$B_2(i) = \sum_{j \in H_2(i)} B_2(i,j)$$

For each $j \in H_2(i) \cup H_4(i)$, let $B_4(i,j)$ be the blocking time suffered by τ_i, caused by all H^+ segments of task τ_j of type 4, and all (but the first) H^+ segments of task τ_j of type 2. Then, the total blocking time suffered by τ_i is:

$$B_4(i) = \max\{B_4(i,j) | j \in H_4(i) \cup H_2(i)\}.$$

For a type-2 task, only the first higher priority segment should be counted in $B_2(i)$, while the remaining segments should be counted in $B_4(i)$. Since multiple type-2 tasks can use their first segments to preempt t_i, $B_2(i)$ is the *sum* of $B_2(i,j)$; since only one type-2 or 3 task can use any of its segments other than the first segment to preempt t_i, $B_4(i)$ is the *maximum* of $B_4(i,j)$.

In order to adapt the HKL algorithm to CMCP, we need to take into account an additional blocking time term:
$B(i) = \max\{C_{kl} | \forall k, l, j, (k! = i) \& (O(\tau_{kl}) = O(\tau_{ij}))\}$ $+ \max\{C_{mn} | \forall m, n, j, (m! = i) \& (P_{mn} < P_{ij}, PO(\tau_{mn}) \cap PO(\tau_{ij})) \neq \phi\}$,
where the first term denotes the blocking time caused by other subtasks sharing the same component with some subtask of thread i due to the RTC semantics, and the second term represents the blocking time caused by other lower-priority subtasks accessing shared passive objects.

The equation for calculating the Worst-Case Response Time (WCRT) of task τ_i is:

$$\begin{aligned}\text{WCRT}(i) &= \text{WCET}(i) + B_2(i) + B_4(i) + B(i) \\ &+ \sum_{j \in H_1(i)} \lceil \frac{\text{WCRT}(i)}{\text{Period}(j)} \rceil \cdot (\text{WCET}(j) + 2{*}\text{CS}) \quad (1)\end{aligned}$$

where WCET(i) is the worst-case execution time of τ_i, and Period(j) is the execution period of τ_j if it is a periodic task, or the minimum inter-arrival time of execution triggers for τ_j if it is a sporadic task. The last term is preemption time caused by type-1 tasks. CS refers to the OS overhead for thread context-switching. This is a recursive equation that can be solved iteratively. τ_i is schedulable if the calculated WCRT(i) is less than its deadline. Given this equation for calculating WCRT, the CSF can be easily calculated using the techniques described in [6].

3.1 The Elevator Control Application

We use the single-processor elevator control application taken from [11] as a example to illustrate our scheduling analysis techniques. As shown in Figure 7, the system consists of 8 components and 1 passive data object. (Components are drawn with thick borders, and shared data objects are drawn with thin borders.) There are three e2e scenarios:

1. **Stop Elevator at Floor**: The elevator is equipped with arrival sensors that trigger an interrupt to the component *arrival sensors interface* when the elevator approaches a floor, which, in turn, sends a message *approaching floor* to the component *elevator controller*. The *elevator controller* invokes a synchronous method call on the passive data object *elevator status and plan* to determine whether the elevator should stop or not.

2. **Select Destination**: The user presses a button in the elevator to choose his destination, which triggers an interrupt to the component *elevator buttons interface*, which in turn sends a message *elevator request* to the component *elevator manager*. The *elevator manager* receives the message and records destination in the passive object *elevator status and plan.*

3. **Request Elevator**: The user presses the up or down button at a floor, which triggers an interrupt to the component *floor buttons interface*, which in turn sends a message *service request* to the component *scheduler*. The *scheduler* receives message and interrogates the passive object *elevator status and plan* to determine if an elevator is on its way to this floor. If not, the *scheduler* selects an elevator and sends a message *elevator request* to the component *elevator manager*. The *elevator manager* receives the message and records destination in the passive object *elevator status and plan.*

Consider a building with 10 floors and 3 elevators. All e2e scenarios are driven by external interrupts. We estimate the worst-case arrival rate of the interrupts and use them as approximations for periods assigned to each task. For example, the **Request Elevator** scenario is assigned a period of 100ms by assuming that all 18 floor buttons (up and down buttons for each floor, except for the top and bottom floors) are pressed within 1.8 seconds,

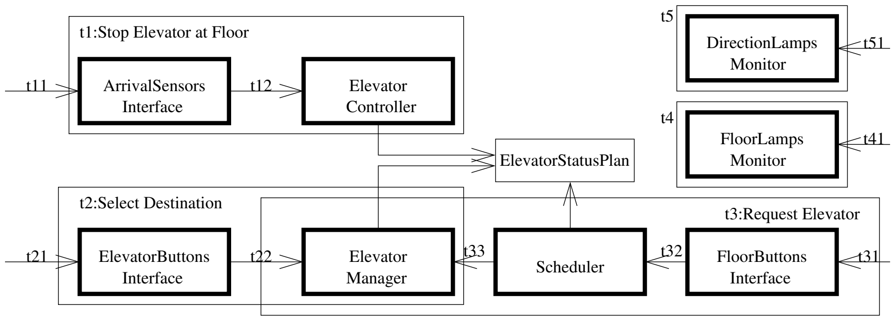

Figure 7. Collaboration diagram for the elevator control application.

Task	Period	WCET	Priority	WCRT
t_1				
t_{11}	25	2	9	-
t_{12}	25	5	6	19
t_2				
t_{21}	50	3	8	-
t_{22}	50	6	5	38
t_3				
t_{31}	100	4	7	-
t_{32}	100	12	4	-
t_{33}	100	6	5	46
t_4, t_5				
t_{41}	200	5	3	43
t_{51}	200	5	2	48

Table 2. Task set of the single-processor elevator control system. Time is measured in ms. Larger priority number denotes higher priority.

which is likely to be the worst-case arrival rate. We adopt CBMT, and apply the scheduling analysis technique discussed in Section 3.

Table 2 shows the task set of the elevator control system, assuming that all tasks run on a single processor. Priorities are assigned rate-monotonically. In addition, the interrupt-handlers, or the *Interface subtasks*, are given priority over the other subtasks in order to avoid missing any interrupts [9]. Other priority-assignment schemes are also possible. We only address the scheduling analysis problem given an existing priority assignment here, and leave the priority assignment problem to Section 4.

As an example, let's consider the e2e task t_2 **Select Destination**, which consists of two subtasks with WCET 3 and 6, priorities 8 and 5, respectively. Its canonical form is a single task with execution time 9 and priority 5. Other tasks can be classified as follows.

- t_1 is a type-1 task, with a single higher-priority segment with WCET 7.
- t_3 is a type-2 task, with a higher-priority segment t_{31} followed by lower-priority segments t_{32} and t_{33}.
- t_4 and t_5 are type-5 tasks, with all segments having priorities lower than 5.

The blocking time $B_2(2)$ caused by type-2 tasks is $\text{WCET}(t_{31}) = 4$. There are no type-4 tasks. To keep it simple, we assume the context-switching overhead is zero. The blocking time due to RTC semantics is $\text{WCET}(t_{33}) = 6$, and that due to shared passive objects is $\max(\text{WCET}(t_{12}), \text{WCET}(t_{32})) = \max(5, 12)$. We use Eq. (1) to get:

$$\text{WCRT}(2) = \text{WCET}(2) + B_2(2) + B_4(2) + B(2)$$
$$+ \sum_{j \in H_1(2)} \lceil \frac{\text{WCRT}(2)}{\text{Period}(j)} \rceil \cdot \text{WCET}(j)$$
$$= 9 + 4 + 6 + \max(5, 12) + \lceil \frac{\text{WCRT}(2)}{50} \rceil \cdot 7 = 38.$$

We can calculate WCRT for all the e2e scenarios based on Eq. (1), as shown in the WCRT column of Table 2. (We associate WCRT of e2e scenarios with the last segment of the task in the table.) No deadlines are missed, hence the system is schedulable. Note that t_4 and t_5 have relatively small WCRT despite the fact that they have the lowest priority, since they do not suffer from blocking time caused by RTC semantics or shared passive objects.

4 Optimization with Simulated Annealing

Having considered the problem of scheduling analysis given a certain configuration of component-to-thread mapping and priority assignment, we now address the issue of how to arrive at a configuration that optimizes a system design objective by performing design space exploration, including choosing multi-threading strategies and assigning priorities to threads. There are certain constraints on component-to-thread mapping. Some components are mutually-exclusive, and cannot be allocated to the same thread. For example, a component with a large WCET should not be allocated to the same thread as another component that needs to finish its work within a short deadline. There may also be application-level semantic reasons for mutual-exclusion. For example, certain components are safety-critical and should not be allocated to the same thread with certain non-safety-critical components in order to achieve a certain degree of isolation. Also, thread precedence relationships cannot contradict component precedence relationships. For example, if there is a component precedence relationship $C_1 \rightarrow C_2 \rightarrow C_3$, we cannot allocate C_1 and C_3 to T_1 and C_2 to T_2. This is a useful constraint that reduces the search space considerably.

Given these design constraints and the optimization objective of maximizing the CSF, we have a constrained optimization problem. It is generally infeasible to explore the design space exhaustively except for small systems, since the size of the design space grows exponentially with the number of components. Note that this is not a *convex* optimization problem, since we do not have a smooth analytic relationship between the optimization objective (CSF) and the tunable parameters (component-to-thread mapping and priority assignment). Instead of analytic solutions, we resort to heuristic optimization techniques such as Simulated Annealing (SA), Genetic Algorithms (GA), Tabu Search and Branch-and-Bound, which often work well even without deep insight into the underlying problem. SA and GA are stochastic algorithms while branch-and-bound and tabu search are deterministic algorithms. Since global optimization problems are generally NP-complete, none of these techniques can guarantee to find the optimal solution. Conceptually, most optimization problems can be solved with any one of these techniques, but certain problems can be formulated and solved more naturally in one approach than in others. We choose SA as our optimization approach due to its simplicity and ease of problem formulation. SA is a global optimization method that tries to find the global optimal point in the design space by jumping over local optimal points. The basic idea works as follows. From current state, pick a random successor state. If it has a better value than current state, then accept the transition, that is, use the successor state as current state. Otherwise, do not give up, but instead flip a coin and accept the transition with a given probability, which is lower as the successor is worse. So we sometimes "un-optimize" the value function a little with a non-zero probability in order to avoid being stuck in a local optimal point. The SA algorithm works similarly to random search at high temperatures, and to greedy steepest-descent at low temperatures.

An important part of SA is the definition of the *energy function*, and the *annealing schedule*. The energy function determines the optimization objective, which is defined as the negation of CSF. Therefore, minimization of the energy function amounts to maximization of CSF. The annealing schedule has an important impact on optimization quality. It is measured with *temperature step size*, the size of temperature drop at each iteration of the SA algorithm. Intuitively, the slower the temperature cools down, the better the optimization quality, but the algorithm running time also gets longer. Hence there is a tradeoff between optimization quality and amount of time spent during the optimization procedure. Our annealing schedule is chosen such that the algorithm running time is generally within minutes or tens of minutes.

Algorithm 1 The Simulated Annealing algorithm.

```
Parameters
$\delta t$; /*Temperature step size*/
$T_{threshold}$; /*Lower bound on temperature*/
Initialization
$T := T_0$; /*Initial temperature*/
Config := InitConfig; /*Initial system configuration*/
$E := -CSF(Config)$; /*Initial system energy*/
while true do
  /*Find a new system configuration and calculate $E_{new}$
  using RT scheduling algorithms described in Section 3;*/
  /*If we reach a system configuration with a lower energy,
  then accept it.*/
  if $E_{new} < E$ then
    Accept the new configuration
    $E := E_{new}$
  else
    /*If we reach a system configuration with a higher
    energy, then accept it with a certain probability. */
    if $e^{(E-E_{new})/T} <$ random$(0,1)$ then
      Accept the new configuration
      $E := E_{new}$
    else
      Reject the new configuration
    end if
  end if
  /*Lower the annealing temperature.*/
  $T := T - \delta t$
  /*If we have reached the lower threshold of the annealing
  temperature or the system energy, then finish.*/
  if $T \leq T_{threshold}$ then
    break
  end if
end while
```

Algorithm 1 shows the pseudo-code for the SA algorithm. The initial system configuration is defined as follows: assign each component to its own thread, and assign thread priorities rate-monotonically. For threads with the same execution rate, assign priority in the order of execution precedence relationship, i.e., threads near the head of the e2e scenario are assigned higher priority than those near the tail. The new system configuration, i.e., neighborhood of a given point in the design space is found with one of the following actions: randomly swap the priority assignments of $O(\tau_{ij})$ and $O(\tau_{ik})$ in any e2e scenario i; randomly choose two adjacent threads in the same e2e scenario, assign them the same priority and merge them into one thread; randomly choose one thread containing more than one component, and break it up into multiple threads with different priorities. Note that it does not make sense to have two adjacent threads in the same e2e scenario assigned the same priority, since this configuration only adds additional context switching time without any other benefits. Therefore, we always assign different priorities to adjacent threads, or merge them if they are assigned the same priority. The algorithm ends when the temperature drops below a threshold $T_{threshold}$.

5 Experimental Evaluation

In order to stress-test our optimization algorithm with varying workloads, we constructed artificial component dependency graphs with the following parameters: number of components: 10–200; scenario size (the number of components contained in each e2e scenario): 5–30; period: 10–1000ms; WCET: 0.5–8% of period; context-switch overhead: 30μs.

The experiments were performed on a PC with 933MHz CPU speed and 256MB of memory running Windows XP. Figure 8 shows sample experimental results for two different annealing schedules, i.e., temperature step sizes. Results may differ depending on the characteristics of different applications, but Figure 8 is representative of other experimental results.

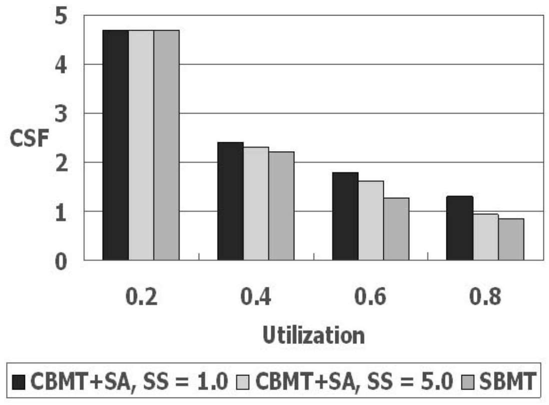

Figure 8. CSF plotted against total system utilization. SS refers to temperature step size of the annealing schedule.

As shown in Figure 8, when system utilization is low, that is, the system is lightly-loaded, there is not much difference between SBMT and CBMT. But when utilization gets higher, the benefit of CBMT becomes more apparent. When utilization grows to 0.8, CSF for SBMT drops below 1.0, meaning that the system is not schedulable, while CSF for CBMT is 1.30 when the temperature step size is 1.0. The reason for this is that we are able to find a better design by exploring a much larger design space using CBMT. When we adopt a larger temperature step size, i.e., a faster annealing schedule, we achieve less optimal results. When temperature step size is 5.0 and system utilization is 0.8, the SA algorithm was unable to find a configuration that makes the system schedulable (CSF ≥ 1), even though it did manage to find a configuration with a larger CSF value (0.93 instead of 0.84).

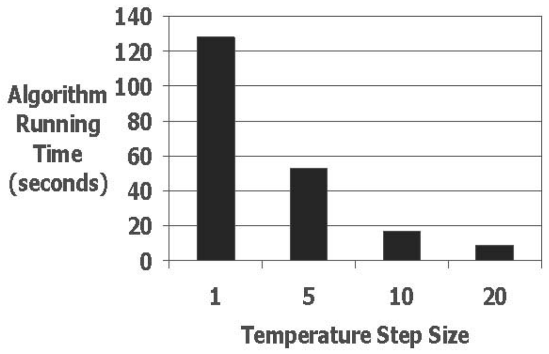

Figure 9. Running time of the SA algorithm for a small application example with 30 components.

The algorithm running time for our examples generally falls within minutes to tens of minutes, depending on the number of components, the component connection topology and the annealing schedule. In this paper, we have considered application examples that are fairly small. For realistic applications with thousands of components, the running time can easily reach hours or days. As shown in Figure 9, increasing the temperature step size has a considerable impact on reducing algorithm running time, at the expense of generating a less optimal system configuration. Since the optimization is performed offline, it is generally acceptable to have long algorithm running times. However, it is still desirable to keep the running time within a certain range, since the designer may sometimes have to run the SA algorithm multiple times, each time with a different seed for the random number generator, in order to find the configuration with an acceptable quality metric.

Note that SBMT is not a special case of CBMT, which requires fixed, static priorities to be assigned to groups of components. Instead, SBMT requires fixed priorities to be assigned to each e2e scenario, so each component may execute at different priority levels, depending on which scenario it is involved in. Therefore, CBMT does not always achieve a more optimal design than SBMT. In fact, evaluation experiments show that for a certain class of applications, SBMT actually results in an implementation with a larger CSF despite the larger design space and more exhaustive design-space exploration for CBMT. From observation of application characteristics, we propose the following heuristics for helping the designer choose a suitable multi-threading strategy: if there is very little interaction between different application scenarios, then SBMT is appropriate. However, if there are extensive interactions among different scenarios, then CBMT is more appropriate in order to avoid excessive locking and unlocking of shared components. This is the case for the elevator control application, where the shared passive component *ElevatorStatusPlan* is the main cause of close interaction among scenarios *t1*, *t2* and *t3*.

6 Related Work

Merrick *et al.* [12] developed a priority refinement method using SA for dependent tasks distributed throughout a heterogeneous multi-processor environment. Kodase *et al.* [13] used SA to assign priorities to components, then group all components that are assigned the same priority into the same thread. Our work improves upon previous work by having integrated component-to-thread allocation and priority assignment, and exploring additional degrees of freedom in the design space in order to find a more optimized solution.

There has been a lot of work on model-based and component-based design tools. Representative projects include CoSMIC [14], VEST [15], Time Weaver [4], the ESML-Based Tool-Chain [16], and others. CoSMIC [14] uses the Platform-Independent Modeling Language (PICML) to enable developers to define component interfaces, QoS parameters and software building rules, and generate descriptor files that facilitate deployment of applications based on CORBA Component Model. PICML is designed to help bridge the gap between design-time tools and the actual deployed component implementations. VEST (Virginia Embedded Systems Toolkit) [15] is an integrated environment for constructing and analyzing component based embedded systems. Aspect-checks are used to check for cross-cutting non-functional properties, and prescriptive aspects are used to apply cross-cutting advice to design models. Time Weaver [4] is a real-time software composition framework that allows the designer to experiment with different physical design/deployment decisions from components to the physical platform. As part of Time Weaver, de Niz *et al.* [17] recently developed heuristic algorithms based on bin-packing for component allocation to distributed processors that minimize the number of processors needed as well as the network communication load. The ESML-Based Tool-Chain [16] provides an open and integrated development environment that can be used to perform real-time schedulability analysis and model-checking. There tools typically provide capabilities to generate glue code, e.g., configuration files for component assembly, and to analyze system schedulability. However, they do not address the problem of proactively searching for an optimal thread configuration that optimizes a scheduling objective such as CSF. Also, these tools mainly target the Avionics Mission Computing [16] application from Boeing, which follows the SBMT runtime model.

Regehr [5] developed a framework for robust scheduling by introducing two new scheduling abstractions, task cluster and task barrier, and using greedy randomized search algorithms to find optimal task attribute assignments that maximize the CSF. It is straightforward to convert randomized search into simulated annealing by adding logic to probabilistically accept inferior solutions. Bartolini *et al.* [18] developed heuristic algorithms to map from a dataflow logical model to a multi-tasking implementation using Earliest Deadline First (EDF) scheduling. Fredriksson *et al.* [19] used genetic algorithms to find an optimized allocation from components to tasks, with the goal of reducing context-switching overhead and memory consumption while respecting constraints on allocation. None of these authors considered the task model of each e2e scenario consisting of subtasks with varying priorities, and the additional blocking time caused by the RTC semantics of the component-based, event-driven interaction style.

7 Conclusions and Future Work

In this paper, we have considered the class of component-based software models with interaction style of buffered asynchronous message passing between components with ports, a prevalent interaction style for large-scale RTE software. We consider different multi-threading strategies for implementation synthesis, and focus in particular on the CBMT runtime model, which creates a large design space of different options for component-to-thread mapping and priority assignment. Since this runtime model does not fit the assumptions of classic real-time scheduling theory (RMA), we have developed real-time scheduling techniques for it by enhancing the HKL algorithm with additional blocking time caused by RTC semantics. We have also developed optimization techniques based on simulated annealing to find a system configuration with largest CSF. Experimental evaluation shows that our techniques can synthesize high-quality multi-threaded implementations with reasonable running time of the optimization algorithm. Our approach helps bridge the gap between a logical software model and its physical implementation on the target platform, by giving the designer real-time scheduling analysis techniques for evaluating different alternatives, as well as design-space exploration techniques for generating a multi-threaded implementation from a logical software model. Our work focuses on para-functional issues involved in implementation synthesis, and is complementary to existing code generators for component-based design tools, which focus on generating functional code in programming languages or configuration files for component assembly.

There are a number of directions for future work. First, in order to achieve industry acceptance, we are considering integrating our techniques into CASE Tools such as the Eclipse environment [20], which has a plug-in for UML 2.0 metamodel, and possibly open-source scheduling analysis tools such as OpenSTARS [21]. Second, we would like to consider other resource constraints and para-functional requirements besides real-time, e.g., memory size and power consumption, during the optimization process. Third, since the running time of SA can get very long for large systems, we would like to develop efficient heuristic algorithms for finding feasible and/or near-optimal solutions quickly. Fourth, we would like to extend our current optimization techniques to address distributed systems, which add an additional degree of freedom to the design space, i.e., allocation of software components or threads to different processors [17].

Abbreviations Used

CBMT	Component-Based Multi-Threading
CSF	Critical Scaling Factor
e2e	End-to-End
RTC	Run-To-Completion
SBMT	Scenario-Based Multi-Threading
WCET	Worst-Case Execution Time
WCRT	Worst-Case Response Time

Acknowledgement

The work reported in this paper was supported in part by the Army Research Office under Grant No. DAAD19-01-1-0473.

References

[1] B. Selic, G. Gullekson, and P. T. Ward, *Real-Time Object Oriented Modeling*. Addison Wesley, 1994.

[2] (2005) The SDL Forum website. [Online]. Available: http://www.sdl-forum.org

[3] (2005) The Quantum Framework website. [Online]. Available: http://www.quantum-leaps.com/qf.htm

[4] D. de Niz and R. Rajkumar, "Time weaver: A software-throuhg-models framework for embedded real-time systems," in *Proc. ACM Conference on Languages, Compilers and Tools For Embedded Systems (LCTES)*, 2003, pp. 133–143.

[5] J. Regehr, "Scheduling tasks with mixed preemption relations for robustness to timing faults," in *Proc. Real-Time Systems Symposium (RTSS)*, 2002, pp. 315–326.

[6] S. Vestal, "Fixed-priority sensitivity analysis for linear compute time models," *IEEE Trans. Software Eng.*, vol. 20, pp. 308–317, 1994.

[7] M. Saksena and P. Karvelas, "Designing for schedulability: integrating schedulability analysis with object-oriented design," in *Proc. IEEE Euro-Micro Conference on Real-Time Systems (ECRTS)*, 2000, pp. 101–108.

[8] S. Kim, S. Hong, and N. Chang, "Scenario-based implementation architecture for real-time object-oriented models," in *Proc. IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, 2002, pp. 147–152.

[9] M. H. Klein, T. Ralya, B. Pollak, and R. Obenza, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems.* Kluwer Academic Publishers, 1993.

[10] M. Harbour, M. H. Klein, and J. Lehoczky, "Timing analysis for fixed-priority scheduling of hard real-time systems," *IEEE Trans. Software Eng.*, vol. 20, no. 2, pp. 13–28, 1994.

[11] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML.* Addison-Wesley, 2000.

[12] J. R. Merrick, S. Wang, K. G. Shin, J. Song, and W. Milam, "Priority refinement for dependent tasks in large embedded real-time software," in *Proc. IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, 2005, pp. 365–374.

[13] S. Kodase, S. Wang, and K. G. Shin, "Transforming structural model to runtime model of embedded software with real-time constraints," in *Proc. Design, Automation and Test in Europe Conference (DATE)*, 2003, pp. 170–175.

[14] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt, "A platform-independent component modeling language for distributed real-time and embedded systems," in *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2005.

[15] J. A. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis, "Vest: an aspect-based composition tool for real-time systems," in *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2003, pp. 58–69.

[16] Z. Gu, S. Wang, S. Kodase, and K. G. Shin, "An end-to-end tool chain for multi-view modeling and analysis of avionics mission computing software," in *Proc. IEEE Real-Time Systems Symposium (RTSS)*, 2003, pp. 78–81.

[17] D. de Niz and R. Rajkumar, "Partitioning bin-packing algorithms for distributed real-time systems," *International Journal of Embedded Systems*, 2005.

[18] C. Bartolini, G. Lipari, and M. D. Natale, "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 (RTAS)*, 2005, pp. 458–467.

[19] J. Fredriksson, K. Sandstrom, and M. Akerholm, "Optimizing resource usage in component-based real-time systems," in *Proc. ACM International Symposium on Component-Based Software Engineering (CBSE)*, 2005, pp. 49–65.

[20] (2005) The eclipse website. [Online]. Available: http://www.eclipse.org

[21] K. Bryan, T. Ren, J. Zhang, L. C. DiPippo, and V. F. Wolfe, "The design of the OpenSTARS adaptive analyzer for real-time distributed systems," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.