QoS-Optimized Integration of Embedded Software Components with Multiple Modes of Execution *

Zonghua Gu
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Hong Kong, China

Qingxu Deng
Department of Computer Science
Northeastern University
Shenyang, China

Abstract

Component-Based Software Engineering (CBSE) is an effective approach for tackling the increasing complexity of large-scale embedded software development projects. Software components often have multiple modes of execution, each characterized by a set of resource requirements such as maximum stack and heap memory size, worst-case execution time, etc, and one or more Quality of Service (QoS) values that measure its benefit. There is typically a tradeoff relationship between the resource requirements and utility value. In this paper, we address the problem of optimizing system utility when composing multiple software components into a complete system, each having a set of discrete modes of execution with different resource requirements and utility values. We present an optimal branch-and-bound algorithm and another fast heuristic algorithm for solving the optimization problem.

1 Introduction

Embedded software components are often modal, each having several modes of execution, and each mode having a different set of QoS value and resource consumption characteristics. For example, a software component in an embedded control system may have two possible runtime modes: one implementing a sophisticated control algorithm with good control performance but has high resource requirements in terms of execution time and memory size, and the other implementing a simplistic control algorithm with below-average control performance but has low resource requirements. In this paper, we assume that we can assign an application-specific utility value to each mode to quantify its QoS. For example, in control theory, the control performance can be measured with a composite index number that combines the steady-state error, maximum overshoot, settling time and rise time.

As a more concrete example, Figure 1 shows a modal component taken from Avionics Mission Computing [1] with two high-level modes of execution: active and inactive. The active mode further has two sub-modes within it, each offering the same input/output interface but with different resource consumption and performance characteristics. When the component is in the TacSteering mode, it produces results with high accuracy but also has high CPU requirements; when it is in the NavSteering mode, it produces results with lower accuracy but has lower CPU requirements. The component can switch to a different mode at runtime depending on runtime resource availability. This can be implemented by having multiple implementations for each component, loading all of them into memory at system startup, and switching among them dynamically at runtime.

As a demonstration of the modal component in ac-

\*This work is partially supported by the Cultivation Fund of the Key Scientific and Technical Innovation Project (No 706016), Ministry of Education, China and the Competitive Earmarked Research Grant (No 613506), Research Grants Council, Hong Kong.
Figure 2. An application scenario.

ation, Figure 2 shows an application scenario that contains the modal Steering component. The behavior of this scenario can be described in three stages depending on the mode of the Steering component, which can switch between three modes, i.e., Inactive, NavSteering and TacSteering, at runtime:

1. The GPS component wakes up upon a 50ms (20Hz) interval timeout and issues a Data Available event. The Airframe component receives the event and issues its own Data Available event. The Steering component receives this event, updates its internal state, but does not issue events since it is initially inactive.

2. PilotControl issues a command that enables the NavSteering mode of the Steering component. Now when the Steering component receives the Airframe’s Data Available event, it updates its state and issues a Data Available event, causing the NavDisplay to display the navigational steering data obtained from the NavSteeringPoints component.

3. PilotControl issues a command that enables the TacSteering mode of the Steering component. From this point on, the NavDisplay component displays the tactical steering data instead of the navigational steering data.

Embedded systems used to be designed in a fairly static fashion for the sake of predictability, with a fixed set of tasks run at a static cyclic schedule. However, modern embedded systems are required to handle more dynamic situations, where unexpected tasks may arrive and leave the system at runtime, and system resource availability may also change dynamically due to runtime load variations and hardware faults. This requires the engineer to add flexibility and adaptivity to runtime execution, in order to protect the system from overload by trading off resource requirements for QoS and achieve graceful degradation. Existing work on dynamic runtime adaptation is typically programmed in an ad hoc manner, and reconfiguration code is tangled with application functional code. This breaks component modularity, a key benefit of CBSE, and hinders system maintenance and evolution. We propose an approach of predefining a set of system modes, each of which is a combination of component modes designed and optimized offline at design time according to some optimization criteria. In each mode, a different set of components are active, and each active component may be in one of its sub-modes depending on the current active system mode. At runtime, one of the system modes is activated as the initial default mode. The system may change modes during execution to accommodate dynamic resource availability. The runtime middleware monitors runtime conditions and dynamically switches between system modes based on runtime resource availability by changing component modes. Its role is similar to a transaction manager in database systems, and coordinates system-wide mode changes while maintaining global consistency. This approach allows the designer to use time-consuming optimization techniques to predefine and optimize system modes at offline design time, yet incurs low overhead for performing dynamic mode changes at runtime. Figure 3 shows the overall workflow.

Possible resource constraints include total memory size, CPU utilization (percentage of time when the CPU is busy), network bandwidth, etc. For example, for a system with n tasks scheduled in a rate monotonic fashion, the upper bound of the total CPU utilization in order to guarantee schedulability is $n(2^{1/n} - 1)$. We solve the optimization problem to obtain a set of system modes, each characterized by a tuple [CPU utilization, total memory size, system utility] expressing the tradeoff relationships between the available resources and the achievable system utility. At runtime, the middleware adaptively switches among different system modes to cope with varying system load and resource availability, e.g., when the system is lightly loaded, then upgrade to a system mode with high utility; when overload occurs, downgrade to a system mode with low utility by producing lower quality results. In this paper, we focus on the design stage, where the engineer
selects a mode for each component in the application to form multiple system modes, each optimized for a different design objective, such as high performance or low resource requirements.

The rest of this paper is structured as follows: we present the problem formulation in Section 2, then present two optimization techniques in Section 3, one optimal algorithm based on exhaustive search and another fast heuristic algorithm. We present performance evaluation results in Section 4. We discuss related work in Section 5, and draw conclusions and discuss future work in Section 6.

2 Problem Formulation

An embedded software application consists of a set of real-time tasks, each containing a set of software components. Here we assume that each software component belongs to at most one task, and it is not possible for multiple tasks to share one component, as discussed in [2]. There are \( n \) software components in the system \( \{C_1, C_2, \ldots, C_n\} \), taking into account components in all tasks. The available system resources are given by a vector \( \mathbf{R} = (R_1, \ldots, R_m) \). Each component \( C_i \), where \( i = (1, \ldots, n) \), has \( l_i \) possible execution modes \( C_{ij} \), where \( j = (1, \ldots, l_i) \), each of which has a set of resource requirements \( \mathbf{r}_{ij} = (r_{ij1}, \ldots, r_{ijm}) \), and a user-defined utility metric \( v_{ij} \). (There may be multiple utility metrics in general, but we only consider the case with a single utility metric here.) There may be mutual exclusion constraints, e.g., no more than one component among a set of components is active in any system mode, and mutual attraction constraints, e.g., a set of components are always active simultaneously, or none of them are active in any system mode. The component integrator must define a small set of system modes such that each satisfies resource constraints and optimizes the utility metrics. For example, one system mode may be high-performance with high resource consumption, while another may be degraded-performance with minimal resource consumption, designed to cope with the situation where certain faults may have occurred and reduced the system resource availability.

Assuming each component mode is associated with a set of resource requirements and a utility value, the optimization problem can be formulated as finding the combination of component modes with the largest total utility value given total cost constraints. A special case is when all constraints are mutual attraction, and the system has a few fixed system-level modes of certain component mode combinations without any freedom for design space exploration. In this paper, we assume that there is some flexibility in choosing component modes, hence there is a need for searching through a potentially large design space.

Suppose we have two resource constraints, CPU utilization \( U \) and total memory size \( M \); one optimization objective, an application-specific utility value \( V \), with a monotonic tradeoff relationships \( V \) grows larger with larger \( U \) and \( M \). We can formulate the optimization problem in several different ways:

1. Given an upper bound on \( U \) and an upper bound on \( M \), maximize \( V \).
2. Given a lower bound on \( V \) and an upper bound on \( M \), minimize \( U \).
3. Given a lower bound on \( V \) and an upper bound on \( U \), minimize \( M \).
4. Optimize \( U \), \( M \) and \( V \) jointly to obtain a Pareto optimal curve using Evolutionary Algorithms [3].

In other words, we can optimize one objective while treating the others as constraints, or we can jointly optimize all objectives simultaneously. The formulation of joint multi-objective optimization is more general.

\[ \text{A Pareto optimal curve for a multi-objective optimization problem contains all the solutions where no solution is inferior or superior to any other solution for all objectives.} \]
than single objective optimization, since the single-objective solutions can be easily obtained from the Pareto optimal curve. In this paper, we focus on the 1st problem formulation by using both exhaustive search and fast heuristic algorithms.

3 Solving the Optimization Problem

Branch-and-bound is widely used for solving combinatorial optimization problems. It works by constructing a search tree, and each node in the tree represents a solution where some variables are assigned and others are free. A node is expanded by assigning value to a free variable, e.g., expanding a node by assigning value to a binary variable \( x \) generates two nodes: one with \( x = 0 \) and another with \( x = 1 \).

For our optimization problem, each leaf node in the search tree is a complete solution represented by a vector \( x = (x_{ij}) \), where \( i = 1, \ldots, n \) and \( j = 1, \ldots, l_i \). For component \( C_i \), \( x_{ij} = 1 \) means that mode \( j \) of component \( i \) is active, and \( x_{ij} = 0 \) means that it is inactive. We use another bit vector \( b = (b_1, \ldots, b_n) \) to represent the status of mode assignment for components \( (C_1, \ldots, C_n) \). \( b_i = 1 \) indicates that \( C_i \)'s mode has already been assigned, and \( b_i = 0 \) indicates that \( C_i \)'s mode has not been assigned. The algorithm has the following key steps: start with the initial state where none of the component modes have been assigned. Find the tree node \( N \) with the largest upper bound on utility value. If the bit vector \( b \) for node \( N \) contains all 1's, then all component modes have been assigned. Record a solution and backtrack. Otherwise, the bit vector \( b \) contains at least one 0, then expand node \( N \) by generating \( l_i \) children nodes to represent all possible mode assignments for a component \( C_{ij} \) whose mode has not been assigned. Prune any generated tree node that violates resource constraints and backtrack. Iterate until all component modes have been assigned, or no feasible solution is found.

Since combinatorial optimization problems are NP-Hard in general, running time of the branch-and-bound algorithm grows exponentially with the number of components. Suppose an application consists of 3 components, and each component has 2 execution modes, then we have \( 2^3 = 8 \) possible system modes. In order to scale up to larger models, we next present a heuristic algorithm for solving the optimization problem. The algorithm has the following key steps:

1. Start with the system mode where each component is in the mode with the smallest utility value. If this system mode does not satisfy memory size or utilization constraints, then there is no feasible solution, and the algorithm terminates.

2. Find the additional resources \( r = (r_1, \ldots, r_m) \) needed to upgrade each component to the next higher utility value, and choose the component mode upgrade that provides the largest utility gain divided by additional resource demand.

3. If a feasible component mode upgrade cannot be found, then return the current solution and terminate. Otherwise, go to step 2.

In the above description, we start from a feasible solution with lowest utility value and gradually upgrade each component to a higher utility mode until the resource bound is violated. We then choose the solution that satisfies the resource bounds and has the largest utility. It is also possible to start from an infeasible solution with highest utility value and gradually downgrade each component to a lower utility mode until the resource bound is satisfied. The former approach may converge to the solution faster when resources are scarce, and the latter approach may converge faster when resources are abundant.

4 Performance Evaluation

To evaluate the performance of our optimization algorithms, we generate sets of modal components, and assign random values of CPU utilization and utility to each component mode. We assume that CPU is the bottleneck resource and do not consider memory size constraints, therefore the resource vector \( r \) becomes a single scalar value. We optimize the total system utility given different upper bounds on CPU utilization. The optimization algorithms are run on a Pentium PC with 3.2 Ghz CPU and 1GB memory. Table 4 shows the performance results. \( N_c \) denotes total number of components; \( M_r \) denotes number of modes per component; \( UB \) denotes the utilization bound; \( V_{BB} \) denotes the maximum utility achieved with branch-and-bound algorithm. When the utilization bound is 1.0, we scale and normalize the maximum utility to be 1 for easy comparison. \( V_{HEU} \) denotes the maximum utility achieved with the heuristic algorithm; \( T_{BB} \) denotes the branch-and-bound algorithm running time; \( T_{HEU} \) denotes the heuristic algorithm running time. For the heuristic algorithm, we start from the system mode where each component is assigned the mode with lowest utility value, and scale up gradually. As we can see, running time of the branch-and-bound algorithm increases sharply with the number of components and number of modes per component, while running time of the heuristic algorithm stays more or less constant. Optimization results of the heuristic algorithm are remarkably close to the true optimal calculated with
branch-and-bound, as we can see by comparing the columns labeled $V_{DB}$ and $V_{HEU}$. Note that the algorithm running time gets smaller when the utilization bound is reduced, which results in reduction of design space size. The final result is a set of system modes, each consisting of a collection of component modes and characterized by a tuple [CPU utilization, system utility]. At runtime, different system modes may be activated depending on runtime resource availability.

Table 1. Performance evaluation results.

<table>
<thead>
<tr>
<th>$N_c$</th>
<th>$M_c$</th>
<th>$V_{DB}$</th>
<th>$V_{HEU}$</th>
<th>$T_{DB}$</th>
<th>$T_{HEU}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>0.8</td>
<td>0.99</td>
<td>1.7</td>
<td>0.21</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0.6</td>
<td>0.78</td>
<td>0.9</td>
<td>0.11</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>1.0</td>
<td>1.0</td>
<td>0.94</td>
<td>34.2</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>0.8</td>
<td>0.86</td>
<td>0.82</td>
<td>27.3</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>0.6</td>
<td>0.67</td>
<td>0.64</td>
<td>20.8</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>1.0</td>
<td>1.0</td>
<td>0.95</td>
<td>403.2</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>0.6</td>
<td>0.59</td>
<td>0.53</td>
<td>316.2</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>1.0</td>
<td>1.0</td>
<td>0.97</td>
<td>2598.1</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>0.8</td>
<td>0.91</td>
<td>0.87</td>
<td>2109.1</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>0.6</td>
<td>0.64</td>
<td>0.62</td>
<td>1600.5</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>1.0</td>
<td>1.0</td>
<td>0.89</td>
<td>9374.0</td>
</tr>
</tbody>
</table>

5 Related Work

Most existing work on CBSE focuses on the issue of component integration based on functional criteria, i.e., how to choose a set of components that implement certain functionality and are compatible with each other. Our focus is on component integration based on non-functional criteria, where the designer must choose among multiple component modes to form system modes that offer similar functionality, but have different resource requirements and performance/utility attributes.

Some researchers have addressed the problem of embedded software component integration. Sedigh-Ali [4] presented a graph-based model for component-based software development with optimization objectives such as system reliability, complexity metric and cost constraints, but no details on solving the problem were provided. Neema et al [5] developed a design space exploration tool (DESERT) based on Boolean Decision Diagrams (BDD) data structure. The tool starts with a set of design templates, each template consisting of several possible implementation alternatives, and synthesizes fully specified models that meet selected design constraints. However, the DESERT approach allows pruning of design space based on design constraints, but does not offer the capability of optimization. That is, it gives the designer a set of implementation alternatives that all satisfy design constraints, but does not give a ranking of the alternatives based on one or more optimization objectives. Wandeler et al [6] introduced a component system for interface-based design of systems with mixed Fixed Priority, Rate Monotonic and Earliest Deadline First scheduling. Henzinger et al [7] presented an assumption-guarantee interface algebra for real-time components that supports both the incremental addition of new components and the independent stepwise refinement of existing components. Gößler et al [8] proposed a framework for component-based modeling using an abstract layered model for components, which ensures correctness by construction of a system from properties of its interaction model and of its components. The properties considered include global deadlock-freedom, individual deadlock-freedom of components, and interaction safety. However, optimization of QoS under resource constraints is not considered.

Abdelzaher et al [9] proposed a model for quality-of-service (QoS) negotiation in building real-time services with the goal of guaranteeing predictable performance under specified load and failure conditions, and ensuring graceful degradation when these conditions are violated. Since their algorithm was designed to operate at runtime, an efficient greedy heuristic is adopted to tradeoff computation time with solution optimality. Our approach is based on offline optimization of multiple pre-defined system modes and online switching among them, therefore we can afford to use time-consuming optimization techniques such as branch-and-bound to achieve more optimal results.

6 Conclusions and Future Work

In this paper, we address the problem of optimized integration of embedded software components with multiple execution modes. Even though we have used memory size and CPU utilization as resource constraints, and a user-defined utility metric as optimization objective, our problem formulation is quite general and abstract and can be easily extended to address other resource and utility function definitions. As an example, Critical Scaling Factor (CSF) [10, 2] was proposed to be a metric that measures the sensitivity of a real-time system to variations of task execution times. It is defined as the largest coefficient by which execution time of all tasks can be simultaneously multiplied while preserving feasibility. For example, if a system has CSF of 1.17, then if we multiply the execution time of all tasks by a number $n \leq 1.17$, the system would still be schedulable. However, any
$n > 1.17$ would render the system unschedulable. A larger CSF value means that the system is more robust to timing faults caused by inaccuracies in execution time estimation. A system is schedulable if and only if its CSF $n \geq 1$. Therefore, it is desirable to adopt scheduling algorithms and task timing attribute assignments to maximize the CSF. We can treat CSF as another optimization objective in addition to one or more utility metrics. One way to solve the multi-objective optimization problem is by converting it to a single-objective one using a weighted sum of the multiple objectives. However, this approach is often not desirable since the assignment of weights can be arbitrary when combining multiple unrelated metrics. Another approach is to formulate a multi-objective optimization problem and solve it using evolutionary algorithms [3].

References


