Performance Modeling and Analysis Techniques for Integrated Embedded Control Software Design

by

Shige Wang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2004

ABSTRACT

Performance Modeling and Analysis Techniques for Integrated Embedded Control Software Design

by
Shige Wang

Chair: Kang Geun Shin

Today's software for embedded control systems has become large and complicated. Model-based design and analysis methods are therefore crucial for fast and low-cost development of embedded control software. In this dissertation, we developed techniques to support the performance analysis of embedded control software at various design phases. These techniques include a performance modeling framework and a set of modeling and analysis methods based on the framework. The modeling framework defines the performance parameters that capture the performance information required by performance analysis and performance-aware design. The values of these parameters are first represented as platform-independent virtual resource demands, then converted to the true resource demands after the software deployment is determined. Our modeling method is based on annotations supporting performance reuse and performance model evolution during the functional design of the software.

The performance analysis methods include methods for both early design performance estimations and runtime model performance analysis. The performance estimation requires only a software architecture model with performance annotations. The analysis results are bound estimations both of end-to-end response delays and of system resource demands, which are derived from the best-

case and worst-case configurations in an ideal execution environment. Such estimations can be used for performance analysis at an early design phase with software models containing no deployment information. We further demonstrated how to use the analysis results for software architecture and platform design. The analysis of a runtime model requires a complete software model, including its execution environment and deployment. Our runtime model analysis methods adopt existing real-time analysis algorithms, which we modify to fit our model. The results help identify the performance issues and can be used for design refinement.

Another technique developed in this dissertation is a performance-aware method of transforming design models. It takes the models of the software architecture and the platform as the inputs, and transforms the software model into a runtime model with the software deployed on the platform in such a way that both timing and resource constraints are met. Evaluations based on a set of randomly-generated system models have shown this method to be scalable and effective.

In order to collect the performance characteristics of application components and system software services, we developed a performance measurement method. It uses an end-to-end measurement with a combination of synthetic workloads and micro-benchmarks. Results of performance measured using this method can be reused in the performance analysis of a family of applications and platforms of a designated domain.

Finally, all the techniques we developed have been implemented in an embedded control software design toolkit, called the AIRES toolkit. The implementation has shown that these techniques can be easily integrated with any generic software modeling tool to support performance analysis. This is a significant advance in current software development tool support because it fills the gap in integrating software modeling with performance analysis.

To my wife, my son, and my parents.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to my advisor, Professor Kang G. Shin, who brought me to real-time and embedded research area and guided me to complete my degree. Besides knowledge, I have also learned patience, hard work, and pursuit of excellence, which are essential for a successful researcher, from him. I also thank Professor Toby Teorey, Professor James Freudenberg, and Dr. Daniel Kiskis who kindly agreed to serve on my dissertation committee, reviewed the draft of my dissertation, and provided valuable comments.

My appreciation also goes to the members in the MoBIES research group at the University of Michigan, including Sharath Kodase, Zonghua Gu, Channy Kim, and Jeffrey R. Merrick, and many other previous and current members in RTCL, including John Reumann, Padmanabhan Pillai, Daji Qiao, Haining Wang, Mohamed Gandy, Wei Sun, Min-Gyu Cho, and Zhigang Chen. They made my graduate student life fun and memorable.

I am also thankful to the DARPA MoBIES and NEST Programs for funding this research, and the engineers at Boeing Company, General Motors Corp., and Ford Motor Company for evaluating the research tool reported in this dissertation.

Finally, my deepest thanks go to my family. Through all these years, my wife, Chunzhi Bi, has provided her full support on every aspect of my life and shared every moment of my frustration and joy in this journey. My son, Boan, always brings me the best comfort with his lovely smile and naive behaviors. I am also grateful to my parents, Dejun Wang and Rulan Fu, who always show their unquestionable confidence in me and unconditional support, and to my little brother, Shili Wang, who was selflessly doing my duty at home when I studied abroad.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

Today's embedded control systems, such as automotive vehicle control, avionics mission control, manufacturing machine control, and nuclear plant control, have become large in size and complex in interactions. The embedded control software (ECSW) for these systems usually contains many drivers for various I/O devices, modules for control algorithms, and functions for system management (e.g., mode change and fault detection). Besides the functional and structural complexity, the ECSW must meet stringent timing and performance constraints in order to respond to stimuli from the external physical world. However, it requires a significant effort to design ECSW to meet all the timing and performance constraints since the execution environment for an embedded control system usually has limited resource for size and cost reasons. Recent research [23, 17, 52, 21, 84] has shown that performance-aware system design and model-based analysis have great potential for meeting combined functional and non-functional constraints, thus leading to low-cost and fast-to-market embedded real-time control systems.

To support model-based performance analysis and performance-aware ECSW design, new fundamental theories that can lead to a correct and effective ECSW design are requisite. These theories would also form a foundation for the new ECSW design methods (including design models and design processes) and development tool support. Developing such theories and demonstrating their usage and applications are therefore the objectives of this dissertation. In general, the design processes and methods used in ECSW design must be domain-specific to be effective, as different application domains encounter different design issues. For example, the design concerns of a sensor network are drastically different from those of an automotive vehicle control, although both are embedded systems. The design of the former must deal with a large number of very resource-limited (processor speed, memory, and power) communication-intensive devices with simple software, while the design of the latter must deal with complex software architecture but only a small number of devices with a relatively large software workload on each device. For the purpose of this research, we have limited our application domain to the latter, a complex software architecture running on a relatively small number of devices ($\sim 10^2$). For such a domain, the software is complex due to the large number of software components and interactions, so it requires scalable modeling and analysis methods for constructing and analyzing design models.

1.1 Models and Methods in Current ECSW Design

To develop methods for performance-aware design and performance analysis, it is necessary to understand the methodologies, models, and techniques used in current practices. Figure 1.1 shows the current ECSW development process in the *V-diagram* [21].

In the V-diagram, the process on the left side focuses on the system design. Each step in this process refines the design models generated at the previous step by filling in more design details. The process on the right side focuses on the system validation, which is accomplished through comparing obtained results with the requirements of the corresponding level of design. It is common for different models and methods to be used by multiple design groups at each step in the V-diagram. Some commonly-used design models and methods in current ECSW development are summarized below.

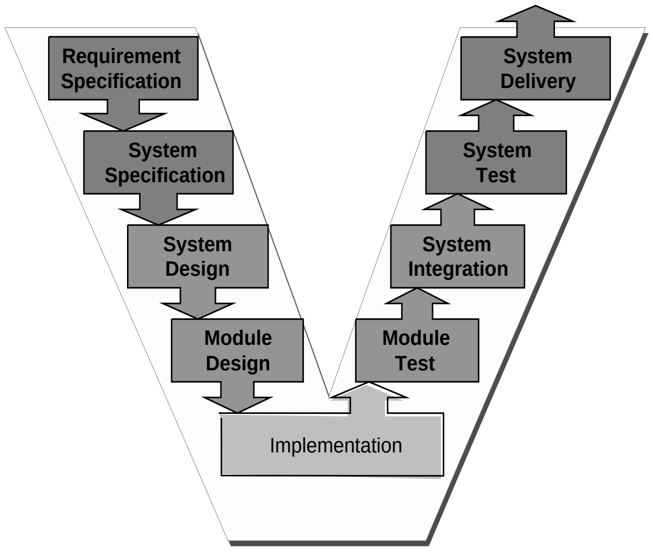


Figure 1.1: Overall ECSW development process.

1.1.1 System models in ECSW design

The system models used in current ECSW design can be generally classified into four categories [23]: *state-oriented*, *activity-oriented*, *structure-oriented*, and *heterogeneous*.

State-oriented models. A state-oriented model represents the system as a set of states, transitions between states, and events that trigger the transitions. The designed ECSW can be modeled as variables representing the system states, and functions generating events and implementing transitions. Since the correctness of state-oriented models can be mathematically verified and visualized, they are usually used to model and verify the dynamic behaviors of the ECSW. However, the implementation of the ECSW design in a state-oriented model is not straightforward due to the indirect mapping between the state-oriented model and the implementation model.

Activity-oriented models. An activity-oriented model represents the system as activities, and dependencies (including both data and execution dependencies) among the activities. Activity-oriented models are composable, and are suitable for systems with multiple concurrent-sequential operations. These models can be analyzed formally, using a formal method such as Concurrent

Sequential Process (CSP) [73], and can also be visualized. The ECSW design in an activity-oriented model is easy to implement as the modeling elements can be directly mapped to an entity in a programming language. This property is the reason why the activity-oriented models such as control and/or data flow graphs are widely used in ECSW design.

Structure-oriented models. A structure-oriented model represents a system as interconnected modules (called components). The designed system functionalities are implemented by composing the components, each of which implements a portion of the functionality. Structure-oriented models such as UML collaboration diagrams are also commonly used in ECSW design. As more software modeling languages and design tools use the structure-oriented models in current practices, many ECSW designs are now being modeled in structure-oriented models. Such a model is also easy to implement with the support of an object-oriented programming language such as C++ and Java. However, the system behaviors of a structure-oriented model are usually difficult to verify formally due to the lack of mathematical support.

Heterogeneous models. A heterogeneous model combines some of the above models to describe different aspects of ECSW in a single model. Such a model is usually used to capture interrelated properties of different system aspects simultaneously. For example, Real-Time Object-Oriented Model (ROOM) [74] uses a structure-oriented model for the software structure, and a state-oriented model for the software behaviors. Most of the current research models, including ModeChart [36], *chart [27], Actor model [2], and Multi-graph model [42], belong to this category. Because of the flexibility and power of such models, more of today's ECSW design tends to use heterogeneous models. Consequently, methods of design and analysis, and development tool support using heterogeneous models are still being actively researched.

1.1.2 Design methodologies

Commonly used design methodologies in current ECSW are based on the concept of separation of concerns [8]. These methods allow the designer to focus on a small set of design issues at a time, and to construct a complete design through a sequence of refinement steps. The correctness of the resultant design is ensured by the process and rules of refinement. Some well-known ECSW design methods include multi-view design [23, 21], software/hardware co-design [48], component-based software architecture [83], and product-line architecture [7].

Multi-view design. With multi-view design, a designer constructs a set of views. Each view models some aspect of the ECSW, and requires different modeling methods for construction and analysis. The views in ECSW design include *behavioral*, *structural*, and *physical*. A behavioral view specifies control of the system using state- and/or activity-oriented models. A structural view specifies that ECSW architecture implement the behaviors in the behavioral view using the structure-oriented model. A physical view specifies the runtime architecture and execution environments of ECSW, including the platform and deployment configuration. Most of the current performance analysis methodologies are applicable to the physical view only after the simulation or prototype system is constructed. This creates an information dependency cycle between design and analysis.

Software/hardware co-design. With this method, a designer first partitions system functions into software and hardware to find an optimal solution (e.g., minimum system cost) for an application. Given a system function, a hardware-implementation tends to run faster and more reliably, while software implementation is cheaper, more flexible, and more portable. The partitioned hardware and software functions are then designed separately as a set of hardware components (e.g., different chips, sensors, and actuators) and software components running on the designed hardware components. The performance analysis in this method is usually conducted during the system integration phase when hardware and software are integrated. As cost and time become more critical in ECSW development, current co-design methods are focusing on maximizing functions in software. Models used in the software/hardware co-design include continuous models for control design, and structure-oriented models once the functions have been partitioned into hardware and software.

Component-based software architecture. With component-based software architecture, ECSW is designed by integrating software components that include both the application components and the underlying system services. These components are implemented and reused for the construction of a family of applications. The components' structures and their allowed interactions are both defined in the architecture. The architecture also defines the common services required to support the components. Examples of component-based software architecture include CORBA, Java Bean and RMI, Microsoft CE .NET, and OMAC. The models used in component-based software architecture, such as UML, are usually heterogeneous, as the architecture needs to model many different aspects of ECSW. The performance of the design is usually analyzed through prototyping systems. The methods needed to verify and analyze the ECSW constructed using the component-based architecture are still an active research area because of model heterogeneity [14, 42, 53].

Product-line architecture. Product-line architecture supports parallel development of a family of applications that are functionally similar. The method used in such development is based on model parameterization using refinements and compositions. Product-line architecture includes both abstract components and concrete components in initial design. Both can be reused in software construction. With the product-line architecture, a designer first composes layers using abstract components, each of which implements a feature shared by multiple applications. The designer refines the abstract components at each layer with concrete implementations. Such a process can also be viewed as template parameterization. There is no effective analysis method for the performance analysis of the product-line architecture due to the system abstraction in the design process. The product-line architecture has been used successfully in avionics ECSW construction such as mission computing and navigation systems [20]. In these applications, the analysis is currently done through prototyping after the design is completed.

1.2 Proposed Research and Contributions

The objective of this research is to develop techniques for model-based performance analysis and performance-aware model transformation in ECSW design. These techniques should be integrated with the models and methods for functional design in current ECSW design practices, and should support the verification of non-functional properties of ECSW with a large number of components. The techniques must also include methods for detecting design flaws that cause violation of any performance constraint at various design phases, as well as methods to guide the design on component selection, runtime system generation, and implementation according to the analysis results.

1.2.1 Research problem and proposed solutions

The research problem addressed in this dissertation can be stated as follow.

> *Develop modeling and analysis methods that can be used with functional design models and processes to define the performance of the modeling constructs (components, interactions, and environments), derive the effects of implications of design decisions and policies on the system performance, and guide/automate the refinement of the design models to meet the resource and performance constraints on the given execution platform.*

The problem can be divided into a set of related subproblems, including the development of methods for performance modeling, measurement, performance estimation and verification, and performance-aware model transformation. These are delineated as follows:

Performance modeling framework: The modeling framework defines the modeling constructs and their attributes used in performance modeling and analysis. This framework should integrate performance modeling with the ECSW functional model. Such integration allows one to build relations between the system performance and design choices, and hence supports the performance-aware design. The framework should also define the performance associated with the functional models at different design phases to support automatic evolution of performance models along with the functional models as the design advances. As the ECSW functional model is constructed through synthesis of software components, the performance of the software components should be represented in a platform-independent form for reuse and early design performance evaluation.

Performance parameter measurement method: Measurements are essential for quantitative performance analysis. The measurement method must be able to measure the performance metrics of individual components. These measured metrics are used as modeling parameters in performance model construction and analysis. The measurement method should also support the measurements and/or derivations of the performance variances for a measured object under certain interferences derived from the interaction and usage patterns commonly used in an application domain. This is especially useful for the measurement of the underlying supporting software, whose performance depends on the co-existing components and their interactions. To increase the accuracy of the analysis, the measurement method is necessary to minimize disturbance to the measured parameters.

Performance analysis methods: Effective performance analysis methods are required to analyze the performance of the design at various design phases. Given many analysis methods that already have been developed, a major issue in this area is to find an effective analysis method for each design phase, and to make the models fit the assumptions of the analysis. The analysis methods should generate detailed results so the designer can identify the cause of a performance failure, if detected. The methods should be scalable in order to handle a large number of software components with complex interactions.

Performance-aware design methods: Performance-aware design methods use the analysis results to guide the design to meet the performance constraints. The methods should explore the design space, rank different design choices, and indicate results and effects of each choice using the given functional and performance models. The methods should also automatically transform an earlier-stage design model to a later-stage design model without altering its performance. During the transformation, the methods must consider multiple resource constraints and minimize the resource consumption. Such methods must be scalable to be applicable to the ECSW that has a large number of software components and interactions.

Our solution to the performance modeling framework is based on performance annotations. The functional design models in our framework include the software components' structures, the software architecture, the platform configurations, and the runtime deployment model. These models are constructed at different design phases, and their relationships are captured in the framework. These design models are then annotated with a set of performance modeling parameters. To support early analysis and reuse, we introduce the virtual resource concept, and model the performance in terms of virtual resource demands before the execution environment of the ECSW is determined. Our performance measurement method is sampling-based end-to-end measurements, which combines synthetic workloads and microbenchmarks. The sampling-based end-to-end measurement method collects measured information as an external observer, thus minimizing intrusiveness in the measurements. Synthetic workloads allow one to measure the performance variances under different configurations, while microbenchmarks allow one to measure individual metrics and components. Our methods focus on the performance analysis of the software architecture at an early design phase and the performance analysis of the runtime model. The solution to the performance analysis of the software architecture is based on bound estimations and total system workloads. Bound estimations allow analysis without having to know the platform configuration and software deployment. The solution to runtime performance analysis is based on existing schedulability analysis methods. The solution to the performance-aware design method is based on model transformation using graph partitioning and merging, subject to the performance constraints. Such transformation follows the relation between functional models at different design phases and their corresponding performance defined in the modeling framework. To this end, the performance modeling and analysis framework is the centerpiece of our solution. All design and analysis methods are built around this framework.

To show the effectiveness of the techniques developed, we further investigated how designs using other modeling frameworks can be translated into ours. This will enable our methods to be used with different modeling languages.

1.2.2 Summary of contributions

In this research, we developed a performance modeling and analysis framework and a set of analysis and transformation methods. Our performance modeling framework integrates performance models with functional design models at different design phases through annotating performance parameters to software components, software architectures, platform models, and runtime models. The innovations of the framework include both the relationship definitions of design and performance models for different design phases, and the virtual resource for platform-independent performance modeling. The model relationship definitions are the bases for multi-stage performance model construction and analysis, and performance-aware design. Performance representations in virtual resources are essential for reuse and early-design-stage performance analysis.

A major contribution to the performance analysis is the leverage of analysis to an early design phase with an incomplete design model. The analysis method is based on bound estimations. Existing performance analysis methods require knowledge of the complete design, including software execution environments and deployment. Our solution, on the other hand, requires only the software architecture with demands in virtual resources. We showed the estimated performance is useful for software architecture design comparisons and for the platform design. Another contribution to the performance analysis is to have a set of scalable methods that determine runtime model parameters and performance. These methods bring the performance analysis to the design model and architecture level, instead of performance tuning at the implementation level. At such a higher level of abstraction, both design quality and optimality can be improved by exploring more design choices. This set of methods further avoids costs resulting from belated performance error detection and labor-intensive simulations-based evaluations.

Another contribution of this research is the development of an innovative method for runtime model generation. The method consists of a multi-step process that transforms an early design model to a runtime model in a way that both resource constraints and performance constraints are met. We show the conditions leading to correct transformation at each step, and use them to guide the model transformation process.

Our contribution to performance measurement is the method that can measure the metrics of interest for individual components. The method uses the sampling-based end-to-end measurement with a combination of synthetic workloads and microbenchmarks. It supports the measurements of both application components and underlying supporting system services. In this work, we also demonstrate how to use the method by measuring selected real-time operating systems. The results obtained can be used directly in the analysis of the systems using these operating systems.

Finally, the software design toolkit implemented in this dissertation contributes to current ECSW design and analysis. The toolkit provides a graphic interface and a modeling environment for the designer to use the techniques developed in this research. The performance analysis and model transformation results generated by our methods are also visualized in the tool.

1.3 Organization of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 presents our performance modeling framework. The modeling methods support both the refinement and the realization hierarchies. The software components are modeled as process-oriented port-based objects with a set of annotated performance parameters, including both resource demands and performance constraints. With this framework, the initial ECSW design is described in a structural model and consists of inter-communicating software components. The performance constraints are specified in an end-to-end form. The execution environment of the designed ECSW is described in a platform model consisting of computation devices, communication links, and supporting software. Each device in the platform is annotated with a set of resource characteristics such as overheads and capacities. The final ECSW design that can be implemented on the given platform is modeled in a runtime model, which contains a set of tasks and their inter-communications. Similar to the structural model, a set of performance parameters are annotated to the tasks and their inter-communications in the runtime model, which can be derived from the performance of the structural model.

Chapter 3 presents the performance analysis method for estimating early-design phase performance based on the ECSW structural model. Such an analysis assumes that there is no knowledge of the platform. The analysis is based on the bound estimations derived from the performance of constituent components in the structural model. Methods of estimating end-to-end response time bounds and system workload are developed. Since finding optimal performance bounds is NP-hard, our estimation method uses a heuristic to improve its scalability. The analysis results can then be used for the platform configuration design.

Chapter 4 presents the method for performance-aware model transformation from the structural model to the runtime model. The transformation is done while meeting all resource and timing constraints at the same time. Our transformation method uses the graph partitioning technique. It first allocates the components in the structural model to the computation devices in the platform according to the designer-specified allocation strategy. The system-level end-to-end timing constraints are then distributed over the components to be used for constructing a feasible schedule. Finally, the components are merged into tasks while taking into consideration of both schedule flexibility and resource consumption minimization.

Chapter 5 analyzes resource usage and performance of the runtime model. The analysis includes both schedulability and end-to-end timing analyses. In this dissertation, we assume that the system software on a platform supports only priority-based scheduling. The analysis, therefore, requires priorities be assigned to tasks in the runtime model. These priorities are assigned using a heuristic method. After the priorities are assigned, the timing and schedulability analyses of tasks are performed for each device in the platform, while also considering the tasks' runtime properties such as dependencies, release offsets and deadlines, and invocation periods. In case some task fails the above test, we apply a method of end-to-end timing to the task chain to obtain a tighter bound of the end-to-end response time.

Chapter 6 presents a method for measuring performance characteristics. The method is applicable to the measurements of both application components and system software services. It introduces the end-to-end measurement, combining synthetic workloads and micro-benchmark techniques. The collected data are then converted to a platform-independent format and stored along with the application components and system software services. This method has been applied in the measurements of some selected RTOS services, showing its effectiveness.

Chapter 7 shows an ECSW design toolkit with the developed techniques built into it. The toolkit, called AIRES, consists of a graphic generic modeling environment as a user interface, a meta-model implementing the modeling framework, and a set of interpreters implementing the analysis and transformation algorithms. The usefulness and features of the toolkit are demonstrated in the design and analysis of a simplified automotive electronic throttle control software.

Chapter 8 states the conclusions and potential future work of this research.

CHAPTER 2

Performance Modeling Framework

This chapter presents the performance modeling framework, based on which we develop model-based ECSW design analysis techniques. The framework is defined by considering modeling and analysis requirements for applications at various design phases, the availability of different underlying system service support, and the information commonly required by existing analysis algorithms. The models in our framework include software component models, software structural models, platform models, and runtime models. For each model, the framework defines a set of performance parameters annotated to the functional modeling constructs. The thus-defined framework not only supports the performance modeling and analysis at various design phases, but also captures the information flow among the tools used in ECSW design.

2.1 Modeling Process and Method

A modeling framework is generally closely-coupled with the modeling method in a design process. As the ECSW design is a multi-discipline activity, the design process is strongly related to the control design, and includes the following phases [21].

Control design. At this phase, the design focuses on continuous time.

Control discretization. At this phase, the continuous control is discretized with the models restricted to discrete-time/discrete-event blocks. Control discretization also provides the names and characterizations of the signals passed among blocks. The performance constraints are defined in this phase.

Software design. At this phase, the software architecture is determined in order to implement the control models. The design software architecture implements only the control functionalities. No performance constraints are considered at this phase.

Platform design. At this phase, the execution environment for the software is constructed, including hardware and support system software.

Deployment design. At this phase, the software architecture is augmented with execution locations, invocation properties, and scheduling policies.

System implementation. At this phase, the software is coded and loaded onto the target for execution.

In this process, the steps of software design, platform design, and deployment design have great impacts on the final system performance, and thus require performance analysis. Specifically, the software architecture determines the components and their inter-communications, which implement the control blocks and the signals in the control model, respectively. The resource demands and delays of the components and communications will affect the the control performance of the final system. The platform design decides the capability of the platform and the available services for runtime resource management. Whether the software can achieve the control performance or not depends heavily on the platform. The deployment design describes how the software should be organized on the platform if there exists a design to meet the performance constraints. As our research focuses on the ECSW design under the assumption that a correct control design is given, our performance modeling framework targets only the performance analysis at software design, platform design, and deployment design phases.

To align it with the current ECSW design, our framework is defined to be used with two layered modeling methods: the *refinement modeling method* for the software architecture design and the *realization modeling method* for the deployment design. As both methods are used for functional design, we use annotation as the method for performance modeling in this framework.

2.1.1 Refinement modeling method

The model constructed using the refinement modeling method describes the ECSW at different levels of abstraction, called layers. The model at a higher layer is more abstract, and can be refined with more implementation details to generate a lower-level model. Figure 2.1 shows an example of the model construction for a machine control system using the refinement modeling method. In this example, the machine controller software consists of two subsystems at the highest modeling layer, *machine control (MC)* and *process control (PC)*. The MC subsystem communicates with the PC subsystem to make decisions on how the axes of the machine should move. At a lower layer, each subsystem is refined with several components. The MC subsystem is refined with the axis coordination control component (AxisGroup) and three independent axis control components (Axis), while the PC subsystem is refined with a force acquisition component (ForceAcq), a force supervisory control component (ForceSup), and a tool broken detection component (ToolDet). Meanwhile, the communication between PC and MC is also refined as two connections: one from ForceSup to AxisGroup and the other from ToolDet to AxisGroup.

The refinement stops at a predetermined layer when the models are partitioned to the blocks in a predefined granularity. These models can then be implemented using existing software components

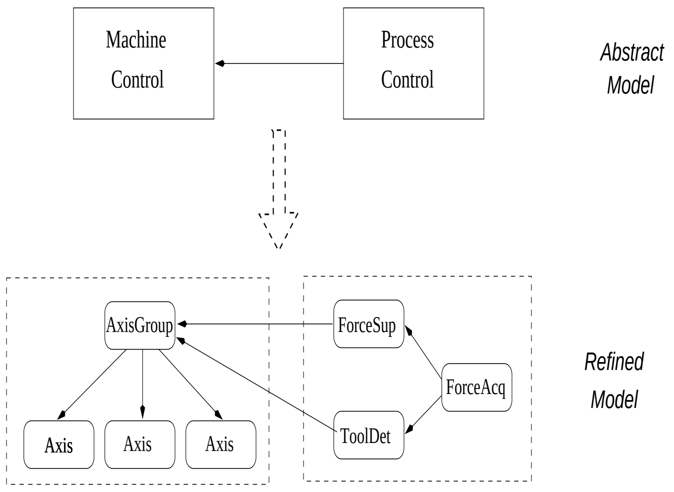


Figure 2.1: A machine control model constructed using the refinement modeling method.

and a *describe-and-synthesize* methodology [23].[1] The granularity and construction methods of these software components are highly domain- and organization-dependent, and beyond the scope of this research. Examples of such components include various device drivers, data processing components, and control algorithms. The resultant model at the end of the refinement describes the software architecture without runtime and deployment information. The component in the resultant model can be implemented either by coding in a programming language or by reusing the code from other applications/libraries. In this dissertation, we call the final refinement model a *structural model*.

2.1.2 Realization modeling method

A model constructed using the realization modeling method describes the relationship between the ECSW and its execution environment. It models the ECSW from the system perspective instead of from only the software perspective. A realization model is also a hierarchical model. Different from a refinement model where the model at a lower layer describes the same system as the model at

[1]The method was first developed for small-scale embedded systems. It combines the concepts of both component-based design and software-hardware co-design. As our work deals only with software, it becomes a component-based design method.

a higher layer but with more implementation details, the model at a lower layer in a realization model defines an execution environment for the model at a higher layer. An example of the realization hierarchy used in ECSW design is shown in Figure 2.2.

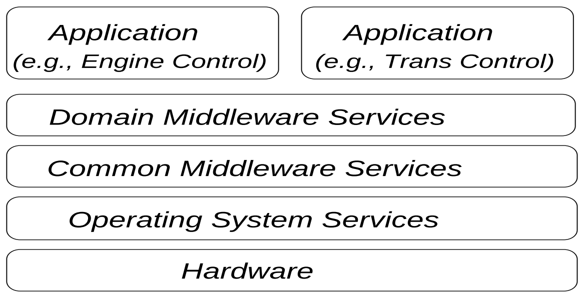

Application
(e.g., Engine Control)
Application
(e.g., Trans Control)
Domain Middleware Services
Common Middleware Services
Operating System Services
Hardware

Figure 2.2: The modeling hierarchy used in the realization modeling method.

In a realization model, the executions of high-level components require the services of low-level components. These low-level services in a realization model form a virtual machine for the high-level components. Once the interactions between different layers are defined, the models of different layers can be designed and implemented separately and in parallel. With the realization modeling method, the models of software architecture and of the platform are designed separately and integrated to generate the model of the system at the deployment design phase. The generation of deployment can also be viewed as a refinement process, where the component in a structural model is refined with the information of the execution location and the scheduling policy. In this dissertation, the model generated at the deployment design — the one which models the software allocated on the designed platform and describes the system runtime architecture — is called a *runtime model*.

2.1.3 Performance modeling method

A key requirement of the performance modeling method in our framework is to relate the modeling parameters of performance to the functional model. In our framework, we use performance annotation as the modeling method. This method attaches a set of performance modeling parameters to the modeling constructs in the framework. The annotation method is more efficient and scalable for performance model construction of a large system than other methods, such as modeling performance parameters as modeling constructs [4]. The annotation method allows the performance model to evolve along with the functional design advances.

The performance modeling parameters annotated to the functional models are classified into two disjoint categories: *constraints* and *characteristics*. Such classification is employed to suit the refinement and realization modeling methods for the functional model, and to support the performance reuse. The constraint parameters define the performance that the system must meet to guarantee the correctness of the control. Examples of the constraint parameters include the invocation rates and deadlines of control flows, and the distance of consecutive outputs. The values of the constraint parameters are application-dependent and are typically imposed by the designer according to the results of control discretization. For example, the invocation rate of a component is a constraint parameter. The invocation rate for the throttle controller is a constant for a given model of vehicle, but it varies from one vehicle model to another.

The characteristics parameters define the performance of a modeled component. At the same time, they are the properties of the component. The characteristics parameters considered in this dissertation include various resource demands of the software components and the system, including computation resource demands (typically in execution times), communication demands (in end-to-end delay or message size), and memory demands (in code and data size). The values of the characteristics parameters depend on the implementation of the component and, sometimes, its execution environment. For example, the computation resource demand of a throttle controller depends on how the controller is coded and the number of instructions in the final executables.

The annotations of performance parameters allow the performance model to evolve along with the functional design model. With the software model constructed using the refinement modeling method, a higher layer model is built first. Due to the abstraction of the higher layer model, only constraints are known for the model. As the model is refined, the constraints are partitioned and distributed over the lower layer models, thus reducing the overhead of regeneration of performance modeling information. During the analysis, the derived constraints are used to compare with the performance characteristics of the model at a refinement layer. Since the performance characteristics can only be known after all components are determined at the end of refinement process, the performance analysis can only be performed using the structural model.

The performance model of ECSW implementation can be constructed using the annotated performance parameters of both application components and environment components. Specifically, the values of the performance characteristics of a component can only be determined after its execution environment is decided. Thus, in the realization layer model, the performance characteristics of a higher layer component can be derived from the resource demands of the component and the performance characteristics of the lower layer services for the component's execution.

With the above discussed performance annotation, performance-aware design and performance reuse can also be achieved. When a performance failure is indicated in the analysis, a performance model constructed with such a method allows the designer to pinpoint the cause in the software structure, platform configuration, and/or runtime property assignments, and to suggest corresponding design alternatives. Further, as the performance constraints are application- and product-dependent, and the performance characteristics are platform- and implementation-dependent, these parameters and their values can be reused with the components in different ECSW designs if these designs have similar control requirements and component implementations (both environment and code).

Note that the performance parameters are dedicated to the modeling constructs. In other words, the components in structural models, runtime models, and platform models have different performance modeling parameters annotated to them. In the following sections, we will discuss these models and their annotated performance parameters in detail.

2.1.4 Model relationships

Our modeling framework defines the functional models used for different ECSW design phases (after control discretization and before coding), their relationships, and their associated performance. Figure 2.3 shows the relationships between these functional models and their performance models.

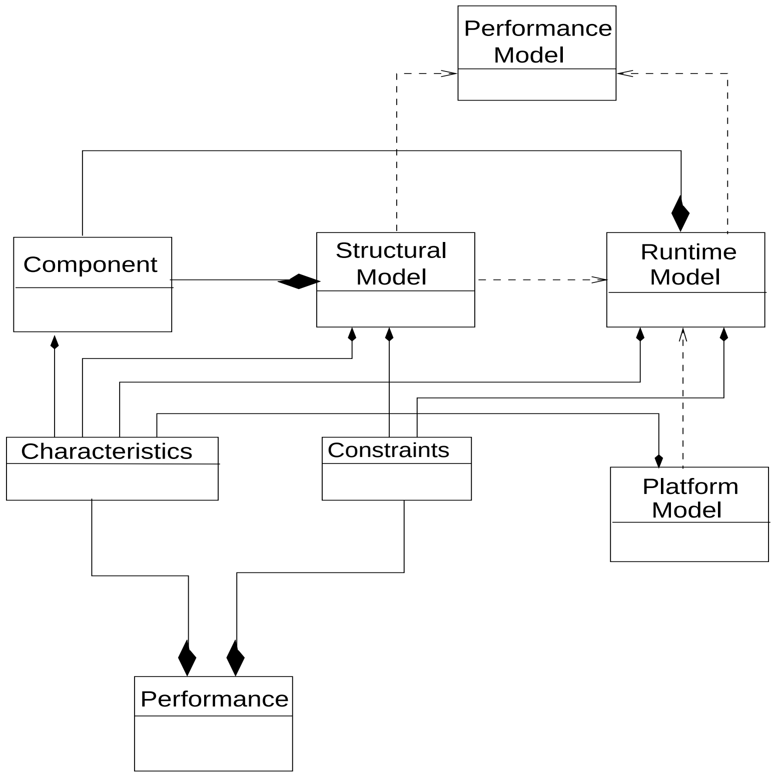


Figure 2.3: Relationships of the models in our framework.

In this framework, software components are basic building blocks with a common structure. The software architecture of a designed ECSW is described in a structural model consisting of interconnecting components. The execution environment of the ECSW is described in a platform model that includes both hardware and supporting software. The deployment and runtime organization of the ECSW is described in a runtime model, which is derived from the structural model and the platform model and contains the same set of components as the structural model. All these functional models are annotated with performance, including characteristics and constraints. Performance models are derived using the corresponding functional models and the annotated performance. Because a structural model is constructed using software components, its performance model depends on the components' performance. Similarly, because a runtime model is derived from a structural model and a platform model, its performance model must contain the performance derived from the performance of the structural and platform models.

Within the scope of this work, performance constraints are only required for the structural and runtime models in performance modeling and analysis. This is different from performance characteristics, which are needed for all models. In general, however, all models can have both performance characteristics and constraints specified in the modeling and analysis. Also, for the purpose of this work, we have constructed performance models for only the structural and the runtime models, as they are the most critical design models for an ECSW. Details of these models are given in the following sections.

2.2 Software Component Model

A software component is a basic building block used to construct the ECSW, and is contained in the lowest layer of a refinement model. In our modeling framework, a software component is formally modeled as:

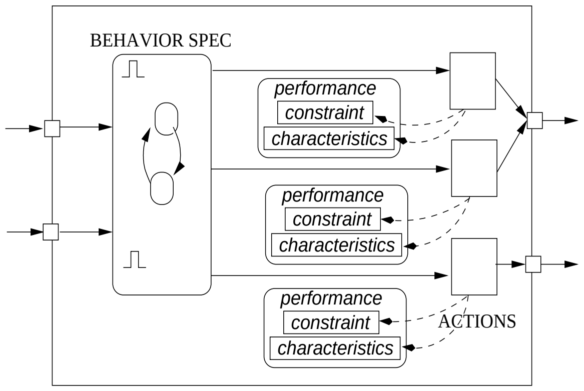


Figure 2.4: Software component model.

Definition 2.1. *A software component M_c is defined as a port-based object*

$$M_c = (A, I, O, B, F)$$

where

- *A is a set of computations, called actions, which implement the component's functionality;*
- *I is a set of input ports through which the component receives its inputs;*
- *O is a set of output ports through which a component exposes its computation results;*
- *$B \subseteq E \times I \rightarrow A^+$ specifies a predefined behavior of the component, and E defines a set of events; and*
- *$F : A \rightarrow \{Q^+\}$ defines a set of functions mapping each action to a set of positive rational numbers Q^+, representing the values of its performance parameters;*
- *$I \cap O = \emptyset$.*

The structure of the software component model defined in Definition 2.1 is shown in Figure 2.4. In this model, a software component contains a set of actions, input and output ports, a behavior specification, and a set of performance parameters. Each action performs some predefined computation that implements some portion of the system behavior. Components communicate through their input and output ports. An input port of a component may receive information from one or more output ports of other components. Similarly, an output port of a component may send results to multiple input ports of other different components. No port can be used for both inputs and outputs in our component model. Note that the ports in the component model are all logical ports. In an implementation, the same communication object can be used for both input and output ports. For example, a shared memory object can be used for both reading (input port) and writing (output port) a data value. This constraint forces explicit modeling of the port types, and hence translates the components to typed components to support type checking in the model construction. Type checking support is essential for automatic system composition and formal analysis.

The behavior specification in the model defines the reaction of the component upon arrival of an event. A behavior contains a sequence of actions of the component, and is triggered only when some event arrives at the input ports. The behavior specification may define multiple behaviors required under different system modes. Thus, a pair $(event, input\_port)$ uniquely determines the behavior that a component is required to perform. The $(event, input\_port)$ pairs of a component can then be viewed as external interfaces, and other components can interact with it in an ECSW construction to invoke a desired behavior.

The performance of a component is modeled as a set of function F annotated to the actions. Given an action $a \in A$, $F(a)$ derives the values of performance parameters of a, including both constraints and characteristics. The constraint parameters are applicable only to the components at the intermediate refinement layer. The component at the lowest refinement layer inherits the constraints of its parent components. For example, a mode switch component may be used for both the 1 Hz monitor process and the 20 Hz manager process in a throttle controller. Its invocation rate does not depend on the implementation of the mode switch component, but is determined by the process that it participates in, that is, the mode switch component runs at 1 Hz in the monitor process, and 20 Hz in the manager process. On the other hand, the monitor component, when viewed as a composite component, must have the constraint parameters of invocation rate and deadline associated with it.

The software component model defined in this framework is *process-oriented*, meaning all behaviors of a component are designed to transform the inputs to the outputs. This implies that all behaviors of a component implement the same or similar functionality but with different input/output parameters (in number and type) and internal algorithms. This differs from the *object-oriented* component models used widely in current software development. As information encapsulation is the goal of an object-oriented model, the core of the object-oriented model is the data/attributes in an object. The behaviors of an object, called *method*, are mainly for data accesses, although the data processing methods may also be implemented. Although process-oriented and object-oriented models are functionally equivalent, we argue that process-oriented models are more suitable for ECSW design and performance analysis since they reflect the nature of the control process, and performance is typically associated with the behavior of a system rather than the values of its attributes. On the other hand, due to the popularity of object-oriented models in current practice, it is more desirable to build the transformation method to map one to the other. In our work, this is achieved by port-dependency graphs during the structural model construction. This will be discussed in detail in the next section.

2.2.1 Component performance modeling

The modeled performance characteristics of software components include resource demands for computation, output data sizes for communication, and code+data size for storage. For a software component used in ECSW design, its storage resource demand can be determined immediately after the component is coded. As the code size of a component does not change during the execution and the required data size can be bounded (with deterministic memory allocations), we treat the storage resource demand of a component as a constant for any platform. On the other hand, the resource demands of computation and communication, in execution times and communication delays, usually depend on platform implementation. In order to support early design phase analysis before the platform has been determined, we need techniques to represent these resource demands in a platform-independent form, and yet reflect the relative amount of resources needed during executions to make the comparison and quantitative analysis meaningful. Such platform-independent values should also be easy to transform to platform-specific values after the platform is designed. To this end, we have introduced the virtual resource concept to represent the values of performance characteristics of components. First, we must define the service rate of the resource.

Definition 2.2. *Let W denote the maximum amount of work that can possibly be done within a time duration D on a given resource res. The* service rate *of res, denoted as $\rho(res)$, is then defined as a uniform uninterrupted rate $\rho(res)$ computed by*

$$\rho(res) = \frac{W}{D}. \tag{2.1}$$

The above-defined device service rate is platform-dependent. Given the same type of devices (e.g., processor or network), the workloads completed by individual devices within the same duration can be different due to the difference in their capacities. Therefore, the service rates of different devices can vary significantly. To remove the service rate differences among these devices, we artificially define a service rate shared by all devices, called a *virtual service rate*. A device running at the virtual service rate is called a *virtual resource*. The resource demand of a component can therefore be modeled as the time required to complete the workload on a dedicated virtual resource. Such a resource demand is called a *virtual resource demand*. With the virtual resource concept, the service rate differences of the devices are transformed to the capacity differences. In other words, a device with a fast service rate can be viewed as containing a greater number of virtual resources than a device with a slow service rate. For example, devices A and B have the service rate $\rho(A) = 5$ and $\rho(B) = 3$. Assume a virtual service rate $\rho(v) = 2$. Then A contains 2.5 virtual resources, and B contains 1.5 virtual resources. Suppose two components, c_1 and c_2, both with a virtual resource demand 1. Then c_1 and c_2 can run together on A, but only one can run on B.

The virtual resource demand of a component can be derived from measurements or profiling. Given a measured execution time $e_c(p)$ of component c on a device p with service rate $\rho(p)$ and virtual service rate $\rho(v)$, the virtual resource demand $w(c)$ of component c can be computed as:

$$w(c) = \frac{e_c(p) \cdot \rho(p)}{\rho(v)}. \tag{2.2}$$

In Eq. (2.2), both the device service rate $\rho(p)$ and the standalone execution time of c on p, $e_c(p)$, can be measured. Particularly, $\rho(p)$ can be measured statistically by running various representative synthetic workloads on p. Thus-derived $\rho(p)$ includes all factors such as processor architecture and clock speed, memory architecture and size, and resource management overheads. The method used for such measurements is detailed in Chapter 6.

Similarly, we can define the virtual resource for communication links. The virtual resource demand of a message over the communication link can also be represented independently of the physical link, and can be derived using the measured message transmission time $e_m(l)$ over a link l, the service rate (link speed) $\rho(l)$, and an artificially selected virtual service rate $\rho(v)$. Since the communication workloads of a message—the data size—are typically known, the resource demands of a communication is commonly modeled as its message data size, which implies the $\rho(v) = 1$.

The computation devices and communication links are different types of resources generally running at different service rates. The virtual service rates for these resources should also be different to reflect the realistic scenarios. In this analysis, however, these resources need to be treated uniformly to provide certain results; for example, to identify a bottleneck. To this end, we need to build a relationship between the virtual service rates of computation resource and communication resource. This is done through a *computation-communication service ratio*, δ, which is defined as:

$$\delta = \frac{\rho(comp_v)}{\rho(comm_v)}$$

where $\rho(comp_v)$ and $\rho(comm_v)$ are the virtual service rates of computation virtual resource and communication virtual resource, respectively.

It is critical to obtain δ close to the real target platform for accurate analysis. The closeness of δ to the target platform determines the accuracy of performance analysis results, and consequently, the quality of the design model, if the design follows the guidance of the analysis results. By selecting different values of δ, a designer can test and evaluate many ”what-if” scenarios of both software architecture and platform configuration.

In this dissertation, we assume the software components are basic building blocks whose performance characteristics are known. The constraints of a component, on the other hand, can only be derived after the software architecture is determined. Therefore, in our later discussion, F of a software component will refer only to its performance characteristics.

2.3 Structural Model

The designed ECSW is first modeled as a set of inter-communicating software components called a *structural model*. A structural model implements the control model generated at the control discretization phase. With the structural model, a designer can capture both functional and performance design flaws at an early design stage through analysis, and therefore avoid expensive redesign. Since software architecture has more impact on system performance than component-level code optimization, a design can be further optimized at the system level using the structural model. Because the behaviors of an ECSW are typically modeled as information processing flows derived from the control design, a structural model of the designed ECSW can be viewed as an aggregation of information processing flows, called *transactions*. Formally, a *transaction* can be defined as a weighted directed graph as below.

Definition 2.3. *A* transaction *is defined as a weighted directed component graph,*

$$M_x = (C, L, F, H)$$

where

- C *is a set of software components in* M_c*, each of which is represented as a node in the model;*
- $L \subseteq \bigcup_{a \in C} O_a \times \bigcup_{b \in C} I_b, (a \neq b)$ *is a set of directed links representing partial relationships from the output ports of some component(s) to the input ports of some other component(s);*
- $F : C \cup L \rightarrow {Q_0}^+$ *defines a set of functions for performance characteristics.*
- $H : R \cup O \cup D$ *defines a set of performance constraints.*

In a transaction, the nodes are components defined in a software component model. Each component performs some predefined behavior(s) that contributes to achieving the overall system behaviors defined in the control design model. A link in a transaction indicates the information dependency between two components. A link is valid if it connects an output port of one component to an input port of another component. No link between an input and an output port of the same component is allowed. This implies that all communications among the ports of the same component should be implemented internally. Since a transaction models an information processing flow, the links in a transaction model must be *synchronous*, meaning the arrival of an event/data/message over the input links of a component triggered the behavior of the component. Any component with multiple input links is triggered when all its inputs have arrived (an AND operation). Such a component is also called a *synchronization component* in some literature [84]. In a system, it is possible to have a component providing some common service. Such a component is called a *shared component*, and its execution can be triggered upon arrival of any input. In our structural model, we duplicate a shared component and its downstream successors in every transaction that involves the shared component. Such duplication yields an equivalent model of one containing shared components with OR links.

In the transaction model, a component without any input link is called an *input component*. An input component models a start point of the transaction. Typical input components include the drivers of input devices, such as sensor reading and/or status monitoring. The input components can be triggered by signals from its environment, including interrupts or time-out signals from underlying support system software. Similarly, a component without any output ink is called an *output component*. An output component models an end point of the transaction. Typical output components are the drivers for output devices, such as actuators. Other components with both input link(s) and output link(s) are called *processing components*. The execution of a transaction starts from its input component(s) and ends at its output component(s).

2.3.1 Structural model performance modeling

The performance parameters of a transaction include both constraints and characteristics. The performance characteristics can be derived from the resource demands of the constituent components. We consider three types of resources in the transaction performance characteristics function F:

- Computation: $F_c : C \times B \to Q^+$. This defines the computation resource demands of the components under different behaviors. B is the set of all behaviors of the component set C. Q^+ is the set of non-negative rational numbers. The resource demand of a component depends on the resource demand of its actions in a behavior. Suppose a component c performs a behavior $b \in B$ implemented as a sequence of actions $a_1, a_2, ..., a_n$, its resource demand can be determined by

$$F_c(c, b) = \sum_{i=1}^{n} a_i$$

- Communication: $F_l : L \to Q^+$ defines the communication resource demands of the links.

- Memory $F_m : C \to Q^+$ defines the memory resource demands of the components.

The components' resource demands of computation and memory are annotated to the corresponding nodes as node weights, while the communication resource demands are annotated to the links as link weights. The resource demands of a transaction can then be computed by traversing the graph of the transaction model.

Since a transaction models the ECSW at a higher layer of the refinement model, the performance constraints are applicable for the transaction. Such constraints are specified as a set of end-to-end constraints, H, of each transaction, including:

- Invocation rate r: defines the frequency at which the transaction is invoked. Since the communications within a transaction are synchronous, the components of a transaction share the same invocation rate. As a transaction starts from its input component(s), the invocation rate is typically modeled as its input rate.

- End-to-end deadline D: defines the time duration within which a transaction must be completed. The end-to-end deadline bounds the time duration between the earliest start input component and the latest finished output component.

- Input separations S: defines the bound of timing difference between a pair of inputs. The input separations are typically defined as the release phases of different input components of the transaction.

- Output jitter constraint J: defines the bound of timing difference between two consecutive generations of the same output.

Note that Definition 2.3 allows existence of cycles in a transaction. This is desired in order to facilitate modeling control functions like closed-loop feedback control and multi-rate control. These controls commonly exist in many embedded control systems. As many analysis methods can only be applied to directed acyclic graphs, we have developed a method to eliminate the cycles and transfer the transaction model to an acyclic one. This method is discussed in Chapter 3.

Given the transaction defined as a weighted directed graph, a *structural model* of an ECSW design can be defined as a set of mode-dependent transactions.

Definition 2.4. *A* structural model *is defined as a transaction set active under all system modes,*

$$M_s = (m, T, \Phi)$$

where

- *m is a set of system execution modes;*

- *T is a set of transactions modeled in the transaction model M_x;*

- *$\Phi : m \to T^+$ is a function mapping a system mode to a set of transactions. Given $m_i \in m$, $\Phi(m_i) = \{T_i^1, \ldots, T_i^n\}$ lists the transactions that are active in mode m_i.*

It can be seen that the structural model contains a three-level hierarchy. The top level is the system structure consisting of a set of transactions. The middle level is a set of transactions that model the end-to-end system behaviors. The low level contains inter-communicating components that form the transactions.

2.3.2 Translation of models in UML

The components used the structural model are process-oriented. In current practices, many software systems are modeled using Unified Modeling Language (UML) [66], which is becoming a standard modeling language for software design. Components in such models are typically data-centric and object-oriented, and are not suitable for system-level analysis. With UML-like modeling languages, the system behaviors are usually modeled as some interaction diagrams, such as collaboration diagrams and use-scenario diagrams. These models must be converted to one compliant with our structural model definition in order to use the developed model-based analysis and design model transformation methods. We solve this by the construction of a *port-dependency graph* (PDG).

Definition 2.5. *A* port dependency graph *is a directed graph $M_{PDG} = (V, E)$ where*

- V *is a set of ports, which can be either data communication ports or method calls;*
- E *is a set of directed links among the ports. A link can be either a dependency between the ports of different objects, or a dependency between ports of the same object.*

In a UML-like model, each port of an object is uniquely linked to an operation/method call. Thus, the ports of a PDG can be treated as the process-oriented components. On the other hand, a UML-like object contains links both to other objects and among its different ports. The process of the PDG construction of a UML-like model is as follows:

1. For each method/function of a UML object in the model that is invoked by other object(s), create a node in the PDG.

2. For each precondition a PDG node needs to satisfy for its execution, including both events and data, create an input port for it.

3. For each result a PDG node generates, including both events and data, create an output port for it.

4. For each invocation link in the original model, create a link between corresponding ports of nodes in PDG.

5. If there exist multiple events and data links between a pair of nodes in PDG, merge them into one.

6. Repeat the process for each mode.

A thus-obtained PDG is a model with process-centric components. The resource demands of different methods of a component in the object-oriented model are generally different. The resource demands of a method can be annotated to the port in the PDG after the conversion. It is easy to see that the thus-obtained PDG model is a structural model defined in our framework.

2.4 Platform Model

A platform consists of components that support the execution of the designed ECSW. It models the execution environment of the ECSW with the required resources. The components in a platform model include hardware components such as processors, memory modules, and network links, and system software such as operating systems, network protocol stacks, and middleware. In our framework, a platform is modeled as a collection of communicating devices in a weighted undirected graph.

Definition 2.6. *A* platform model *is a weighted undirected graph,*

$$M_p = (PROC, NW, R)$$

where

- $PROC$ *is a set of computation devices providing computation and storage resources;*
- NW *is a set of communication links connecting the computation devices;*
- R *is a set of "availability" functions of the modeled resources.*

In the platform model, the nodes represent a computation device. A computation device consists of a processor, memory, and the support system software (e.g., operating system) running on it. In the analysis of a model at a higher realization layer, all models of lower layers are considered as a whole forming the environment. This implies that we treat the hardware and the system software on it as a whole in the application performance analysis. Each link in a platform model represents a communication link shared by two or more computation devices. We assume that the computation devices are fully connected, meaning that a message from one device can reach every other device. The communication between any pair of devices may go through multiple links and other devices. However, we assume the traffic forwarding by a device on a path does not introduce additional computation overhead for the device. Note that this assumption is true for the communication links with resource reservation and control mechanisms, such as real-time channel [99], CAN bus [98, 101], and RT-Ethernet [50]. Similar to the computation devices, a link and the protocol to manage its transmissions are considered as a whole in the application performance analysis.

The performance of a platform is modeled in a set of characteristics, and they are annotated to the nodes and links in a platform model as weights. According to the realization layer, the characteristics of hardware components and system software components are different, and are considered separately. For hardware components, the performance characteristics model the resource capacity provided by the modeled devices/links, defined by the *resource capacity functions*, R. Here we consider three type of resources with their capacity functions defined as follows.

- Computation resource: $R_c : PROC \rightarrow Q^+$ defines the computation resource capacity of a device in service rate.

- Communication resource: $R_l : NW \rightarrow Q^+$ defines the communication resource capacity of the links in service rate.

- Storage resource: $R_m : PROC \rightarrow Q^+$ defines the memory resource capacity of a device in data size.

The resource capacity functions are annotated to platform components in the model. The functions for computation resource capacity, R_c, and for storage resource capacity, R_m, are annotated as weights to the nodes $PROC$ in the model. The functions for communication resource capacity, R_l, are annotated as weights to the links NW. The heterogeneity of the platform is reflected in these resource capacity functions. Different R_c and R_m of different nodes indicate heterogeneous computation devices, while different R_l for different links indicate heterogeneous communication links.

The resource capacity functions can be constructed through measurements. We can measure a representative workload on a reference platform and derive the service rate of the device using Eq. (2.1). The thus-obtained service rate is fundamentally different from the classic processor or bus speed used in product descriptions since it includes the effects of all activities and interferences during the execution. For example, the service rate of a computation device includes the effects of pipeline, memory operation, and OS activities, and the service rate of a communication link includes the overheads to establish a connection and synchronize the transmissions. The function

service	parameter	symbol
scheduling	overhead	R_{sch_o}
timing	overhead	R_{time_o}
	jitter	R_{time_j}
IPC	overhead	R_{ipc_o}
	delay	R_{ipc_d}
synchronization	overhead	R_{sync_o}
	waiting time	R_{sync_w}

Table 2.1: Performance parameters of OS services.

can be complex as the effects usually vary with the system status.

The system software performance characteristics are modeled as a set of overheads and variances introduced to the system execution. The components of the system software are basic services needed for application execution. Examples of such basic services include timing service, scheduling service, and communication service. The performance characteristics of these services are modeled as functions that derive the performance values based on hardware characteristics and workloads, and annotated to the system software. Table 2.1 lists the commonly required services and their performance characteristics. In this dissertation, to simplify the investigation, we have considered only timing and scheduling parameters, as they are of the most interest in the system performance analysis and performance tuning.

According to previous research [90, 43, 12, 11], the resource functions of these services depend both on the implementation strategies of the operating system, workloads of a service, and the configuration of the services. For example, the scheduling service performance is a function of the number of processes in the system, priority levels, and the invocation mechanisms of the scheduler. The overhead of a timing service depends on the number of processes using the service, while its jitter is a function of the timer resolution, and the processes using the services. As the performance functions of these services are hardware-specific, to model system software and hardware components as a whole is more convenient in the design and analysis. These functions can be obtained using an end-to-end measurement presented in Chapter 6.

2.5 Runtime Model

A *runtime model* specifies the ECSW runtime architecture with all implementation details filled, including both execution properties (such as execution locations of the software components and execution timing properties) and scheduling policies. A runtime model is constructed at the deployment design phase by refining the structural model. Such a model is the final design model before the software implementation. The building blocks in the runtime model must be objects schedulable and manageable by the system software in the platform. Detailed and accurate ECSW performance analysis, specifically timing and schedulability analysis, are applied to the runtime model.

Definition 2.7. *A* runtime model *is defined as a weighted directed acyclic graph*

$$M_r = (\tau, L, P, D, o, w, loc)$$

where

- τ *defines a set of tasks that are basic schedulable execution units in the system.*
- $L : \tau_i \preceq \tau_j (i \neq j)$ *defines the precedent constraints between tasks.* $\preceq$ *is a partial ordering relationship, indicating that either the execution of* τ_j *depends on the execution of* τ_i*, or* τ_i *and* τ_j *have no dependency and can run concurrently.*
- $P : \tau \rightarrow Q^+$ *defines a set of invocation periods for* $\tau_i \in \tau$*. The value of* $P(\tau_i)$ *can be any positive rational number (*Q^+*). For a task* τ_i *whose period is unassigned,* $P(\tau_i) = \infty$*. Otherwise,* $P(\tau_i)$ *defines the duration between two consecutive invocations of* τ_i*. If* τ_i *is invoked periodically,* $P(\tau_i)$ *is a positive constant (denoted as* P_i*). If* τ_i *is invoked sporadically*

or aperiodically, $P(\tau_i)$ is a function for the intervals between consecutive invocations of τ_i.[2]

- $D : \tau \times \tau \rightarrow Q^+$ *defines a set of deadlines. Given a pair of tasks τ_i and τ_j, the deadline can be any positive rational value. If the deadline between τ_i and τ_j is unassigned, $D(\tau_i, \tau_j) = \infty$. If the deadline is assigned, all tasks between τ_i and τ_j in a dependency chain must complete within $D(\tau_i, \tau_j)$, denoted as D_{ij}.*

- $o : \tau \rightarrow Q_0^+$ *defines the released offset functions, which is a non-negative rational value. The value of $o(\tau_i)$ indicates the time duration between the readiness of τ_i and the beginning of τ_i's invocation period.*

- $w : \tau \cup L \rightarrow Q^+$ *defines a set of resource demands of tasks τ and links L, whose values are positive rational numbers. The resource demands in a runtime model are in real execution times instead of the virtual resource demands as the devices/links for the tasks/communications have all been determined.*

- $loc : \tau \rightarrow PROC$ *defines a function that maps each task to a proper computation device. Since we assume only one link between a pair of devices, the links for task communications are consequently determined after loc function is decided.*

A task in the runtime model contains a sequence of components in the structural model, and can typically be implemented as a process or thread in the system software (e.g., OS) of the platform. We formally define a task as follows:

Definition 2.8. *A* task τ_i *is modeled as a tuple with*

$$\tau_i = < \Psi_i, P_i, d_i, o_i, w_i, loc_i >$$

[2]As the analysis of real-time and embedded systems usually considers only the worst case, the minimum interval between invocations can be used as P for sporadic and aperiodic tasks.

where Ψ_i defines a sequence of components executed in a first-in-first-out (FIFO) fashion with run-to-complete semantics, P_i is τ_i's invocation period, d_i is τ_i's relative deadline, o_i is τ_i's release offset, w_i is τ_i's resource demands, and loc_i is τ_i's execution location.

The periodicity of a task depends on the invocation properties of the components in the task. Some tasks can run as pseudo-periodic tasks, whose invocation depends on some periodic tasks. The links in the runtime model specify the dependencies between the tasks. These dependencies include both data and control dependencies. All dependencies are implemented as synchronous messages.

Dependent tasks in a runtime model form an information processing flow, called *task chain*. The task chains implement the transactions in the structural model. A runtime model contains a 4-level hierarchy. In the hierarchy, the top-level system model contains a set of task chains, which consist of dependent tasks, which consequently consist of software components. The task chains may communicate with each other asynchronously. The tasks in a task chain are dependent and communicate synchronously. The components inside a task execute sequentially and share the same scheduling parameters (e.g., scheduling policy and priority) of the task. In this work, we assume statically constructed tasks, meaning that the components in each task are statically assigned and do not change during task execution. The task execution location and scheduling properties are also statically assigned.

2.5.1 Runtime model performance modeling

The performance parameters of the runtime model include both performance constraints and characteristics of the tasks/links. The performance characteristics of tasks and their communications are annotated as node weights and link weights in the runtime model. The characteristics parameters include the resource demands of various resources. Similar to the structural model, we consider the following three types of resources for the runtime model:

- Computation resource $w_c : \tau \times PROC \rightarrow Q^+$ specifies the tasks' computation resource demands. Typically, we use the service times (execution times) in wall-clock time for this parameter.

- Communication resource $w_l : \tau \times L \rightarrow Q_0^+$ specifies the communication resource demands for the links $l \in L$. Typically, we use the message delays excluding contentions between two nodes for this parameter. Such a delay includes the overheads for the channel establishment and the disconnection.

- Storage resource $w_m : \tau \times PROC \rightarrow Q^+$ specifies the tasks' storage resource demands. We assume the memory (both ROM and RAM) is the only storage on a computation device. The resource demands are modeled as the total size of the code and data (including dynamic data).

The resource demands of a task depend on the total computation and communications of the task's constituent components. According to the Eq. (2.2), we can compute the resource demands of a task, both computation and communication, as follows:

$$\begin{aligned} w_c(\tau_i, PROC_p) &= \frac{\sum_{a \in \tau_i} F_c(a) \cdot \rho(v)}{\rho(PROC_p)} \\ w_l(\tau_i, l) &= \frac{\sum_{a \in \tau_i} F_l(a) \cdot \rho(v)}{\rho(l)}. \end{aligned} \tag{2.3}$$

The constraints parameters in the runtime model include invocation periods, deadlines, and release offsets. The invocation periods are defined for the *input task(s)*, which have no predecessor, in a task chain. The invocation periods can be propagated to every pseudo-periodic task along the synchronous communication links. The deadlines include both end-to-end deadlines and relative task deadlines. The former constrains the execution duration between an input task and an *output task*, which has no successor in the task chain. The latter constrains the execution of individual tasks in the task chain. The release offsets constrain the start time of a chain as well as individual tasks in the chain. Typically, these constraints are inherited from the end-to-end constraints in the structural model. As many analysis methods require the constraints of every task, such end-to-end constraints may need to be distributed over the tasks in a chain. For example, the tasks' relative deadlines and release offsets can be derived based on the end-to-end deadline D using techniques such as deadline distribution [38]. Similarly, the period of pseudo-periodic tasks can be obtained using the rate propagation.

In our runtime model, we allow for the invocation period of a task chain to be shorter than its end-to-end deadline, i.e., $P < D$. This implies that multiple instances of a task chain can be active in a system. The executions of these multiple instances must be pipelined on different computation and communication devices in the designed platform with total resource capacities that are sufficient to support such a pipeline execution.

2.6 Related Work

Many modeling frameworks have been proposed for modeling and analysis of software performance. The one proposed by Kevin Bradley [11] is called the real-time modeling framework. The framework contains modeling constructs for describing all aspects of the software design model, partitioned into application model constructs and target-platform model constructs. The framework contains a modeling hierarchy, from top to bottom, of domain, architecture, component, and physical models. Although this framework is similar to our modeling framework, the modeling information and method used to derive them are fundamentally different. A model constructed using the framework in [11] assumes the application and platform models are constructed independently, and the analyses are only performed after the application model has been allocated to the target. Therefore, there is no need to model the performance of the application model and components before they are allocated to the target, and consequently no analysis can be performed before then. Such a process significantly limits the scope of the performance modeling and analysis, making it difficult to support early performance modeling and true performance-aware design.

The *Ptolemy* modeling environment [53] uses a different modeling framework consisting of *actor* and *actor frameworks*. An actor is a reactive component and can perform a set of partially-ordered computations and communications. An *actor framework* defines an environment for an actor and controls its executions and interactions with other actors. An application can contains multiple heterogeneous models consisting of different frameworks and actors organized hierarchically. The performance analysis of such a system model is based on design-level simulation without the platform information. Such analysis is only useful for control design evaluation. The runtime model in *Ptolemy* uses a *timed multitask model*, which is built with the assumption of a synchronous data flow model, and restricts sending the output only at the end of each invocation cycle. Despite the power of *Ptolemy* in control design modeling and simulation, the model does not contain sufficient information for realistic performance analysis of ECSW. The *timed multitasking model* can only be implemented on a specially-designed operating system or a time-triggered environment. Ignoring the platform model and its resource capacities also limits the *Ptolemy* verification to only such features as control stability, under the assumption that all activities can be completed within the designed constraints, which may not be the case after the components are implemented and allocated on a target.

Timewaver is another modeling environment that defines the modeling framework for both function and performance modeling [18]. The modeling framework consists of port-based software components and couplers. The couplers are used to connect software components. The framework allows performance to be annotated to the components and couplers. However, *Timewaver* is only a modeling environment. The analysis is achieved using a different complementary tool, *TimeWiz* [85], which is based on a different modeling framework.

Gerber, Hong, and Saksena developed a design verification process based on an asynchronous task graph [26]. The modeling framework contains tasks, asynchronous buffered channels for the communications. Each task has a set of performance parameters associated with it, including a period, a deadline less than its period, an execution time, and a release offset. The modeling framework also defines a set of system-level performance constraints, including data freshness, correlation, and separation. The framework assumes either a homogeneous platform or that the tasks and communications have been allocated to the target platform. The work is later extended to include the task chain and the platform in the model, delay constraints in the system-level parameters, and probabilistic resource demands [40]. The framework has also been integrated with the Real-Time Object-Oriented Modeling (ROOM) framework [74] for modeling and analysis system performance [70].

A large number of performance analysis techniques have been developed based on the runtime model framework which consists of tasks. Traditionally, the system is modeled in a *simple task model* containing a set of independent tasks characterized by their execution times, periods, and deadline [47, 6]. Harbour, Klein, and Lehoczky built a *complex task model* by allowing tasks to contain subtasks with different priorities, deadlines, and release offsets [29]. The platform models for both simple and complex task models are limited to uniprocessor. Both models also assume independent tasks. Such models are not applicable to current ECSW modeling as the system typically involves multiple dependent tasks running on a multiprocessor environment. To address this issue, modeling frameworks that include modeling elements for both task dependencies and multiprocessor platforms are required. Examples of such frameworks include the distributed task graph [81], the task-module interaction graph [64], and the directed task graph [51].

Other modeling frameworks have also been proposed and evaluated for embedded and real-time system design. These include many UML-based performance modeling frameworks [4, 92, 3, 33]. Such a framework usually extends the existing UML modeling language with performance and platform modeling components. Although these frameworks have the advantage of compatibility with existing models, as UML is widely accepted, the methods for accurate analysis are difficult to develop due to the complicated interactions between UML components and their invocation mechanisms. The frameworks based on queuing networks have also been developed and evaluated [28, 95, 55]. The model constructed with these frameworks can be mathematically analyzed using mature queuing theories. However, the results generated are less useful and accurate due to the nature of the probabilistic model.

For platform modeling, Mok and Feng proposed a method of using a *virtual resource* [60]. The virtual resource concept allows one to design and analyze the system without knowing the target platform. However, their work is limited to a single processor. Kettler, Katcher, and Strosnider modeled the operating system according to the operating system's structure and measured data [43]. Brown and Seltzer developed a micro-benchmark, *hbench:OS*, to measure the low-level system calls, and built the performance model of the operating system hierarchically from bottom up. Methods for modeling middleware systems have also been developed [39, 72, 65]. Although the models constructed using these methods and frameworks reflect the platform performance at certain levels, they are not adequate for use in application level performance model and derivation since the modeling of this type of system software does not include the workload consideration of the system.

Our objective in this work is to construct the modeling framework that integrates all aspects of the ECSW development cycle, while still maintaining model independence so the designer can focus only on the information in which he or she is interested. To this end, we have constructed our modeling framework to contain a correlated system aspect for component, structure, runtime, and platform with performance information annotated to modeling components.

2.7 Summary

ECSW performance analysis requires modeling the performance of individual components and performance constraints at each design phase. In this chapter, we have presented a framework for performance modeling and analysis at all different design phases, including software architecture development and evaluation, platform construction, and deployment design. The models constructed at these design phases are the structural model, platform model, and runtime model. The performance is annotated to these models as a set of design-phase-dependent performance parameters. The performance parameters are further classified into constraints and characteristics to improve reusability of their values. With this framework, the ECSW is first modeled in a structural model, consisting of a set of transactions. Each transaction is modeled as a component graph, which is a weighted directed graph. The characteristics are annotated to nodes and links in the graph as weights, while the constraints are annotated to transactions. The components in the structural model are port-based objects with the ports being used for synchronous communications between components. Since the component's performance characteristics depend on its execution environments, a virtual resource concept is introduced to model the component's resource demands in a platform-independent form.

The platform model specifies the ECSW execution environment and consists of hardware and system software. It is modeled as a weighted undirected graph. The nodes in the model represent computation devices. These contain both hardware components and system software essential for the application executions. The weight functions of each node model the resource capacities of the device and the overheads of the system software services. The links represent the physical connections between the computation devices and the protocol software managing the link. Similarly, the weight function of a link defines its resource capacity and the overhead of the protocol.

The runtime model is a weighted directed acyclic graph, representing the same system as the structural model. The nodes in a runtime model are tasks — the basic schedulable units on a computation device. A task is composed of a sequence of components. The annotated weight functions specify the resource demands of the tasks. The links model task dependencies, and the annotated weight functions specify the communication resource demands between tasks. The tasks with their inter-dependencies form a task chain, which models the same behavior of a transaction in the structural model. As the tasks in the runtime model have been allocated to computation devices on the platform, the resource demands are represented in a real wall-clock. A set of constraints, including invocation periods, deadlines, and release offsets, are annotated to the task chains as well as individual tasks.

Our performance modeling framework integrates the functional and performance modeling, while allowing them to be analyzed separately. We consider the ECSW development process as an iterative model transformation process that generates a new model by refining the model at a previous phase. The rest of the dissertation will detail the model transformation and the performance analysis methods based on the modeling framework in this chapter.

CHAPTER 3

Estimation of Structural Model Performance

A structural model is the first software model constructed after the control discretization. The performance of the structural model can then be estimated using the annotated performance of the constituent software components and their interactions. Since the concrete performance has to wait until the platform is decided and the software model is transformed to a runtime model, the performance of the ECSW can only be estimated using the structural model. The performance estimate can help evaluate the software architecture modeled in the structural model and assist the platform design to provide sufficient resources.

The estimated performance of a structural model is in the form of virtual resource demands. In this work, we use *bound estimations* to evaluate the performance of a structural model. The estimations include both end-to-end resource demands of each transaction and total system resource demands of all transactions. The end-to-end demand estimation requires identifying the longest execution path and beneficial parallelism. The total system workload estimations require considering the concurrency of the transactions. This chapter presents the techniques that address these analysis issues.

3.1 Metrics of Performance Estimation

The performance metrics we will derive for the structural model includes the end-to-end response delay of the transactions and the total resource demands of each type of resource required by the structural model. The end-to-end response delay of a transaction is in virtual time, derived from the resource demands of the components involved in the end-to-end execution. This end-to-end response delay can be used for the comparison of different software architectures. It can also be used to derive the minimum service rate (capacity) of the platform with which the end-to-end timing constraints can be met. The total resource demands of each type of resource are useful for the platform design and configuration with consideration of all required resources. Since the resource demands of the software in the real-world time can only be determined after the platform design, we use *bound estimations* to derive the values of these performance metrics of the structural model.

The *bound estimations* quantify the values of the performance metrics as a range from the best-case estimate (lower bound) to the worst-case estimate (upper bound). To define the best- and worst-case estimates, we need to first define the *structural configuration* because the values of estimates vary along the configurations.

Definition 3.9. *Given a structural model M, a* structural configuration *of M partitions the components in M into groups with*

- *each group executing a dedicated identical virtual resource;*
- *different virtual resources for the links between groups and the links within a group; and*
- *the components in the same group executed sequentially.*

The structural configuration defines how the components and links in the structural model are organized. Different group formation results in different parallel and sequential executions of components and groups. Such parallel/sequential executions result in different end-to-end response delays. With the structural configuration, we can now define the *best-case estimate* and *worst-case estimate*.

Definition 3.10. *Given a structural model M and a performance metrics of interest η, the* best-case estimate, *$\varphi_b(M)$, is defined as*

$$\varphi_b(M) = min\{\varphi : \eta \leftarrow \Omega(M)\}$$

where φ is the value of the metrics η resulting from a structural configuration of M, $\Omega(M)$.

This definition indicates that the best-case estimate $\varphi_b(M)$ is the minimum value among the performance of all configurations $\Omega(M)$. We call the corresponding configuration the *best-case configuration*, $\Omega_b(M)$. Similarly, the *worst-case estimate* can be defined as follows.

Definition 3.11. *Given a structural model M and a performance metrics of interest η, the* worst-case estimate *of $\varphi_w(M)$ is defined as*

$$\varphi_w(M) = max\{\varphi : \eta \leftarrow \Omega(M)\}$$

where $\varphi_b(M)$ is the value of the metrics η resulting from a structural configuration $\Omega(M)$.

According to the definition, the software performance depends on the configuration of the system. As the design advances to the later phase with more implementation details filled in, the real software configuration, called the *system configuration*, will be determined according to the platform configuration and runtime management requirements. Given any system configuration $\Omega(M)$, the performance of $\Omega(M)$ of an interested metric η must be bounded by the base-case and worst-case estimates, i.e.,

$$\varphi_b(M) \leq \varphi(M) \leq \varphi_w(M)$$

Bound estimations are suitable for early design phase performance analysis since the results are independent of the software execution environments and are robust under high model uncertainty. This uncertainty of the structural model is caused mainly by the communications among the components. In a given structural model, the resource demands of the components are fixed after the selection of components.[1] Later, the design concentrates on determining where and how each component should be executed. The execution locations of the components determine the communication resource demands. The way a component is executed may introduce blocking and preemption times to the component as mutually exclusive access resources may exist. These properties can only be determined when a runtime model is constructed.

In this analysis, we estimate the bounds of the performance metrics with no assumption—such as type and number of processors, support system software, and network protocols—on the platform configuration. The performance bounds derived in this way are only sensitive to the software architecture design, and therefore, can be used both for architecture comparison and for platform design assistance.

3.2 Transformation of Transactions

In a structural model, the control behaviors of the ECSW are modeled as multiple concurrent transactions under each system mode. From the performance analysis perspective, we are interested in the end-to-end delays of these transactions and the total resource demands (workloads) of the ECSW. A simple and straightforward approach to obtaining the values of these metrics is to traverse the transaction model, find the best-case and worst-case configurations, and compute the metrics using the resource demands of the components and connections. This requires the tech-

[1]The method for selecting the components that both meet functional requirements and minimize the overall computation/storage resource consumptions requires determination of proper granularity of the system, formal model of components, and consideration of system behaviors (e.g., mutual exclusive access of a component), and is therefore beyond the scope of this work.

niques of graph search. Some effective graph search algorithms [15] are not directly applicable to our structural model due to (1) the existence of cycles in our transaction models, and (2) the existence of multiple input/output components of a transaction. We have to convert our transaction models to directed acyclic graphs (DAG) to apply the existing techniques. In this section, we present the methods for eliminating the cycles and multiple inputs/outputs of a transaction.

3.2.1 Cycle elimination

Our approach to eliminating cycle in a transaction model is to separate the cycle as another transaction and replace it with an aggregated node in the original model. The cycles in our structural model represent the multi-rate and/or closed loop feedback control that commonly exists in control systems. A cycle is modeled as an inner loop in a transaction in the structural model, which is running at a higher frequency than the invocation rate of the transaction. The transaction's rate and its contained cycle's rate must be harmonic with the cycle's rate as an integral multiple of the transaction's rate. The components in the cycle must be invoked at least once in each execution of the chain.

Due to different invocation rates of the cycle and the transaction, the executions of the cycle must be triggered separately at runtime by a signal, for example, hardware or OS timer, or a control event and signal, different from that of the transaction. Higher invocation frequency of the cycle implies the component(s) in the transaction beyond the cycle cannot supply or consume every new version of data required or produced during the execution of the cycle. It also implies that the feedback data generated by a component in the cycle can only be used in the computation in the consecutive invocation of the cycle. Based on these properties, we can then equivalently separate the cycle from the transaction as follows.

1. Eliminate the feedback link;

2. Create a new transaction for the component subgraph of the cycle with the feedback link connecting to an added dummy component;

3. Assign the invocation rate of the new transaction as the rate differences between the original cycle and the transaction.

In the newly-created transaction, the components consuming data from the original transaction are input components, and those producing data for the original transaction are output components. To preserve the resource demands w_f of the feedback connection f, we add a dummy node at the end of the new transaction with the computation resource demand $w_i = 0$. The resource demand of the link between the output component and the dummy node is assigned to be w_f. The newly-created transaction is running at its own rate, which is the difference between original cycle rate and the transaction rate. Figure 3.1 shows the elimination of cycles in a transaction.

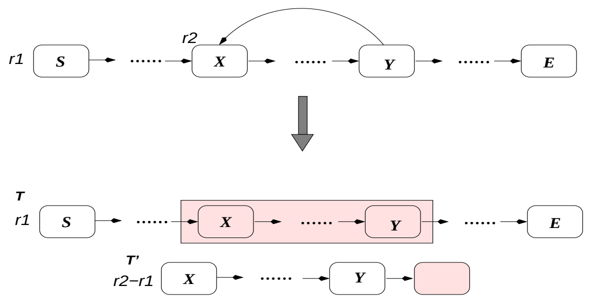


Figure 3.1: Elimination of cycles in a transaction model.

Although the two transactions are modeled as independent ones after the cycle elimination, their invocations are mutually exclusive. In other words, the sequence $X \ldots Y$ in T and the sequence of T' in Figure 3.1 execute during different invocation periods. If $X \ldots Y$ in T executes at a period P_i ($P_i = 1/r_2$), T' does not execute in P_i. This ensures the executions of $X \ldots Y$ in T and T' do not interfere each other, while maintaining the total system workloads and causality of components' executions.

It is also possible that the cycles are nested in a model. In such a case, the transformation should be performed iteratively, starting from the innermost cycle until all cycles are eliminated in a transaction model.

3.2.2 Multiple inputs/outputs elimination

After eliminating the cycles, the transaction models in a structural model become directed acyclic graphs (DAG). The techniques for DAG manipulation can be applied to find the configuration by exploring the parallelism in the model between one pair of input and output. We adopt a single-source single-destination graph traversal algorithm, such as Dijkstra's shortest path algorithm, to do this. However, with the existence of multiple input components and multiple output components in a transaction, it may be necessary to run the algorithm multiple times to find the best/worst-case configuration. In the worst case, given a transaction model with m input components and n output components, finding the configuration of interest requires executing the algorithm $m \times n$ times. To resolve this issue, we insert a dummy node as a start/end of a transaction with multiple input/output components. For multiple input components, a dummy node *start* is inserted before all input components, with a link from *start* to each input component. For multiple output components, a dummy node *end* is inserted after all output components, with a link from each output component to *end*. The resource demands for these dummy nodes and their links to input/output components are assigned with $w_{start} = w_{end} = w_l = 0$.[2] In this way, a transaction with multiple input and output components can be converted to one with a single input and output. Figure 3.2 is an example of a transaction graph after making such a transformation.

After the transaction models are converted to single-input single-output DAGs, the configuration can be identified, and consequently the performance of the transactions and systems can be deter-

[2]Some constraints such as the correlations between inputs and outputs [26] can be built with this transformation by assigning some connections with non-zero resource demands.

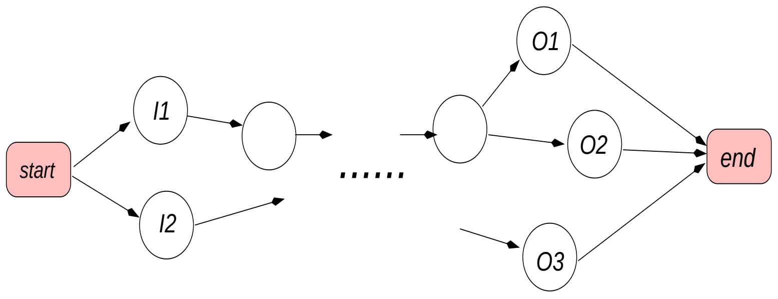


Figure 3.2: Elimination of multiple inputs and outputs.

mined by running the estimation algorithm, which is developed based on a revision of techniques for weighted graph manipulation. The algorithm is discussed in detail in the next section.

3.3 Performance Estimation of a Structural Model

Our bound estimations of structural model performance is based on the construction of the structural configuration. The constructed configuration can be viewed as running the ECSW on a virtual platform. This platform contains any required number of virtual devices with the resources resulting in the best or the worst performance. All these devices are assumed to have the identical service rate.

With the obtained configuration, the performance of the structural model can be computed using the virtual resource demands of the constituent components and the connections. Depending on the components sharing the same virtual device, the end-to-end response can be different for a component sequence. Similarly, the end-to-end response also varies according to the virtual channel used for the connections (intra-device or inter-device). Although the total computation resource demands of the system are constant for a structural model, the computation workload at some peak time and the communication resource demands typically vary with the configurations. In this section, we will discuss how to determine these performance metrics by finding a corresponding configuration.

3.3.1 End-to-end response delay

The end-to-end response delay of a transaction is the time duration between the readiness of the transaction's input and the delivery of the transaction's outputs. By the end of the end-to-end response, the executions of all components in the transaction are completed. The end-to-end response delay measures how quickly a transaction can react to the system. It usually affects the control quality of the ECSW. A better design should minimize the end-to-end response delay of every transaction in the presence of the interferences of other concurrent transactions and system activities (such as OS and middleware operations).

To obtain the end-to-end response delay bound of each transaction, it is essential to know the best-case and worst-case configurations. To ascertain this, we must explore the parallelism of the concurrent transactions as well as the concurrent components within a transaction. As the concurrent transactions are modeled as isolated graphs, our study focuses on the components' parallelism within a transaction. Such parallelism of the concurrent components is modeled as concurrent paths in a transaction model.

Definition 3.12. *Given a transaction model in graph G containing multiple paths between nodes (i, j), the paths are called* concurrent paths *if no common node besides i and j is shared between any two paths.*

An example of a transaction with concurrent paths is shown in Figure 3.3.

This example contains the following concurrent paths:

- Two concurrent paths between (A, C): $A \rightarrow B \rightarrow C$, and $A \rightarrow C$;
- Two concurrent paths between (B, F): $B \rightarrow E \rightarrow F$, and $B \rightarrow C \rightarrow F$;
- Three paths between (A, F): $A \rightarrow B \rightarrow E \rightarrow F$, $A \rightarrow C \rightarrow F$, and $A \rightarrow D \rightarrow F$.

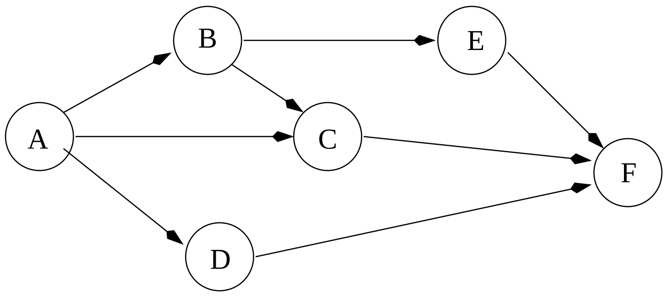


Figure 3.3: An example of transaction with concurrent paths.

The total number of paths between (A, F) is four.

In a transaction with concurrent paths, its end-to-end response delay depends on the time taken to complete the components on the longest execution path if all paths are executed in parallel.

Theorem 3.1. *Given multiple concurrent paths between nodes (i, j) in a transaction model, denoted as $P^1(i, j), P^2(i, j), ..., P^k(i, j)(k \geq 2)$, which are executed in parallel, the end-to-end response delay $\varphi(P(i, j))$ between (i, j) is determined by the path that takes the longest time to complete:*

$$\varphi(P(i,j)) = max_{1 \leq x \leq k}(\varphi(P^x(i,j)))$$

Proof. The proof is straightforward. Since no node is shared by any two paths, the end-to-end response delay of a path $P^x(i, j)$ can be computed as

$$\varphi(P^x(i,j)) = \sum_{c_i \in P^x(i,j)} w_c(c_i) + \sum_{l_j \in P^x(i,j)} w_l(l_j)$$

Supposing the path $P^x(i, j)$ takes the longest time $\varphi(P^x(i, j))$ to complete, we have

$$\forall y, 1 \leq y \leq k, y \neq x, \varphi(P^y(i,j)) \leq \varphi(P^x(i,j))$$

Since there is no interference between the executions of different paths, by the time $\varphi(P^x(i, j)) - e_j$ occurs, at which $P^x(i, j)$ execution reaches j, the executions of all other paths have also reached

j. This means that all components and communications on all paths before j have been completed. Therefore, the end-to-end delay $\varphi(P(i,j)) = \varphi(P^x(i,j))$, which is the longest one. □

According to Theorem 3.1, the end-to-end response delay of a transaction depends on the the completion time of the longest path among all parallel paths, which consequently depends on the structural configuration of the ECSW. In a structural configuration, we can consider each group as a component in Theorem 3.1, and the links between groups as the component links. The worst-case delay of each group is computed as the sum of the resource demands of the components and communications in the group. Therefore, to estimate the end-to-end response delay, we need to transform a transaction model to the structural configuration containing multiple concurrent paths, and assume no path shares the same virtual resource. The rest of this section discusses the methods used to construct the structural configurations for the bound estimations.

Best-case configuration

To determine the best-case configuration, we need to fully explore the parallelism of the graph with the assumption of an ideal execution environment. We therefore assume an execution environment consisting of an unlimited number of virtual devices and links as a virtual platform. On such a virtual platform, no resource contention is introduced to the components' executions except those caused by shared software (such as shared data or codes). All component groups and links in the later-generated structural configuration will run on dedicated devices and channels with sufficient resources. Only dependency constraints affect the invocation of the components, except for the start component, which is triggered by an external signal like data arrival or timer firing. There is also no interference between transactions as different transactions should run on different sets of dedicated virtual devices, thus eliminating the physical resource sharing. The virtual devices can be mapped to some physical devices later during platform design/evaluation and component allocation.

Given each transaction running on a set of dedicated virtual devices, the best-case response delay of a transaction can be computed by finding an allocation with a component sequence that fully explores the beneficial parallel executions of the components in the transaction. A parallel execution is said to be *beneficial* if the end-to-end response delay of the concurrent components is shorter when they are executed on different virtual devices than when sharing the same virtual device with consideration of both computation and communication resource demands. The problem of finding the best-case end-to-end response delay (BERD) is formally defined below.

> Given a transaction tr modeled as a component graph $M_x(tr)$, a model M_p of virtual platform with virtual device set P ($|P| \leq |C|$) and virtual links L, find a structural configuration of C on P, $\{C(P_i)\}$ with the execution sequence of the components in each $C(P_i)$ such that the end-to-end response delay is the minimum, i.e., $\varphi_b(tr) = min_{alloc\&sched}\{\varphi(tr)\}$.

The BERD problem is NP-complete, as proved in Theorem 3.2.

Theorem 3.2. *The BERD problem is NP-complete for any platform model with $|P| \geq 2$.*

Proof. The proof is similar to the one in [71], by reducing the 3-PARTITION problem to BERD.

We first show the problem is in $BERD \in NP$. Given a structural configuration with sequenced components of a transaction tr, we can verify its end-to-end response delay to be less than a constant T by finding the longest execution path between the start node s and the end node e. At each step, the verification algorithm computes the completion times of all immediate successors of the current components. Since the graph is a directed acyclic graph, the algorithm takes a polynomial-time to generate the completion time for every component. We can then compare the completion time of the e node with T.

We then construct a polynomial algorithm to transform an instance of the 3-PARTITION problem to an instance of BERD problem. Given an instance of the 3-PARTITION problem with $A = (a_1, ..., a_{3m})$, $s(a_i)$ and a bound mB, we construct a graph $G =< C, L, F, s, e >$ of BERD as follows:

- The components are constructed as $C = A \cup X \cup Y \cup s, e$,

  where $A = \{a_1, ..., a_{3m}\}$ is the set of 3-PARTITION, $X = \{x_1, ..., x_m\}$ is a set of m elements, and $Y = \{s, y_1, ..., y_{m-1}, e\}$ is a set of $m + 1$ elements.

- The links among components are constructed as

$$\begin{aligned} L = \; & \{(s, a_i)\} \cup \{(s, x_1)\} \cup \{(s, y_1)\} \cup \{(x_i, y_i)\} \cup \{(y_i, y_{i+1})\} \cup \{(y_i, x_{i+1})\} \cup \\ & \{(a_i, e)\} \cup \{(x_m, e)\} \cup \{(y_m, e)\} \end{aligned}$$

  for $1 \leq i \leq m - 1$.

- The resource demands of the components are assigned as

$$F_c(c) = \begin{cases} s(a_i) & \text{if } c = a_i \\ B & \text{if } c = x_i \\ 0 & \text{if } c = y_i \text{ or } s \text{ or } e \end{cases}$$

  The resource demands of the links are assigned as

$$F_l(l) = \begin{cases} 0 & \text{if } l = (s, x_1) \text{ or } (x_i, y_i) \text{ or } (x_m, e) \\ mB & \text{otherwise} \end{cases}$$

The completion time of the constructed BERD is $T = mB$. The construction of BERD is a polynomial time process of mapping a_i to BERD graph with addition $2m$ components, and creating total $9m$ links ($6m$ for (s, a_i) and (a_i, e), $3m$ for (x_i, y_i), (y_i, x_{i+1}) and (y_i, y_{i+1})), and assign the F values for each component and link.

We now prove that a configuration is a solution to the 3-PARTITION problem if and only if its corresponding configuration of G is a solution to the BERD problem. We first show the 3-

PARTITION solution is a BERD solution. Given a 3-PARTITION solution, we have $\sum_{a_i \in A} s(a_i) = B$. We can then assign X to one device P_1 and assign $A \cup Y$ to another device P_2. The response time on P_1 is

$$\varphi_{P_1} = \varphi(X) = \sum_{x \in X} F_c(x) = mB$$

Since the link resource demands between components on the same processor is 0, the end-to-end response delay on P_2 is

$$\begin{aligned}\varphi_{P_2} = \varphi(A \cup Y) &= \sum_{c \in A \cup Y} F_c(c) \\ &= \sum_{a \in A} s(a) + 0 \\ &= mB\end{aligned}$$

The communication delay over the link between P_1 and P_2 is 0. This is because the $F_l(x_i, y_i) = F_l(y_i, x_{i+1}) = 0$. Therefore, the overall end-to-end response delay of the BERD graph is

$$\varphi = max(\varphi_{P_1}, \varphi_{P_2}) = mB$$

Then, we show that a BERD solution is a 3-PARTITION solution. To meet $\varphi \leq mB$, A and Y must be running on the same device to eliminate all communication delays. Assigning any two components $c_i, c_j \in A \cup Y$ to different devices will result in partition G in two disjoint segments, which consequently results in $\varphi \geq 2F_l = 2mB$ due to $F_l = mB$ for any link. Further, no component $x \in X$ can be assigned P_2 if $A \cup Y$ is assigned to P_2, because assigning x_i to P_2 will result in the response delay on P_2 greater than mB. This is shown as follows. Suppose x_i is allocated to P_2 with $A \cup Y$. If x_i is executed before some $a_i \in A$, then the computation time of path A will be $\varphi(A) = mB + B > mB$. The $\varphi = max(\varphi(A \cup Y \cup x_i), \varphi(X)) > mB$. If x_i is executed after all $a_m \in A$, it must complete before e, which is the last component of Y. Then $\varphi(Y) = mB + B > mB$. Since $\varphi(A) = mB$ in this case, and x_i starts after A, $\varphi(X) =$

$mB+(m-i)B > mB$ must be true. Again, this results in $\varphi = max(\varphi(A \cup Y \cup x_i), \varphi(x)) > mB$. Therefore, no x_i can be assigned to the same device of $A \cup Y$.

Given X and $A \cup Y$ assigned to different devices, P_1 and P_2 respectively, we sequence component $a_i \in A$ executed between (y_i, y_{i+1}) for $1 \leq i \leq m-1$. According to the execution dependencies between x_i and y_i, the start time of y_i, $S(y_i)$, can be computed as follows:

$$\begin{aligned} S(y_i) &= S(y_{i-1}) + max(\sum F_c(a_i), F_c(x_i)) \\ &= S(y_{i-1}) + max(\sum s(a_i), B) \\ &\geq S(y_{i-1}) + B \end{aligned} \tag{3.1}$$

Given the execution starts at time 0, $S(s) = 0$, we can derive the following equation according to Eq. (3.1).

$$\begin{aligned} S(y_i) &\geq S(y_{i-1}) + B \\ &\geq S(y_{i-2}) + 2B \\ &\ldots\ldots \\ &\geq S(s) + iB \\ &\geq iB \end{aligned} \tag{3.2}$$

To meet $\varphi \leq mB$, we must have

$$S(y_m) \leq mB$$

$$S(y_{m-1}) \leq mB - B = (m-1)B$$

......

$$S(y_i) \leq iB$$

......

According to Eq. (3.2) and (3.3), we have

$$S(y_i) = iB \tag{3.3}$$

Therefore,

$$max(\sum s(a_i), B) = S(y_i) - S(y_{i-1}) = B$$

$$\sum s(a_i) \leq B$$

This indicates that it is a solution of 3-PARTITION.

□

Since finding the best-case configuration with the minimum end-to-end response delay is NP-complete, we develop a polynomial-time greedy algorithm to find a suboptimal solution for the problem. The algorithm initially configures the system with each component assigned to a different identical virtual device. All concurrent paths are executed in true parallel in this configuration. It then tries to generate the configuration by merging the components to the same device. If the new configuration results in a shorter end-to-end response delay of the longest execution path, the algorithm continues using the new configuration until no new configuration can shorten the end-to-end response delay of the longest execution path. The algorithm is shown in Algorithm 3.1.

The input of the algorithm is a directed acyclic graph with single input s and single output e, a set of components in C, inter-component links in L, and resource demand functions F. The algorithm finds a possible BERD through incrementally reducing the communication delay on a critical path (Step 5 — 18). According to Theorem 3.1, φ depends only on the longest execution path of a transaction. Reducing the completion time of the longest path will shorten φ of the system, which leads to φ_b. So, the algorithm tries to reduce the completion time of the longest path by moving the components with the largest communication cost at each step, after finding the longest path resulting in current φ_b in the current configuration, in the hope of reducing the overall end-to-end

Algorithm 3.1 *Finding BERD.*

```
Input: a transaction graph as $T =< C, L, F, s, e >$,
    $s$: single input node,
    $e$: the single output node.
Output: best-case response time $\varphi_b$ between $(s, e)$,
    configuration $\Omega_b$ containing partitioned $T$ resulting in $\varphi_b$
Begin
  /* each component is on a dedicated virtual device initially */
1      foreach $c \in C$ do   $\Omega(c) \leftarrow \{c\}$;
2      $CP \leftarrow$ find_critical_path($T$);
3      $\varphi_b \leftarrow LCT(e)$;
4      assign unchecked link set $EL \leftarrow L$;
5      while $EL \neq \emptyset$ do
6        find link $l_m(x, y) \in CP \cap EL$ with maximum $F_l$; /* $x, y$ are the components at the ends of $l_m$ */
7        merge $\Omega(x) \cup \Omega(y) \rightarrow \Omega(xy)$;
8        sequence $c \in \Omega_{xy}$ according to $EST(c)$;
9        adjust link cost $F_l(l_m)$;
10         $CP \leftarrow$ find_critical_path($T$);
11         $\varphi_b(CP) \leftarrow LCT(e)$;
12         if $\varphi_b > \varphi_b(CP)$ then
13           $\Omega \leftarrow \Omega \cup \Omega(xy) - \Omega(x) - \Omega(y)$;
14           $\varphi_b \leftarrow \varphi_b(CP)$;
15         else restore $F_l(l_m)$;
16         end-if-else
17         $EL \leftarrow EL - \{l_m\}$;
18       end-while
19       return $\Omega$, $\varphi_b$;
End.
```

response delay. After the merge, the components in the same partition are sequenced according to their earliest start times (EST). Note that such sequencing has a negative effect on shortening the end-to-end response delay because sequencing components' execution in the merged partition, which have been executed in true parallel on different devices, may postpone the start of some component, and consequently introduces a longer delay. To check whether the new configuration yields a better φ, the algorithm finds the new longest path of the updated graph with the updated link cost and sequenced components. If the new longest path has a shorter response delay, it keeps the new configuration with the adjusted communication link cost and the merged partition. Otherwise, the link with the next highest cost on the critical path is eliminated. At each round, the checked link is removed from the list no matter whether or not it results in a shorter response time. The process is repeated until all links are examined.

Algorithm 3.1 calls a function $find\_critical\_path()$ to determine the critical execution path and its response time. The implementation of the $find\_critical\_path()$ function is similar to the Dijkstra's shortest path algorithm [15] but chooses the longest path instead, as shown in Algorithm 3.2.

Algorithm 3.2 *find_critical_path().*

```
Input: a transaction graph as $T =< C, L, F, s, e >$,
    $s$: single input node,
    $e$: the single output node.
Output: the longest execution path $CP$,
    the $EST(c)$ and $LCT(c)$ for each component $c$.
Begin
1   foreach $c \in C$ do $prec(c) \leftarrow \emptyset$;
2   $EST(s) \leftarrow 0$;
3   $LCT(s) \leftarrow EST(s) + F_c(s)$;
4   $path(s) \leftarrow s$;
5   $EC \leftarrow C$;
6   while $EC \neq \emptyset$ do
7     remove component $c \in EC$ with all inputs visited;
8     foreach $u \in succ(c)$ do
9       if $EST(c) + F_l(l_{cu}) + F_c(c) > EST(u)$ then
10        $prec(u) \leftarrow c$;
11        $EST(u) \leftarrow EST(c) + F_l(l_{cu}) + F_c(c)$;
12        $LCT(u) \leftarrow EST(u) + F_c(u)$;
13      end-if
14    end-foreach
15  end-while
16  $c \leftarrow e$;
17  while $c \neq s$ do
18    $CP \leftarrow c$;
19    $c \leftarrow prec(c)$;
20  end-while
21  return $CP, EST, LCT$;
End.
```

In Algorithm 3.2, each component records its immediate predecessor, in a variable $prec$, from which it is reached. It also records the corresponding earliest start time EST and latest completion time LCT. Each component also maintains a list of its successors in $succ(c)$. The $succ(c)$ contains both the component(s) immediately depending on c and the concurrent component(s) scheduled immediately after c. The algorithm starts from the s node, finding a component whose predecessors have all been visited, and checking whether the component is on the longest execution path of its successor. If so, its successor's predecessor is recorded with EST and LCT updated in Step 9-13.

Otherwise, the successor keeps its current predecessor, EST and LCT. The process continues until all components are processed. The longest execution path is then obtained by tracing back from e to s following $prec$ of the visited components.

We now verify that Algorithm 3.1 has a polynomial-time computation complexity. Suppose the transaction graph contains $N = |C|$ components and $E = |L|$ links. The initialization at Step 1 of Algorithm 3.1 takes N. In each main loop (Step 6-15), the algorithm finds a link with the maximum F_l cost, which takes E for the first round, $E - 1$ for the second round, and so on. Scheduling the components after merging two partitions at Step 8 is a process of ordering components according to EST, which takes no more than N. The $find\_critical\_path()$ function is invoked in every loop. In $find\_critical\_path()$ function, it takes at most N time to find a component with maximum EST at Step 7. Since the graph is a directed acyclic graph, the maximum number of successors for each component are $N - 1$, $N - 2$,...,0. The **foreach** loop (Step 8-14) therefore takes at most $N - 1$,...,0 time for each component. Since the **while** loop visits every component exactly once, the total time for the **while** loop is $N^2 + \sum_{i=1}^{N}(N - i)$. So the total time for $find\_critical\_path()$ is $O(N^2)$. As the Algorithm 3.1 visits every link exactly once, the total time for the algorithm is $O(E \times (N + N^2 + N^2) + N^2)$. Therefore, the computation complexity of Algorithm 3.1 is $O(N^2 \times E)$.

The worst-case end-to-end response delay of a transaction can be estimated in a similar way to Algorithm 3.1 and with consideration of other transactions' interferences. Instead of adjusting the configuration to reduce the delay of the longest path at each step, the algorithm for φ_w should adjust the configuration to extend the delay of the longest path. As the worst-case interferences from other transactions depend on both platform and scheduling policies, such a worst case is usually difficult to obtain with only the structural model. On the other hand, knowing the worst case φ_w does not have any valuable impact on the design, since any arbitrary design will yield a performance no worse than φ_w. To this end, we can model the estimated performance bounds of the end-to-end response delay of a transaction as an open range with $[\varphi_b, \infty)$, and ignore the concrete value of φ_w. In contrast, knowing the best-case performance is critical as it can be used for many design aspects such as verifying the satisfiability of the performance specifications, comparing the design alternatives, directing platform selections, and guiding the design model construction at the later phase. The design objective should then focus on how to achieve the best-case response time.

3.3.2 Total resource demands

The total resource demands of designed ECSW consist of the resource demands of all concurrently active transactions, including both computation resources, communication resources, and storage resources. The system resource demands capture the workloads during the system execution. Different from the end-to-end response delay estimations, which focus on resource demands of individual transactions, the total resource demands characterize the ECSW as a whole to the supporting system, such as the platform. The system-level resource demand estimations also consider the interferences among the transactions due to resource sharing and resource contentions. These are useful for system capacity analysis and platform design, which determine the resources provided for the ECSW execution. The derived system resource demands are in virtual resources at this phase, and can be mapped to one or more physical devices in the platform, depending on the resource demands of components and resource capacities of a device. Thus, knowing the total resource demands of the system is essential for constructing a platform with sufficient resources.

We distinguish the system resource demands of computation from the resource demands of communications because these workloads require different types of devices. As the functional components must be executed whenever there is a transaction containing them, the computation resource demands vary only with the system modes, which determine the active transaction set (assume the resource demands for the components are constants). For a system mode $m_i \in m$ defined in a structural model M, the computation resource demands can be computed as:

$$\varpi(M_{comp}, m_i) = \sum_{T_i \in \Phi(m_i)} r(T_i) \cdot (\sum_{x \in T_i} F_c(x))$$

where $\varpi(M_{comp}, m_i)$ is the computation resource demand under mode m_i, $r(T_i)$ is the invocation rate of transaction T_i, and $F_c(x)$ is the performance characteristics of the computation resource of component x.

The upper bound of the computation resource demand can be computed as the maximum demand among all system modes:

$$\varpi_w(M_{comp}) = max_{m_i \in m}\{\varpi(M_{comp}, m_i)\}$$

If the ECSW design defines a degraded mode under which the system can function with less resource, the lower bound of the computation resource demand is then the resource demand at the degraded system mode.

On the other hand, the communication resource demands depend on the configuration of the software, called the *deployment model*. Such a configuration is also a design decision, and depends on the resource availability of the physical devices in the platform. In general, the communication resource demands using a channel within a device (intra-device communication) are different from the communication demands using a channel between two devices (inter-device communication) for the same communication. The resource demands of these two types of channels are complementary to each other. For any type of communication channel, the total communication resource demands of all these types of channels can be computed as:

$$\varpi(M_{comm,type}, m_i) = \sum_{T_i \in \Phi(m_i)} r(T_i) \cdot (\sum_{y \in T_i} F_l(y))$$

where $\varpi(M_{comm,type}, m_i)$ is the communication resource demand of the channel type $type$ under mode m_i, y is a link in T_i, and $F_l(y)$ is the performance characteristics of link y.

The bounds of the communication resource demands are determined by the execution locations of the components. For the inter-device communication, the lower bound is zero when all components are allocated on the same device. The upper bound occurs when every component in the model is allocated on a physical device, implying all communications are via inter-device channels. The maximum resource demands can then be computed as:

$$\varpi_w(M_{comm,inter}) = max_{m_i \in m}\{\sum_{T_i \in \Phi(m_i)} r(T_i) \cdot (\sum_{y \in T_i} F_l(y))\}$$

Similarly, the demands for intra-device communications can also be bounded by $[0, \varpi(M_{comm,intra})]$.

In current systems, the inter-device communications are typically much more expensive than the intra-device communication. Therefore, minimizing the inter-device communications is one of the main design objectives. To minimize the total system cost, the designer needs to make trade-offs between the computation resource capacities and communication resource capacities. Given a platform consisting of more powerful computation devices, more components can run on each device, resulting in less communication. However, the cost of a powerful computation device is higher. On the other hand, a platform with less powerful computation devices requires more communication capacity, which may also lead to increasing system costs due to more communication devices needed. Such issues will be discussed later under platform design in this section and under runtime model generation in Chapter 4.

3.4 Applications of Performance Estimation

The goals of the performance estimations of the structural model may include: (1) verifying the satisfiability of the system performance constraints with a given software architecture, and (2) guiding platform design to achieve the required performance. The performance constraint verification is mainly accomplished using end-to-end response delays. Since response delay estimations are represented in virtual resource demands, and cannot be compared directly with the specified end-to-end timing constraints, we use an indirect approach. In this approach, we try to find a better software architecture consisting of a different number of components and/or different component communications which will result in shorter end-to-end response delays of the transactions. Such software architecture will consequently yield shorter end-to-end delays on a real platform and will more likely meet the constraints. On the other hand, the platform design can be guided by the total system resource demands. This section presents methods of applying the performance estimations to the software architecture and platform design.

3.4.1 Software architecture comparison

The purpose of the software architecture comparison is to evaluate different design alternatives. A control design model can be implemented using different software architectures, with the same or different sets of components. As the software architecture is modeled in the structural model, the goal of the software architecture comparison is to compute the best-case response delays of the transactions in the structural model, and find the one with the shortest end-to-end response delay.

Given two structural models of an ECSW,

$$M_1 = (m, T_1, \Phi_1)$$

$$M_2 = (m, T_2, \Phi_2)$$

we need to first determine the relative importance of each transaction. The transactions model the control processes in the control design. There can be multiple concurrent transactions if a system contains multiple control processes for a mode. The concurrent active transaction set for different system modes can be different. A relatively important transaction should be given a higher weight in the comparison because improving the performance of such a transaction will be more beneficial. For each system mode m_i, we define a combined response delay of a model M as

$$\varphi_b(M, m_i) = \sum_{T_i \in \Phi(m_i)} w(T_i, m_i) \cdot \varphi_b(T_i)$$

where $\varphi_b(M, m_i)$ is the combined response delay of M under system mode m_i, T_i is the active transaction under the system mode m_i, $w(T_i, m_i)$ is the weight of the transaction T_i, and $\varphi_b(T_i)$ is the best-case response delay of the transaction T_i.

The comparison is then based on the $\varphi_b(M)$. We say M_1 is better than M_2 under system mode m_i if $\varphi(M_1, m_i) < \varphi(M_2, m_i)$. Otherwise, M_2 is better than M_1 if $\varphi(M_1, m_i) > \varphi(M_2, m_i)$.

To consider the transactions under all system modes, we can use the same approach by constructing a combined delay function with assigned weights to different modes according to the importance of system modes

$$\varphi_b(M) = \sum_{m_i \in m} w(m_i) \cdot \varphi_b(M, m_i)$$

where $\varphi_b(M)$ is the best-case combined delay for the model M, $w(m_i)$ is the weight for mode m_i, and $\varphi_b(M, m_i)$ is the best-case combined delay of mode m_i. The evaluation of the structural model then depends on the of the combined delay of the model $\varphi_b(M)$.

Besides the response performance, the system resource demands of the structural models should also be considered in the comparison. A structural model demanding more resources implies more cost on the platform to maintain the same performance as the one with less resource demand, or worse performance than the one with less resource demand on the same platform. The resource demands of the structural model can also be combined with the response delay performance in the construction of an evaluation function.

3.4.2 Platform design

The platform design includes selecting physical platform components (processor with OS, network with protocol, sensor with device drivers, etc.) and configuring them to form the physical execution environment of the ECSW. It defines the resource availability and runtime resource management mechanisms, thus determining the satisfaction of the ECSW performance constraints at runtime. The platform in an embedded system is usually subject to cost constraints, which limits the physical platform component options. Thus, the goal of a platform design is to construct a platform to meet both the performance and cost constraints. The performance estimations of the structural model derived by using the methods presented in this chapter can help to achieve this design goal.

We guide the platform design by deriving the minimum required resources that meet the performance constraints. The minimum required resource is modeled in the *minimum service rate* of each resource type. Given a structural model M with a set of transactions T, the minimum service rate for each transaction $T_i \in T$ can be computed as

$$\rho_m(T_i) = \frac{\varphi_b(T_i) \cdot \rho(v)}{D(T_i) - o(T_i)}$$

where $\rho_m(T_i)$ is the minimum service rate required to meet T_i's end-to-end timing constraint, $\varphi_b(T_i)$ is the best-case estimated response delay of T_i, $\rho(v)$ is the virtual service rate for derivation of T_i's virtual resource demands, and $D(T_i)$ and $o(T_i)$ are the deadline and release offset of T_i, respectively. As we use the best-case response delays for the transactions in the derivation of $\rho_m(T_i)$, $\rho_m(T_i)$ is the minimum resource requirement that will meet T_i's performance constraints. This implies that any platform with resource availability less than $\rho_m(T_i)$ will result in a violation of T_i's performance constraints. It is transaction-specific, as the timing constraints are specified to individual transactions in M.

As the best-case structural configuration of T_i is known, the minimum service rates for computation and communication can be separated in ρ_m as

$$\rho_m(T_{i,comp}) = \frac{\sum_{x \in \Omega_b(T_i)} F_c(x) \cdot \rho(v_{comp})}{D(T_i) - o(T_i)}$$
$$\rho_m(T_{i,comm}) = \frac{\sum_{y \in \Omega_b(T_i)} F_l(y) \cdot \rho(v_{comp})}{D(T_i) - o(T_i)}$$

Knowing $\rho_m(T_i)$ for all $T_i \in T$, the minimum required resource of the ECSW, in minimum service rate, can be computed as

$$\rho_m(M) = max_{m_i \in m}\{ \sum_{T_i \in \Phi(m_i)} r(T_i) \cdot \rho_m(T_i)\}$$

where $\rho_m(M)$ is the minimum service rate required to meet the system performance timing constraints, $\Phi(m_i)$ is the concurrently active transaction set under system mode m_i, $r(T_i)$ is T_i's invocation rate, and $\rho_m(T_i)$ is the minimum service rate necessary to meet T_i's performance constraints. The max operation is use to choose the peak workloads under all system modes. Similarly, we can derive the minimum system service rates for computation and communication resources as

$$\rho_m(M_comp) = max_{m_i \in m}\{ \sum_{T_i \in \Phi(m_i)} r(T_i) \cdot \rho_m(T_{i,comp})\}$$
$$\rho_m(M_comm) = max_{m_i \in m}\{ \sum_{T_i \in \Phi(m_i)} r(T_i) \cdot \rho_m(T_{i,comm})\}$$

Providing sufficient resources to meet performance constraints under peak workload implies meeting performance constraints at any time.

The platform can then be designed to provide resources more than $\rho_m(M)$. Given a platform consisting of k processing units $P_1, ..., P_k$ and l connection links $L_1, ..., L_l$, the following conditions must be met for the platform to provide sufficient resources to guarantee the ECSW performance constraints are satisfied:

$$\begin{aligned} \textstyle\sum_{i=1}^{k} \rho(P_i) &\geq \rho_m(M_{comp},) \\ \textstyle\sum_{i=1}^{k} \rho(L_i) &\geq \rho_m(M_{comm}) \end{aligned} \tag{3.4}$$

where $\rho(P_i)$ and $\rho(L_i)$ represent the service rates of device P_i and link L_i respectively. With Eq. (3.4), a designer can select the platform configuration that provides minimum resource to meet the ECSW performance constraints.

Besides the platform components' selections and capacity verification, the estimation results can also be used for platform design comparisons. Given multiple platform alternatives that all meet the minimum required resources, other design constraints, such as the system cost, energy consumption, and/or size, can be further considered when finalizing the platform configuration. A designer can also use these results to test some "what-if" scenarios by replacing some components in the platform configuration or reorganizing them to remove the performance bottleneck and investigate the cost-savings.

After the platform design with a platform model is constructed, the structural model can be refined for implementation on the design platform model. The realization of the structural model on a platform is modeled in a *runtime model*, to which many existing real-time analysis techniques can be applied.

3.5 Related Work

In recent years, many approaches have been developed for and applied to the performance modeling and analysis of software systems. Smith and Woodside [78] developed an engineering process to model and analyze the performance of software starting from an early development stage. This process contains a sequence of steps, including performance specification, abstract performance model construction, model parameter determination, comparison of analysis and requirements, and result interpretation. Their modeling method is based on a queuing network model constructed using an execution graph. It is mainly used for soft real-time systems with a client-server architecture. Our work follows a similar engineering process, but uses different modeling methods and assumptions. This is because the client-server architecture is inadequate for ECSW. Other design models used in performance modeling and analysis at an early design phase include UML model [92, 3, 28], software execution model [77], use case maps [96], and Real-Time Object-Oriented Model (ROOM) [95, 70]. These models are powerful for modeling system structure and behaviors. However, most of them are based on a data-centric functional model, and thus, the performance modeling and analysis is complex due to the data contentions caused by the implicitly modeled data sharing.

Another frequently-used method for performance modeling and analysis with abstract design models is queuing network models. Smith [77] used such a model with a system execution model, which contains details of hardware and software design, to evaluate the performance of the designed system. Woodside *et al.* [94] applied a layered queuing network model (LQN) to evaluate the system performance. In their work, the LQN model is constructed by tracing the execution scenarios, and representing both software components and hardware devices as service centers with queues. The method has been evaluated on various applications such as multimedia and telecommunications [76, 95]. Menasce and Gomma [28] simplified the LQN model by reducing it to a two-layer LQN in which software queuing network forms one layer and the hardware queuing network forms another. The model can be used to analyze software contentions in a design. Although the LQN model supports simple and quantitative performance modeling and analysis, it is not suitable for performance analysis of ECSW with hard real-time constraints since the LQN model provides only statistical analysis results such as throughput and average utilization. To derive some of the LQN model parameters, design details including component allocations and system runtime models are required, but these are usually unknown during an early design phase. For the same reason, powerful performance modeling and analysis techniques based on real-time scheduling theories, such as [29, 81] are not applicable due to the incompleteness of the design at an early design phase when the platform and its resource management policies have not been decided.

Our work is different from all of the above and other existing approaches. Our performance modeling and analysis of a structural model does not assume the availability of information about the platform and software deployment. This makes the model more abstract, and can therefore be used during an early design phase before implementation details are determined. On the other hand, our analysis based on bound estimations provides a rich set of information on individual performance of transactions, and can be used to make quantitative evaluation of different design choices.

3.6 Summary

In this chapter, we have presented a set of methods that can be used to analyze the performance of designed ECSW at an early design phase before knowing its execution environment and deployment. Such performance is modeled as bound estimations, which compute the best-case and worst-case performance by exploring different structural configurations. The performance metrics evaluated in this work include both end-to-end response delays of the transactions and system resource demands. The structural model is first transformed to a set of directed acyclic weighted graphs of transactions with single-input and single-output. The cycle elimination is achieved by separating the inner cycles for a transaction, relinking the feedback link to a dummy component, and assigning a new invocation rate for it. A transaction with multi-input and multi-output can be converted by creating a dummy component for start and a dummy component for end. The end-to-end response delay bounds of the transactions are then derived by exploring best-case and worst-case configurations with assumptions of ideal execution environments. As finding a best-case configuration is NP-hard, we developed a heuristic algorithm to find the best-case configuration and corresponding end-to-end response delay. Since the worst-case delay bound does not contribute to the system design, we leave our end-to-end delay bound as an open range with only the lower bound (best-case delay) derived. The bounds of total system resource demands are also derived with consideration of all concurrent active transactions. We have also demonstrated the use of these estimation bounds in the system design, especially in the software architecture design and comparison, and in the platform design.

CHAPTER 4

Performance-Aware Runtime Model Generation

Given a proper software architecture and platform configuration obtained using the methods in Chapter 3, we now present a method of runtime model generation. Runtime model generation is a model transformation process that refines a structural model with more implementation details to run on a given platform model. A runtime model, as defined in Definition 2.7 in Chapter 2, contains the system detail for implementation, such as the structure of OS-level processes/threads, their execution locations, and execution constraints. These details are further used to derive the schedule of the ECSW at runtime. The runtime model is essential for detailed performance analysis and system implementation.

The performance-aware runtime model generation is a necessity in the ECSW design. Taking into consideration the other design factors, it is common that the designed platform provides resources between the minimum and maximum resource demands. For example, the platform may contain more communication resource but less computation resource due to the high cost of computation devices. In such a case, the platform cannot provide an ideal execution environment, and some configuration of the structural model may lead to violation of the constraints, either the resource constraints of the platform or the timing constraints of the system. Therefore, a method that will find a runtime model meeting all constraints is needed. This is accomplished through a performance-aware model transformation.

In this Chapter, we consider the generation problem as a constraint satisfaction problem instead of an optimization problem, meaning all solutions that meet the specified constraints are considered equivalent. Since the optimality of a solution is application-specific and varies dramatically with different design constraints, it is more flexible to let the designer apply the method repeatedly with refined constraints to find an optimal solution with his/her own objectives and constraints for an application.

4.1 Problem Statement and Transformation Process

4.1.1 Problem statement

Given a structural model M_s and a platform model M_p, a large number of runtime models can be generated by allocating the components and links in M_s to the devices in M_p, forming the processes/threads using the components on the each device, and assigning the timing attributes for the processes/threads. However, not all thus-generated runtime models will meet the resource constraints in R of M_p and performance constraints in H of M_s. To this end, we define the *valid runtime models* to refer to those that meet the constraints.

Definition 4.13. *Given a platform model M_p, a runtime model M_r is* valid *on M_p if M_r satisfies the following:*

- *For any task $\tau_i \in \tau$ in M_r, there exists one and only one device $PROC_i \in PROC$ in the platform model, such that $loc(\tau_i) = PROC_i$. This ensures each and every task runs on one and only one computation device.*
- *For any platform component $\chi \in PROC \cup NW$, the resource consumptions $\sum w_x$ of activities (computation or communication or storage) of M_r allocated on P does not exceed*

the platform capacity $R_x(\chi)$. This ensures the device has enough resource to execute the software allocated on it.

- *For any $\tau_i \in \tau$ in M_r, all constraints of P and D must be met. Meeting the performance constraints of all τ_i ensures M_r and its original structural model M_s meet the same set of end-to-end timing constraints.*

In this definition, all tasks are assumed to be statically allocated on the platform devices, meaning a task does not migrate from one device to another once it is allocated. The second and third conditions guarantee the original specified performance constraints are met in the runtime model. This can be achieved through properly assigning the other parameters of tasks in the runtime model, including execution location loc and release offset o.

Our problem of runtime model generation can be formally defined as a problem of finding a valid runtime model through graph transformation.

Given a structural model in $M_s = (m, T, \Phi)$ and a platform model in $M_p = (PROC, NW, R)$, find a runtime model, $M_r = (\tau, L, P, D, o, w, loc)$, such that:

C1 : For any component $c_u \in \cup_{T_i \in T} C(T_i)$ ($C(T_i)$ are the components in transaction T_i), there exists one and only one task $\tau_p \in \tau$ such that $c_u \in \tau_p$. This implies that every component in M_s is mapped to one and only one task in M_r.

C2 : For any link $(c_u, c_v) \in \cup_{T_i \in T} L(T_i)$ ($L(T_i)$ are the links in transaction T_i), there is a unique link $l(u,v) \in L$ if and only if the two components, c_u and c_v, are in different tasks with $c_u \in \tau_p$, $c_v \in \tau_q$ and $\tau_p \neq \tau_q$. This implies only those connections between the components in different tasks are maintained in M_r.

C3 : M_r is valid on M_p. This implies M_r meets the resource constraints R and performance constraints $P \cup D$, thus consequently meets the constraints H in M_s, i.e.,

$$P \cup D \Rightarrow H.$$

In the problem statement, C1, C2 and C3 ensure the equivalence of the structural model M_s and generated runtime model M_r. C1 and C2 guarantee M_r performs the same functions and behaviors as M_s with the system workloads of M_r no greater than those of M_s, while C3 guarantees M_r meets all resource and performance constraints of M_p and M_s.

Theorem 4.3. *Given a structural model M_s and a platform model M_p, the problem of finding a valid runtime model of M_s on M_p is NP-hard.*

Proof. We use restriction in this proof, by showing an existing NP-complete problem is a special case of our partition graph finding problem. The selected NP-complete problem is the Macro-Dataflow Partition Problem (MDPP) [71]. The problem is defined as:

> Given a directed acyclic weighted graph G with both node weights and link weights standing for their costs, a multiprocessor system with M processors, and a cost bound $CB \in Q_0^+$, find a partition of G, $\Pi(G) = (PN, TP)$, where PN are nodes, TP are tasks, such that the sum of costs of TP on the longest path, $F(\Pi)$, is less than the cost bound CB, i.e., $F(\Pi) \leq CB$.

We restrict our problem of finding a valid runtime model as follows:

1. let a transaction model T_i in M_s be G;
2. let the number of identical devices in the platform be M, $|PROC| = M$;
3. let the deadline D_i of T_i be the cost bound $D_i = CB$;
4. let the F_c and F_l be node and link cost of G;
5. let the tasks in the runtime model be TP, $\tau = TP$;

6. R_c, R_m, R_l are infinity.

With the above mapping, the MDPP is a special case of our problem of finding a valid runtime model. If there exists a solution to the MDPP, we have $F(\Pi) \leq CB$. Consequently, a runtime model M_r will meet its performance constraints of P and D_i in H. As R_c, R_m, and R_l are infinity, the platform resource constraints are met. Thus M_r is a valid runtime model. On the other hand, if M_r is valid, the task chain must meet deadline D_i. As $\tau = TP$ with $F(\tau_i) = F(TP_i)$, we have $F(\Pi) \leq D_i = CB$. So we can conclude that a valid runtime model can be found for a given M_s and M_p if and only if the MDPP has a solution. Since the macro-dataflow partition problem is NP-complete, our valid runtime model problem is also NP-complete.

$\square$

Theorem 4.3 shows that the runtime model generation problem is computationally expensive as the number of components, links, constraints, and types of resources increases. As the ECSW usually contains a relatively large number of components and links, the method for the runtime model generation must be scalable to be useful for practical problems. To this end, we will divide the problem into a set of sub-problems that can be solved in sequence with effective heuristics.

4.1.2 Transformation process

The runtime model generation, as a model transformation process that converts a structural model to a runtime model, can be achieved using graph theories since all models in our definition are represented in graphs. Specifically, we can partition the structural model into a set of component groups that fit in the platform devices. As the components are statically assigned, the component groups can be further partitioned into tasks that can be individually scheduled. The properties, such as execution locations, release offsets, and invocation rates, of these component groups are assigned along with the partition. Corresponding dependencies and communications among the components are transformed into dependencies and communications among the groups, and are mapped to the communication links in the platform model. We therefore introduce a *partition graph* concept to achieve the model transformation.

Definition 4.14. *A* partition graph $M_{pn} = (PN, LN, FN)$ *of a structural model* M_s *is a weighted directed graph, where*

- PN *is the set of partitions. For any component* $c_u \in \cup_{T_i \in T} C(T_i)$ *in* M_s*, there exists one and only one* PN_i *such that* $c_u \in PN_i$*;*
- $LN \subset PN \times PN$ *is a set of directed links between the partitions. A link* $(PN_i, PN_j) \in LN$ *exists if and only if for a link* $(c_u, c_v) \in L$ *of* M_s*, the components* $c_u \in PN_i$*,* $c_v \in PN_j$*, and* $PN_i \neq PN_j$*;*
- $FN : PN \cup LN \to Q_0^+$ *is a set of functions for the resource demands of the partitions and links.*

In this work, the resources we consider include computation, communication, and memory resources. The resource demands of a partition PN_i can therefore be derived as follows:

- Computation: $FN_c(PN_i) = \sum_{c_u \in PN_i} F_c(c_u)$.
- Communication: $FN_l(PN_i, PN_j) = \sum_{l_{uv} \in LN} F_l(l_{uv})$.
- Memory: $FN_m(PN_i) = \sum_{c_u \in PN_i} F_m(c_u)$.

Given an arbitrarily generated partition graph M_{pn}, the following theorem shows the conditions required for M_{pn} to be a valid runtime model.

Theorem 4.4. *Given a partition graph* M_{pn} *of a structural model* M_s*, and a platform model* M_p*, a runtime model constructed using* M_{pn}*,* $M_r = (PN, LN, P, D, o, FN, loc)$ *is a valid runtime model if and only if*

1.

$$P : PN \rightarrow r$$

$$D : PN \rightarrow Q_0^+$$

$$o : PN \rightarrow Q_0^+$$

$$loc : PN_i \rightarrow PROC_j, \text{ for any } PN_i \in P$$

such that H of M_s are met;

2. For any device $PROC_j$ and all partitions PN_i on it,

$$\sum_{PN_i \in PROC_j} FN_c(PN_i) \leq R_c(PROC_j)$$
$$\sum_{PN_i \in PROC_j} FN_m(PN_i) \leq R_m(PROC_j)$$

and

$$\sum_{LN_i \in NW_j} FN_l(LN_i) \leq R_l(NW_j)$$

Proof. The proof is straightforward using the prove-by-construction method and following the run-time model definition. First, a one-to-one mapping can be constructed between PN, LN, FN of a partition graph and τ, L, w of a runtime model:

- $PN \leftrightarrow \tau$;
- $LN \leftrightarrow L$;
- $FN \leftrightarrow w$.

Prove sufficiency. If there exist functions *loc*, *o*, P, and D that meet the constraints H of M_s, we can construct the runtime model as $M_r = (M_{pn}, P, D, o, loc)$. Thus-generated M_r satisfies the definition of a valid runtime model since (i) loc defines a unique $PROC_j$ for any PN_i, (ii) H is met with P, D, and o, and (iii) resource demands of τ and L in M_r are within the capacities of M_p according to condition 2. Therefore, all three conditions in Definition 4.13 are met. So M_r is a valid runtime model.

Prove necessity. If M_r is a valid runtime model, according to Definition 2.7, we have

$$P : PN \rightarrow r$$

$$D : PN \rightarrow Q_0^+$$

$$o : PN \rightarrow Q_0^+$$

$$loc : PN_i \rightarrow PROC_j \text{ for any } PN_i \in P$$

Since the M_r is valid, the function D and P must be assigned with the H met. This results in the condition 1 holds. The FN on each device must also be less than its capacity, which results in the condition 2 holds.

□

Theorem 4.4 indicates that generating a valid runtime model from a structural model can be divided into two parts: finding a proper partition graph and determining the functions for the partitions' attributes. To this end, we develop a multi-step process to generate a partition graph with the proper attributes to implement the model transformation. This process consists of interleaved two partition steps and three assignment steps At each partition step, the model is transformed into a partition graph with more smaller partitions to reduce resource consumptions and increase flexibility in order to meet the resource constraints. At each assignment step, the execution attributes such as timing and location properties are determined for each partition according to the end-to-end performance constraints.

The steps of our transformation process are described below.

STEP 1: Rate assignment. This is an assignment step and the first step of the process. The objective of this step is to determine the invocation rate of every component in the structural model. The derived component's rate is used to compute the workload introduced by each component during the component allocation to meet the resource constraints of the devices in the platform model.

STEP 2: Component allocation. This is a partition step. The objective of this step is to generate a partition graph meeting the resource constraints of the platform. In this step, the components in the structural model are partitioned into groups and allocated to the devices in the platform model with consideration of resource consumption. The allocation is based on both the component resource demands and the device resource capacity. The resultant model is a partition graph with the component group on each device considered as a partition. After this step, we can determine the true resource demands of the components and use them instead of virtual resource demands in the later transformation and analysis.

STEP 3: Timing assignment. This is an assignment step. The objective of this step is to determine the timing constraints for the components in each partition after the communication delays among the partitions are known. The components' timing constraints are determined according to the system-level end-to-end timing constraints. Satisfying such components' constraints results in meeting the end-to-end constraints of the original structural model. These results can be used to determine the execution order of the components in different transactions in the next step of task formation.

STEP 4: Task formation. Task formation is a partition step. The objective of this step is to reduce the resource consumptions of the system software. In this step, each partition on a computation device is further divided into smaller component groups that are schedulable by the system software. The resultant model is a refined partition graph in which the components in each partition are sequenced. In this work, we assume the application runs directly on the operating system, so the tasks can be implemented directly as OS-level processes or threads.

STEP 5: Task graph generation. This is an assignment step, and the final step of the transformation process. In this step, the final runtime model is generated as a partition graph with the partitions being the tasks. All corresponding attributes of the tasks, including the periods, release offsets, deadlines, and inter-task communications and dependencies are specified in the resultant model.

The transformation process with multiple steps helps separate design concerns. In each step, only a small number of constraints are considered. The designer, therefore, can focus on a small number of choices and criteria to optimize the design. However, since we separate the process into multiple steps in which only part of the system properties are considered at each step, some solutions leading to global optimization may be eliminated during the transformation. The resultant runtime model is thus only sub-optimal. This is acceptable under our assumption that all runtime models meeting the constraints are equivalent.

The algorithms used in each step in the transformation process are discussed in detail in the following sections.

4.2 Rate Assignment

The rate assignment determines the invocation rate of every component in the structural model. Knowing the invocation rate of a component is essential for computing the component's resource demands. Among the resources we consider in this work, only computation and communication resource demands are affected by the invocation rates of the components. Memory resource, on the other hand, is assumed to be pre-allocated for each component, and does not change dynamically during the execution.

Our rate assignment is based on the rate propagation technique. The rate propagation traverses the transactions by following the synchronous communications among the components from the input component to the output component. As defined in Chapter 2, the invocation rates are specified for the input of the transactions in the structural model. A transaction starts its execution from its input components, and their outputs trigger a chain of executions of dependent components downstream until the control commands are generated and sent out by the output components. Therefore, the invocation rate assigned to the input components of a transaction indicates the execution rate of all components in the transaction. By tracing each transaction downstream from its input along its synchronous communications, we can derive the rate of all components for each transaction.

An issue that needs to be solved in the rate propagation is the existence of *synchronized components*, whose executions depend on multiple inputs arriving at different rates. Resolving the invocation rate of a synchronized component requires a designer-specified policy, such as the lowest rate or the i-th highest rate among all inputs. For simplicity of discussion without losing generality, we have assumed that the invocation rate of such a synchronized component is the lowest rate of its inputs in the algorithm presentation, which implies the synchronized component is invoked only when all of its inputs arrive.

Algorithm 4.3 shows the rate assignment algorithm. It assigns the invocation rate to all components by following the synchronous communication links in each transaction. It uses a coloring mechanism to mark visited components in the model and finalize the assignments. To distinguish the execution rate of a component under different modes, we use a $< mode, rate >$ pair to indicate the rate assigned to each component.

In Algorithm 4.3, the components' rates are assigned under different system modes. For each active transaction under mode m_i, the input components are sorted in an ascending order of their invocation rates and are colored in WHITE, indicating they have not been visited. From the input

Algorithm 4.3 *Rate assignment.*

```
INPUT: structural model $M_s$ with the rates $r_i$ specified
    for all input components $c_i^{in}$;
OUTPUT: $M_s$ with rate assigned for each mode.
BEGIN
1  foreach mode $m_i \in m$ do
2    foreach transaction $T_i \in \Phi(m_i)$ do color all $c_i \in T_i$ in $WHITE$;
3    sort input components $c_i^{in}$ in $T_i$ in ascending order of its $r_i$;
4    foreach input component $c_i^{in} \in T_i$ do
5      DFS_ASSIGN($c_i^{in}, r_i, m_i$);
6    end-foreach
7  end-foreach
8  return $M_s$;
END.

  DFS_ASSIGN($c, r, m$)
1  BEGIN
2    color $c$ in $GREY$;
3    assign $(m, r)$ as the rate of $c$;
4    foreach $c_s$ is immediate successor of $c$ do
5      if $c_s$ is $WHITE$ then
6        DFS_ASSIGN($c_s, r, m$);
7    end-foreach
8    color $c$ in $BLACK$;
 END.
```

component with the lowest rate, the algorithm traverses forward every component reachable from the input component, assigns a component the same invocation rate as its predecessor, and colors the component in GREY. During the traverse, if a visited component is already colored other than WHITE, which indicates it has been assigned a rate, the algorithm then tracks backward, finalizes the assignment, and colors component in BLACK. As the algorithm starts from the lowest rate and returns when a component has been assigned a rate, the synchronized component with multiple input rates is assigned the lowest rate. In a case where the component should be invoked upon arrivals of the highest rate input, we can sort the input components in descending order of their rates.

4.2.1 Algorithm analysis

The correctness of the Algorithm 4.3 is proved in the following theorem.

Theorem 4.5. *Every component in a structural model M_s is assigned one and only one invocation rate for each mode, which is consistent with the rate of the transaction in which it participates, upon the completion of Algorithm 4.3.*

Proof. The proof is straightforward. For each mode, a component in the active transactions is visited once and only once in the DFS function. This is ensured by the coloring mechanism. Since every component must be reachable from some input component, all components must be visited in the algorithm due to the completeness of the DFS graph traversal function. Further, a successor component inherits the same rate as its predecessor in the algorithm, resulting in the rate of any component along the execution path being consistent with the rate of one of the input components (with the lowest rate). $\square$

The computation complexity of Algorithm 4.3 is as follows. Given m modes with each mode containing t transactions with n_t components and l_t links, the algorithm requires $n_t + l_t$ steps to assign the rate for all components in t. The total number of steps needed for all components assigned for all modes can be computed as $\sum_m \sum_t (n_t + l_t)$. Suppose the structural model M_s contains n components and l links, the steps needed for assignment of each mode can be bounded as $\sum_t (n_t + l_t) \leq (n + l)$. Therefore, the complexity of the algorithm can be bounded by $O(m * (n + l))$. Since m in a system is usually a small constant, the complexity of the algorithm is $O(n + l)$.

4.3 Component Allocation

Component allocation generates an initial partition graph of a structural model by dividing the components into groups according to the devices and their resource availabilities in the platform model. The objective of the component allocation is to transform the structural model to a partition graph meeting the platform resource constraints. Although the designed platform guided by the performance estimations has sufficient resources, it does not specify how to deploy the components so that the resource demand for each and every individual device is not exceeded. The results of the component allocation are the location functions that determine the execution locations for the components.

The inputs of the component allocation algorithm are the structural model and platform model, as defined in Chapter 2. Cycles for closed-loop feedback and multi-rate control in the model are assumed to be eliminated using the method in Chapter 3. The availability of each type of resource is specified in the platform model. The devices in the platform model are allowed to be heterogeneous, and the heterogeneity is modeled as different resource capacities, implying the devices are not identical. The result of the algorithm is a partition graph, containing partitions that satisfy the following:

- each group runs on one and only one computation device with sufficient resources, and
- the total amount of communication between any two groups is within the capacity of the link between the two devices where the groups reside.

Note that the resultant partition model contains only one partition for each device. As before, we consider only computation, communication and memory resources, although the method can be extended to other resource types. During the allocation, we ignore the execution dependencies among the components. This makes the thus-generated results pessimistic, meaning that a solution may exist when our method fails to find one. On the other hand, if our method finds a solution, it is guaranteed to meet all resource constraints. We also ignore other constraints such as end-to-end deadlines and schedulability during the allocation, for they will be addressed in later steps. We further assume that implementation at the later phase follows the component allocation results. This ensures that if such an allocation meets all constraints, so does the subsequent implementation.

With the above assumptions, the component allocation subproblem can be defined as a model transformation problem as follows:

Given a structural model M_s and a platform model M_p, find a partition graph M_{pn} that satisfies

1. each partition PN_i contains some components in $C = \bigcup_{T_i \in T} C(T_i)$ such that (i) for all n partitions, $\bigcup_{i=1}^{n} PN_i = C$, and (ii) for any $i, j, i \neq j$, $PN_i \bigcap PN_j = \emptyset$.

2. there exists a one-to-one mapping $\mathbf{g} : PN_i \rightarrow PROC_i$ such that (i) for any resource r of device $PROC_i$, $FN_r(PN_i) \leq R_r(PROC_i)$, and (ii) $FN_l(LN_{ij}) \leq R_l(NW_{ij})$.

The problem can be viewed as a constraint satisfaction problem, which is NP-complete in general. We thus develop an algorithm using a set of heuristic methods to improve its scalability.

4.3.1 Allocation algorithm

The allocation algorithm is based on *informed branch-and-bound*. Initially, the algorithm creates a set of partitions each of which is dedicated to a device in the platform model. The resource capacities of the partition are assigned to be the resource capacities of the devices (for computation and memory). At each step, an unallocated component is assigned to a partition (branch step). For every unallocated component c_i, we maintain a list of candidate partitions, called *partition domain* $DM(c_i)$, containing all partitions that c_i can be allocated to, without violating any constraint. All partitions PN_i in $DM(c_i)$ are ranked according to a *competence function* $CF(c_i, PN_i)$. At each branch step, the algorithm selects the best partition in $DM(c_i)$ for the allocation of the component a_i (informed). After each allocation, the algorithm adjusts the partition domains of every unallocated component and eliminates the partitions that result in a constraint violation (bound step). This is achieved by a *forward checking* [19] mechanism after a *minimum constraint violation depth* ($MCVD$) is reached. The forward checking ensures that only those partitions meeting all constraints will be further explored in the allocation process. The process continues until either every component is allocated to a partition without violating any constraint, or no allocation can be found for a component subject to all constraints. The algorithm is shown in Algorithm 4.4, which subsequently invokes a recursive function in Algorithm 4.5, to allocate a component at each step.

Algorithm 4.4 *Allocation subject to multiple constraints.*

```
input: a structural model $M_s = (M, T, \Phi)$;
       a platform model $M_p = (PROC, NW, R)$;
output: a partition graph $M_{pn} = (PN, LN, FN)$ with resource constraints satisfied.
BEGIN
1   $MCVD$ is initialized to be $\infty$;
2   create partition $PN_i = \emptyset$ for $PROC_i \in PROC$;
3   $C \leftarrow \bigcup T_i \in TC(T_i)$;  4   order $c_i \in C$ in descending order of the combinational
5     resource consumption $w(c_i)$;
6   foreach component $c_i \in C$ do
7     assign $DM(c_i)$ to be $PN_j$ with $R_r(PN_j) \geq F_r(c_i)$;

8   allocate($C$);

9   if ($C$ is empty) then
10    foreach ($c_i \in PN_j$ and $c_u \in PN_v$ and $l_{iu} = (c_i \to c_u) \in L$ do
11      $LN \leftarrow LN_{jv} = (PN_j \to PN_v)$;
12    return ($succeed, PN$);
13  else
14    return $fail$;
END.
```

One of the key components in this algorithm is the *competence function*. It is designed and used to estimate the combined effect of an allocation and to speed up the algorithm. The function determines the quality of the choice made at each step, which affects the speed of the algorithm. For a component c_i and a partition PN_j $DM(c_i)$, $CF(c_i, PN_j)$ can be computed as follows:

$$CF(c_i, PN_j) = \alpha_1 * C_r(PN_j) + \alpha_2 * L_r(PN_j) + \alpha_3 * M_r(PN_j) \tag{4.1}$$

where C_r, L_r, and M_r are the consumptions of computation, communication, and memory resources, respectively, after allocating c_i to partition PN_j. Since the resource consumptions in

Algorithm 4.5 *Allocate function.*

```
allocate($C$)
BEGIN
1  if ($C$ is empty) then return $(success, PN)$;
2  retrieve first $u$ from $C$ for allocation;
3  if ($DM(u)$ is $\emptyset$) then return $fail$;

4  foreach ($PN_i \in DM(u)$) do
5    compute $CF(u, PN_i)$;
6  sort $DM(u)$ in ascending order of $CF(u, PN_i)$;
7  while ($u$ is unallocated) do
8    if (all $PN_i \in DM(u)$ have been tried) then
9      put $u$ back to $C$;
10       return $fail$;
11     end-if
12     allocate $u$ to $PN_i$ with $min\{CF(u, PN_i)\}$ that has not be tried;
13     if (no constraint is violated by this allocation) then
14       if (total allocated components = $MCVD$) then
15         forward_checking($C$);
16       if (total allocated components $>$ $MCVD$) then
17         minimized_forward_checking($C$);
18       if (any component $x \in C$ with $DM(x)$ is empty ) then
19         return $fail$;
20       allocate($C$);
21       if (function return with $success$) then
22         return $(success, PN)$;
23     else if (allocated components less than $MCVD$) then
24         reset $MCVD$ to the number of allocated components;
25     end-if-else
26   end-while
END.
```

$C_r(PN_j)$, $L_r(PN_j)$, and $M_r(PN_j)$ are of different types and usually are not comparable to each other, we normalize them as follows:

$$\begin{aligned} C_r(PN_j) &= \tfrac{FN_c(PN_j)}{C_{ideal}(PN_j)} - 1 \\ L_r(PN_j) &= 1 - \tfrac{FN_l(LN)}{\sum_{y \in L} F_l(y)} \\ M_r(PN_j) &= \tfrac{FN_m(PN_j)}{M_{ideal}(PN_j)} - 1. \end{aligned} \tag{4.2}$$

$C_{ideal}(PN_j)$ is the ideal computation resource consumption of PN_j, which can be computed according to some pre-defined strategy such as the one proportional to, or even distribution of, total workload over the computation devices. Allocating c_i to a partition PN_j with the minimum $C_r(PN_j)$ complies with the pre-defined strategy. Similarly, we can determine the ideal memory resource consumption $M_{ideal}(PN_j)$ of PN_j. The communication resource consumption of PN_j depends on the total communication in and out of PN_j. A smaller value of $L_r(PN_j)$ indicates less communication resource consumed if allocating component c_i to PN_j.

Constants α_1, α_2 and α_3 in Eq. (4.2) specify the weights of each resource in the competence function. They make the competence function a generic form of allocation strategies. Different strategies can be implemented by choosing the values of α_1, α_2, and α_3. For example, choosing $\alpha_1 >> max(\alpha_2, \alpha_3)$ with $C_{ideal}(PN_j)$ derived from even distribution of the total workload implements the load-balancing allocation strategy. Similarly, choosing $\alpha_2 >> max(\alpha_1, \alpha_3)$ minimizes the communication resource consumption. This is sometimes desired to optimize the response time of a transaction involving a slow communication link because such an allocation can result in the synchronously-communicating components of the same transaction being allocated to the same computation devices.

The algorithm uses the competence function values to assist selection of the best possible allocation of a component. After each allocation, the competence function values are computed for all partitions in every unallocated component's partition domain. Since a smaller competence function value indicates less possibility of violating the constraints, the algorithm first chooses the partition with the minimum competence value. If a future component allocation fails to meet any constraint, the algorithm backtracks and chooses the partition with the next minimum competence value. The process goes on until all partitions in c_i's domain have been explored before declaring a failure of allocating c_i and backtracking other assignments of the components allocated prior to c_i.

Another key component in Algorithm 4.4 is *forward checking*. Forward checking is a process that evaluates the partition domain of every unallocated component and removes the partitions that will result in any constraint violation. The goal of forward checking is to reduce the size of the partition domains of unallocated components. Consequently, it accelerates the algorithm in finding a solution. The effect of removing partitions from a component's domain is local, meaning that the partition domain of a component c_i will be restored to its original one if the assignment of c_i or any component before c_i is backtracked. This ensures the completeness of the algorithm, which guarantees a solution will be found if one exists.

Forward checking incurs additional computation overhead as it must check and manipulate the partition domains of all unallocated components after every assignment. The overhead increases proportionally to the number of unallocated components in the system. Therefore, it is high at early steps of the algorithm when most of the components are unallocated and each component's partition domain contains most of the partitions. This implies that forward checking is beneficial only after a certain number of components have been allocated. Applying forward checking from the beginning of the algorithm is also unnecessary if each component's resource consumptions are much smaller than the available resources of a device/link, which is commonly the case in large embedded systems. So the resource constraints can be violated only after enough components are allocated to a partition. Considering this, we have defined a *minimum constraint violation depth* (MCVD), after which forward checking should be applied. The value of MCVD is assigned as the minimum number of allocated components at which a resource constraint violation occurs. In our algorithm implementation, we initially assign MCVD to infinity and update it when a constraint violation is observed with a total number of components allocated smaller than the current MCVD.

We can further reduce the overhead of forward checking by examining only those partitions whose components are changed. Since one and only one component is allocated at each step, there is only one partition change at each step of the algorithm. Any future resource constraint violation is the result of a partition change. For example, given a current allocating component c

with domain (PN_i, PN_j, PN_k) and an unallocated component u with domain (PN_j, PN_k, PN_l), if c is chosen to be allocated to PN_j, we only need to check PN_j in u's domain for computation resource and memory resource. Other partitions, PN_k and PN_l, in u's domain are maintained after c is allocated on PN_j. In case PN_j in u's domain does not have enough computation or memory resource for u, it should be removed from u's domain. On the other hand, the communication resource constraint cannot be violated if u is allocated to the same partition PN_j as we assume all communications within a partition consume no communication resource. Therefore, the partitions PN_k and PN_l in u's domain need to be checked for communication resource constraints after u is allocated to PN_j, and should be removed from u's domain if allocating u to them violates the constraint. This is implemented as a minimized forward checking function in our algorithm, as shown in Algorithm 4.6.

Algorithm 4.6 *Minimized forward checking.*

```
input: unallocated components $C$ with:
    partition domains $PN$ for each $u \in C$;
    the most recently-assigned component $v$ in $PN_v$;
    constraints on $PN$ and $LN$.
output: reduced partition domains $PN$ for each $u \in C$.
BEGIN
1  foreach $u \in C$ do
2    if $PN_v \in DM(u)$ and $(F_c(u) + FN_c(PN_v) > R_c(PN_v)$ or
3      $F_m(u) + FN_m(PN_v) > R_m(PN_v))$ then
4      remove $PN_v$ from $DM(u)$;
5    if $\sum_{(y=(v,u) \vee (u,v)) \in LN} F_l(y) + FN_l(LN) > R_l$ then
6      remove all partitions but $PN_v$ from $DM(u)$;
7  end-foreach
END.
```

It is shown in [49] that a properly-selected initial order of components can reduce, on average, the number of steps required to find a solution. We, therefore, order the components according to their combinational resource consumptions before allocating any of them. The combined resource consumption is determined as follows:

$$w(c_i) = \beta_1 * \frac{F_c(c_i)}{\sum R_c(PROC_i)} + \beta_2 * \frac{\sum_{y(c_i)} F_l(y)}{RROC_l(L)} + \beta_3 * \frac{F_m(c_i)}{\sum R_m(PROC_i)} \tag{4.3}$$

where β_1, β_2, and β_3 are constants defining the weights of computation, communication, and memory resource; F_c, F_m, and F_l are the component's resource consumption function for computation, memory, and communication resource, respectively; R_c, R_m, and R_l are corresponding resource constraints of a device; $y(c_i)$ represents an incoming or outgoing communication link induced by component c_i. In our algorithm, the components are allocated in descending order of their combinational resource consumptions, implying that a component requiring more resources is allocated first.

4.3.2 Algorithm analysis

The component allocation algorithm transforms the structural model to a partition graph satisfying condition 2 — resource constraints — of the valid runtime model definition. This is proved in Theorem 4.6.

Theorem 4.6. *The partition graph generated by Algorithm 4.4 satisfies*

1. *for all $PN_i \in PN$, $\bigcup_{i=1}^{n} PN_i = C$;*

2. *for any two partitions PN_i and PN_j with $i \neq j$, $PN_i \bigcap PN_j = \emptyset$;*

3. *for any partition PN_i, $FN_x(PN_i) \leq R_x(PROC_i)$ for any resource x.*

Proof. In Algorithm 4.4, the partitions and the computation devices in the platform are one-to-one mapping. At each step of the algorithm, one and only one of the existing partitions is chosen to allocate a component. No new partition is created during the allocation. Therefore, at the end of the algorithm, the number of partitions with one or more components on it, i.e., $|C(PN_i)| > 0$, must satisfy

$$|PN| \leq |PROC|$$

According to the **allocate()** function in Algorithm 4.5, only unallocated components in C are considered. Once a component u is allocated to a partition, it is removed from C. As C contains all components in M_s at the beginning of the algorithm, and is empty upon success when the algorithm is completed, every component must be allocated to some partition; therefore,

$$\bigcup PN_i = C$$

So Property 1 is true.

Since an unallocated component u is removed from C after allocated, and the algorithm only chooses a component from C for allocation, no component can be allocated twice on a successful allocation. When an allocation fails and the algorithm backtracks to component u, it removes all previous allocations after u. The following allocations are a new trial independent of previous allocations. This results in no component being allocated twice upon a success. This implies

$$\forall u \in C, (u \in PN_i) \rightarrow (u \notin PN_j)(i \neq j)$$

which proves Property 2.

At each step, the algorithm allocates a component c_i to a partition PN_i if and only if the consumption FN of PN_i, after allocating c_i to it, does not exceed the resource availability. Therefore, upon the successful return of the algorithm, the resource consumptions of all types of resources of the partition graph should be within the resource availability of the devices or links. This proves Property 3.

□

Theorem 4.6 indicates that a partition graph generated by Algorithm 4.4 has the same workloads as the structural model, and contains partitions that can be executed on the devices in the platform model with the resource constraints met. Therefore, if a runtime model is derived from such a partition graph without introducing additional workloads, the resultant runtime model will surely satisfy the resource constraints. This satisfies condition 2 of the valid runtime model definition. Our later transformation, therefore, only needs to manipulate the resultant partition model generated by Algorithm 4.4 to obtain a valid runtime model.

The computation complexity of Algorithm 4.4 is as follows. The initial steps of creating partitions and computing $w(c_i)$ takes time of $O(n + p)$ for a structural model M_s with n components with p partitions. The **allocate()** function works like a depth-first search in determining an allocation of one component at each step. For each component, there are at most p possible allocations in the worst-case. To allocate n components, the depth-first search algorithm would take worst-case $O(n^p)$. However, with initial component order and forward checking, the solution space has been reorganized at each step, with the solution moving to the front end of the design space tree. For one extreme case with more solution nodes than non-solution nodes in the tree ($F_x(c_i) << R_x(PN_j)$ for any i, j), the algorithm takes $O(n)$ steps to find a solution. For another extreme case with more non-solution nodes than solution nodes in the tree ($F_x(c_i) \sim R_x(PN_j)$), p is significantly reduced at each step as a violation will be found and $DM(c_i)$ will contain a small number of partitions. This is equivalent to reducing the branches of the design space tree. Thus p can be considered as a constant in any case. The complexity of our allocation algorithm is therefore $O(n^c)$, where c is a constant.

4.4 Timing Assignment

The timing assignment distributes the end-to-end timing and performance constraints in H of M_s and assigns them to the components in the partition graph. The objective of such an assignment is to derive the timing information for component schedules in task formation, which will further divide the partitions in the partition graph. The assigned components' timing attributes include the earliest start time (EST) and the latest completion time (LCT). The EST and LCT define a time window for the execution of a component that will result in meeting end-to-end system performance constraints. They will also be used to generate the performance functions of P, D, and o in the runtime model during the task formation.

4.4.1 Timing assignment algorithm

Our timing assignment algorithm is based on the forward and backward tracking of directed acyclic graphs. It traverses the transaction models one-by-one to assign the timing constraints for the components in a transaction. As the assignment determines only the bound of the the allowable execution window for each component, it considers only the precedent constraints in a transaction, and ignores the interferences of concurrent executions of the components in other active transactions under the same system mode. Our timing assignment algorithm determines the execution windows for each component in such a way that the end-to-end performance constraints can be met.

The input of the timing assignment algorithm is the partition graph obtained from component allocation. Since the execution locations have been decided for all components, the resource demands of both computation and communication can be converted from virtual ones to real values. Upon the completion of the timing assignment, all components are assigned with an earliest start time, EST, and latest completion time, LCT. Meeting these component timing constraints will satisfy the end-to-end deadlines D of the transactions. The algorithm of the timing assignment is given in Algorithm 4.7, which invokes Algorithm 4.8 and 4.9 to assign EST and LCT, respectively.

Algorithm 4.7 first invokes **assign_forward()** function shown in Algorithm 4.8 to determine EST for all components in a transaction. Algorithm 4.8 traverses the transaction graph forward from an input component, exploring every component whose predecessors have all been visited.

Algorithm 4.7 *Timing assignment algorithm.*

```
INPUT: partition graph $M_{pn} =< PN, LN, FN >$;
       end-to-end deadlines $D$ in constraint set $H$ of $M_s$;
       rate $r$ of each input component.
OUTPUT: assigned $EST(a_i)$ and $LCT(a_i)$ for each component $a_i$.
BEGIN
 1  foreach transaction $T_i \in T$ do
 2      assign $EST(a_i) = 0$ for all components $a_i$;
 3      foreach input component $a_{in}$ do
 4          assign $EST(a_{in})$ variation bound $\Delta_{in} = 0$;
 5          $Q \leftarrow a_{in}$;
 6      end-foreach
 7      assign_forward($Q$);
 8      assign $LCT(a_i) = \infty$ for all components $a_i$;
 9      foreach output component $a_{out}$ do $Q \leftarrow a_{out}$;
10      assign_backward($Q$);
11  end-foreach
12  return $success$;
END.
```

The $EST(a_i)$ of a component a_i can be determined iteratively as follows:

$$EST(a_i) = \begin{cases} max_j\{EST(a_{i-1,j}) + F_c(a_{i-1,j}) + F_l(a_{i-1,j}, a_i)\} & \text{if } a_i \text{ is not an input component} \\ 0 & \text{if } a_i \text{ is an input component} \end{cases} \tag{4.4}$$

where $a_{i-1,j}$ is the jth immediate predecessor of a_i, $EST(a_i)$ is the earliest start time of a_i, $EST(a_{i-1,j})$ is the earliest start time of the jth immediate predecessor of a_i, F_c is the computation resource demand (execution time) of the component, and $F_l(a_{i-1,j}, a_i)$ is the communication cost (delay) of the link from the component's jth predecessor $a_{i-1,j}$ to the component a_i. Since M_s has be transformed to M_{pn} with each partition dedicated to a computation device in M_p, the real resource demand in the form of time delays can be derived for F_c and F_l. To simplify our discussion while still considering the practices in the current ECSW system, we assume F_l satisfies:

$$F_l(a_i, a_j) = \begin{cases} F_l(a_i, a_j) & \text{if } a_i \text{ and } a_j \text{ are in different partitions} \\ 0 & \text{if } a_i \text{ and } a_j \text{ are in the same partition} \end{cases}$$

This implies that the communication resource demands between the components on the same device

Algorithm 4.8 *assign_forward() function.*

```
INPUT: ready-to-explore components in $Q$;
OUTPUT: $EST$ and maximum $EST$ variation $\Delta_i$ assigned for all components.
BEGIN
 1  while $Q \neq \emptyset$ do
 2    $a_c \leftarrow head(Q)$;
 3    foreach $a_i \in succ(a_c)$ do
 4      if variation bound $\Delta_i$ has not be assigned then
 5        if $a_i$ is a synchronized component then
 6          compute maximum variation $\Delta_i$;
 7        else $\Delta_i = \Delta_c$;
 8      end-if
 9      if $EST(a_i) < EST(a_c) + F_c(a_c) + F_l(a_c, a_i)$ then
10        $EST(a_i) \leftarrow EST(a_c) + F_c(a_c) + F_l(a_c, a_i)$;
11        if all $a_i$'s incoming links visited then $Q \leftarrow a_i$;
12      end-if
13    end-foreach
14  end-while
END.
```

are negligible.

The max operation indicates that the $EST(a_i)$ is the latest time of all its predecessors' outputs. This is because of the assumption that a synchronized component starts only after all its inputs are available. Algorithm 4.8 achieves this by initializing the EST to zero. All components whose predecessors have all been visited are added to a queue. The component at the head of the queue a_c is explored. If a component a_i is a_c's successor and has been assigned with an EST greater than the new value from a_c, it indicates that a_i can be reached from another path with longer time. The function then ignores the current path and explores another successor, if one exists. If EST of a_i is smaller than the value through a_c, indicating that it has been reached from a shorter path, the EST must be updated with the new value through a_c. If all a_i's predecessors have been visited, the assigned $EST(a_i)$ is the final one for a_i, and a_i is appended to the tail of Q.

Algorithm 4.8 also computes the maximum variation of $EST(a_i)$ of each component a_i, denoted as Δ_i. Such EST variations may be caused by different rates of multiple input components. The multiple input rates result in the EST of a synchronized component a_s varying as the arrival

Algorithm 4.9 *assign_backward() function.*

```
INPUT: ready-to-explore components in $Q$;
OUTPUT: $LCT$ for all components;
BEGIN
1  while $Q \neq \emptyset$ do
2    $a_c = head(Q)$;
3    foreach $a_i = prec(a_c)$ do
4      if $LCT(a_i) > LCT(a_c) - \sum_{a_c \prec a_j \wedge loc(a_j)=loc(a_c)}(F_c(a_j)) - F_c(a_c) - F_l(a_c, a_i)$ then
5        $LCT(a_i) \leftarrow LCT(a_c) - \sum_{a_c \prec a_j \wedge loc(a_j)=loc(a_c)}(F_c(a_j)) - F_c(a_c) - F_l(a_c, a_i)$;
6        if all $a_i$'s outgoing link visited then $Q \leftarrow a_i$;
7      end-if
8      if $LCT(a_i) < EST(a_i) + \Delta_i$ then return $fail$;
9    end-foreach
10 end-while END.
```

times of all a_s's inputs vary. The variation bound of any component a_i can then be computed as

$$\begin{aligned} \Delta(a_i) &= max_{0 \leq k \leq N_i}(k \cdot P_i + o_i - min_{1 \leq j \leq m}(\lfloor \tfrac{k \cdot P_i}{P_j} \rfloor \cdot P_j + o_j)) \\ N_i &= \tfrac{LCM(P_1, \ldots, P_m)}{P_i} - 1 \end{aligned} \tag{4.5}$$

where $LCM(P_1, \ldots, P_m)$ is the least common multiple of all invocation periods, and o_i, o_j are the release offsets of input a^i_{in} and a^j_{in}, respectively. Since $\Delta_(a_i)$ depends only on rates and release offsets of inputs, which are fixed parameters in a design, Algorithm 4.8 performs Δ_i computation for each component only once.

The $LCT(a_i)$ of a component a_i is determined similarly by Algorithm 4.9 from the output components of every transaction. The $LCT(a_i)$ can be determined iteratively as follows:

$$LCT(a_i) = \begin{cases} min_j\{LCT(a_{i+1,j}) - \sum_{a_k \in ds(a_{i+1})} F_c(a_k) - F_c(a_{i+1,j}) - F_l(a_i, a_{i+1,j})\} & \text{if } a_i \text{ is not an output component} \\ D_{out} & \text{if } a_i \text{ is an output component} \end{cases} \tag{4.6}$$

where $a_{i+1,j}$ is the jth immediate successor of a_i in M_s, $ds(a_{i+1})$ are the concurrent components which depend on a_{i+1} and reside on the same device with a_{i+1}, and D_{out} is the end-to-end deadline defined in H of a transaction. Considering ds in the computation is due to the fact that all components depending on a_{i+1} and allocated on the same device must be executed sequentially. The min operation indicates that the $LCT(a_i)$ is the earliest time when all its predecessors are completed and their outputs arrive at the component's inputs because of the assumption of a synchronized component. In our algorithm, the components' LCTs are initialized to be the deadline D_{out} of the transaction. Algorithm 4.9 then traverses backward from the output components along the component dependencies in the transaction model. It ignores the component that has been assigned LCTs smaller than the one from the current path, and updates those with greater LCTs. The assigned LCTs of all upstream components must be updated after a component LCT is changed.

Algorithm 4.9 further validates the EST and LCT assignment after both are derived. The validation is done through comparing $LCT(a_i) - EST(a_i)$ and $F_c(a_i)$ for a_i (as well as for links between partitions) with consideration of the EST variations resulting from different input rates. Given the EST, Algorithm 4.9 checks whether the execution time window is longer than the component resource demand under the worst-case arrival. If any component a_i is found with $LCT(a_i) < EST(a_i) + F_c(a_i) + \Delta_i$, the algorithm returns $fail$, indicating no feasible assignment can be found.

The timing assignment algorithm is different from the basic deadline distribution algorithm in [38], although both of them can be used for the timing assignment. The deadline distribution algorithm needs to first identify the critical path (the longest execution path) of each transaction, and distribute the end-to-end deadline D_t of the transaction over the components and links on the critical path. Given n components and m links on the critical path P, each component/link introduces a delay of $F_c(a_i)$/$F_l(a_i, a_{i+1})$, the $EST(a_i)$ and $LCT(a_i)$ of the n components and m links can be computed as follows:

$$s = \frac{D_t - \sum_{c_i \in P} F(c_i)}{n+m}$$

$$EST(c_i) = LCT(c_{i-1}) \tag{4.7}$$

$$LCT(c_i) = EST(c_i) + F(c_i) + s$$

where c_i can be either a component a_i or a link (a_i, a_{i+1}) between a_i and its immediate successor. F is F_c or F_l depending on whether its input is a component or a link. The s is the average slack for each component/link. A component/link c_{i-1} represents the immediate predecessor of the link/component on P. After ESTs and LCTs are assigned for all components and links on the critical path, the ESTs and LCTs of components/links on other paths can be assigned as follows:

$$\begin{aligned} EST(c_i) &= max_j\{LCT(c_{i-1,j})\} \\ LCT(c_i) &= min_j\{EST(c_{i-1,j})\} \end{aligned} \tag{4.8}$$

It can be seen from Eq. (4.7) and (4.8) that the execution time windows (EST, LCT) generated by deadline distribution for any components with precedent constraints do not overlap. Such non-overlap execution windows of sequential components may make it easy to determine the execution order of components in a task. However, as the component sequencing with release times and deadlines is NP-complete, such restriction do not help for large system as the concurrency exists among components on parallel execution paths and in different concurrent transactions. On the other hand, as the deadline distribution limits the execution window to non-overlap, the scheduling overhead of a system with the assigned times is higher than the scheduling overhead of a system with the assignments generated by our timing assignment algorithm.

Theorem 4.7. *The timing assignment generated by Algorithm 4.7 results in a more feasible schedule than the assignment generated by the deadline distribution.*

Proof. We prove this by showing that any feasible schedule generated by a deadline distribution assignment is valid with an assignment from Algorithm 4.7, but not vise versa.

We first show $EST_o(a_i) \leq EST_d(a_i)$ for any component a_i where EST_o and EST_d are the assignments generated by our algorithm and deadline distribution, respectively.

case 1: a_i is an input component. $EST_o(a_i) = EST_d(a_i) = 0$.

case 2: a_i is on the critical path.

$$\begin{aligned} EST_o(a_i) &= EST_o(a_{i-1}) + F_c(a_{i-1}) + F_l(a_{i-1}, a_i) \\ &\leq EST_o(a_{i-1}) + F_c(a_{i-1}) + s + F_l(a_{i-1}, a_i) + s \\ &= EST_d(a_i) \end{aligned}$$

case 3: a_i is not on the critical path.

$$\begin{aligned} EST_o(a_i) &= max\{EST_o(a_{i-1,j} + F_c(a_{i-1,j}) + F_l(a_{i-1,j}, a_i)\} \\ &\leq LCT_d(a_{i-1}) \\ &= EST_d(a_i) \end{aligned}$$

Similarly, we can show $LCT_o(a_i) \geq LCT_d(a_i)$ for the cases that (i) a_i is an output component ($LCT_o(a_i) = LCT_d(a_i) = D_t$), (ii) a_i is on the critical path, and (iii) a_i is not on the critical path.

According to the above discussion, we can conclude that for any component a_i,

$$EST_o(a_i) \leq EST_d(a_i) < LCT_d(a_i) \leq LCT_o(a_i)$$

This indicates that the execution window generated by the deadline distribution for a component is a sub-interval of the execution window generated by the timing assignment algorithm. Therefore, if a schedule with the deadline distribution assignment is feasible, so is one with the timing assignment generated by Algorithm 4.7. On the other hand, a feasible schedule with the assignment generated by Algorithm 4.7 may not be feasible for the deadline distribution assignment. $\square$

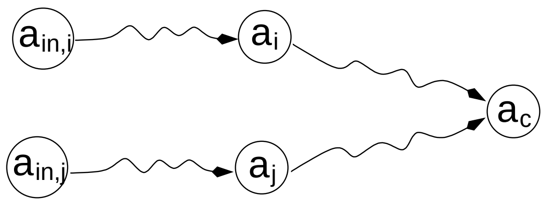


Figure 4.1: Path (a_i, a_c) and (a_j, a_c) are both single sequential paths without any common component.

4.4.2 Algorithm analysis

The timing assignment algorithm assigns the start times and completion times of the components in the structural model. The correctness of the algorithm depends on whether a runtime model meeting the derived constraints will satisfy all end-to-end timing constraints. We will prove this by showing the EST and LCT generated by Algorithm 4.7 is a sufficient and necessary condition for any valid feasible schedule. To do so, we need to prove the derived EST is minimum and its variation can be bounded.

Lemma 4.1. *Suppose a synchronized component a_c with two distinct input paths. If the release phase of the two paths is bounded by δ, the variation of $EST(a_c)$ is at most δ.*

Proof. Suppose there are two paths to a_c, as shown in Figure r4.1.

Denote the time delays of the path (a_i, a_c) as $F_{i,c}$ and (a_j, a_c) as $F_{j,c}$. Without losing generality, we assume $EST(a_j)$ is δ later than $EST(a_i)$. According to Eq. (4.4), the $EST_0(a_c)$ under condition of a_i and a_j released at the same time is

$$
\begin{aligned}
EST_0(a_c) &= max(EST(a_i) + F_{i,s}, EST(a_j) + F_{j,c}) \\
&= \begin{cases} EST(a_i) + F_{i,c} & \text{if } EST(a_i) + F_{i,c} > EST(a_j) + F_{j,c} \\ EST(a_j) + F_{j,c} & \text{otherwise} \end{cases}
\end{aligned}
\tag{4.9}
$$

If $EST(a_j)$ is with offset $x \leq \delta$, the $EST(a_c)$ then becomes

$$\begin{aligned} EST(a_c) &= max(EST(a_i) + F_{i,c}, EST(a_j) + F_{j,c} + x) \\ &= \begin{cases} EST(a_i) + F_{i,c} & \text{if } EST(a_i) + F_{i,c} > EST(a_j) + F_{j,c} + x \\ EST(a_j) + F_{j,c} + x & \text{otherwise} \end{cases} \end{aligned} \tag{4.10}$$

According to Eq. (4.9) and (4.10), we can conclude that

$$EST(a_c) - EST_0(a_c) \leq \delta$$

$\square$

Lemma 4.1 shows that the variation of EST of a synchronized component is bounded by the arrival time difference of any two inputs. Now we need to show that the minimum EST of a synchronized component occurs when all inputs have the same arrival time.

Lemma 4.2. *The minimum $EST(c_i)$ of a component c_i occurs when all the input components that can reach c_i are released at the same time.*

Proof. If c_i is reachable from only one input component c_{in}, we have

$$\begin{aligned} EST(c_i) &= EST(c_{i-1}) + F(c_{i-1}) + F(c_{i-1}, c_i) + \delta(c_{in}) \\ &= \sum_{cin \prec c_k \preceq c_i} (F(c_{k-1}) + F(c_{k-1}, c_k)) + \delta(c_{in}) \end{aligned} \tag{4.11}$$

where c_{i-1} is the immediate predecessor or c_i in the transaction T, and $\delta(c_{in})$ is the input release difference between c_{in} and the earliest released input.

Obviously, $EST(c_i)$ is minimum because c_{in} is the only input component, $\delta(c_{in}) = 0$, which implies c_{in} released at the same time as the earliest released input component.

If c_i is reachable from $m \geq 2$ input components $c_{in}^1, \ldots, c_{in}^m$, released at time point $t_1, \ldots, t_m$, assume T starts at t_0, we have

$$\delta(c_{in}^k) = t_k - t_0, (1 \leq k \leq m)$$

According to Eq. (4.4), we have

$$\begin{aligned} EST(c_i) &= max_{1 \leq k \leq m}(EST(c_{i-1}^k) + F(c_{i-1}^k) + F(c_{i-1}^k, c_i) + \delta(c_{in}^k)) \\ &\geq max_{1 \leq k \leq m}(EST(c_{i-1}^k + F(c_{i-1}^k) + F(c_{i-1}^k, c_i)) \end{aligned}$$

This indicates $EST(c_i)$ is minimum when $\delta(c_{in}^k) = 0$ for all $1 \leq k \leq m$, i.e., when all input components are released at the same time.

Therefore, we can conclude the minimum $EST(c_i)$ for any c_i occurs when all its inputs are released at the same time.

□

We now prove the EST variation of a synchronized component a_s can be bounded, and the bound can be derived from input rates and release offsets.

Lemma 4.3. *Given a component c_i whose invocation rate is the same as input c_{in}^i ($P_i = 1/r(c_{in}^i)$), and is reachable from $m \geq 1$ input components $c_{in}^1, \ldots, c_{in}^m$ with different input period $P_1, \ldots, P_m$ ($P_k = 1/r_k, 1 \leq k \leq m$), released offsets $o_1, \ldots, o_m$. $EST(c_i)$ variation is bounded by*

$$\Delta(c_i) = max_{0 \leq k \leq N_i}(k \cdot P_i + o_i - min_{1 \leq j \leq m}(\lfloor \frac{k \cdot P_i}{P_j} \rfloor \cdot P_j + o_j))$$
$$N_i = \frac{LCM(P_1, \ldots, P_m)}{P_i} - 1$$

where $LCM(P_1, \ldots, P_m)$ is the least common multiple of all invocation periods.

Proof. Since a transaction is considered to start at the time of its earliest input, the release time of transaction T_i when the k-th invocation of c_{in}^i is released is

$$S^k(T_i) = min_{1 \leq j \leq m}(\lfloor \frac{k \cdot P_i}{P_j} \rfloor \cdot P_j + o_j)$$

The release time of c^i_{in} is

$$S^k(c^i_{in}) = k \cdot P_i + o_i$$

The release phase for the k-th invocation of c^i_{in} is

$$\begin{aligned}\Delta^k(c^i_{in}) &= S^k(c^i_{in}) - S^k(T_i) \\ &= k \cdot P_i + o_i - min_{1 \le j \le m}(\lfloor \frac{k \cdot P_i}{P_j} \rfloor \cdot P_j + o_j)\end{aligned}$$

The variation is bounded by the maximum value among all k. Since the release phase will repeat after $k \cdot P_i$ greater than $N_i = LCM(P_1, \ldots, P_m)$, we only need to check c^i_{in} invocations before N_i. Therefore, the $\Delta(c^i_{in})$ is bounded by

$$\begin{aligned}\Delta(c^i_{in}) &= max_{0 \le k \le N_i} \Delta^k(c^i_{in}) \\ &= max_{1 \le k \le N_i}(k \cdot P_i + o_i - min_{1 \le j \le m}(\lfloor \frac{k \cdot P_i}{P_j} \rfloor \cdot P_j + o_j))\end{aligned}$$

Since the c_i runs at the same rate as c^i_{in}, and all communications between c^i_{in} and c_i are synchronous, we can conclude:

$$\begin{aligned}\Delta(c_i) &= \Delta(c^i_{in}) \\ &= max_{0 \le k \le N_i} \Delta_k(c^i_{in}) \\ &= max_{1 \le k \le N_i}(k \cdot P_i + o_i - min_{1 \le j \le m}(\lfloor \frac{k \cdot P_i}{P_j} \rfloor \cdot P_j + o_j))\end{aligned}$$

This proves the given equation defines the bound of $EST(c_i)$ for any components in the transaction.

$\square$

We can now prove the results of Algorithm 4.7 are a sufficient and necessary condition for any feasible schedule.

Theorem 4.8. *A schedule Ψ of a set of components $C = \{c_i\}$ is feasible if only if the start and completion times of component c_i in Ψ, denoted as $s(c_i)$ and $e(c_i)$, falls in c_i allowable execution window $(EST(c_i), LCT(c_i))$, i.e.,*

$$(s(c_i), e(c_i)) \in (EST(c_i), LCT(c_i))$$

Proof. Prove sufficiency. If $(s(c_i), e(c_i)) \in (EST(c_i), LCT(c_i))$ holds for a schedule Ψ, $EST(c_i) \leq s(c_i)$ and $e(c_i) \leq LCT(c_i)$ are satisfied for every component c_i in Ψ. $EST(c_i) \leq s(c_i)$ indicates that c_i's execution in Ψ does not occur earlier than it can be released. For every output component c_o, we can conclude that $e(c_o) \leq LCT(c_o)$. Since $LCT(c_o) = D(T_x)$, $e(c_o) \leq LCT(c_o)$ for every c_o indicates all transactions meet their deadlines. Thus Ψ is feasible.

Prove necessity. We prove by contradiction. If $s(c_i) < EST(c_i)$, c_i starts before $EST(c_i)$ in Ψ. This implies that c_i starts before some of its predecessors completes. This is because c_i starts after the input along the longest sequential path is ready, and the total resource demands of the longest sequential path from an input component to c_i cannot be less than the total of resource demands of all components and links on the path. c_i starting before its predecessor contradicts the fact that Ψ is valid as it alters the component's precedence constraints. Thus, $EST(c_i) \leq s(c_i)$ must be true.

Similarly, if $LCT(c_i) < e(c_i)$, the completion of c_i in Ψ must be after $LCT(c_i)$. According to $LCT(c_i)$ computation in Eq. (4.6), $LCT(c_i) < e(c_i)$ implies that all components on the critical path after c_i to the output component will complete their execution after their LCTs. Then, the output component will also have $LCT(c_o) = D(T_i) < e(c_o)$, indicating the schedule Ψ causes missing the e2e deadline. This contradicts the fact that Ψ is a feasible schedule. Thus, $e(c_i) \leq LCT(c_i)$ must be true. $\square$

Note that the schedule Ψ in Theorem 4.8 can be any schedule, including a local schedule with only a subset of the components in the application (for example, a component on one processor or within one task), and the system schedule with all the components. This implies that the derived component's timing constraints must be met for every component in a feasible schedule, regardless of the interferences among the concurrent components. This allows the components in different partitions to be organized independently and still meet the end-to-end performance constraints. However, Theorem 4.8 only ensures the feasibility of the schedule with regard to the timing constraints, and does not guarantee the validity of the schedule from the perspective of the components' dependencies. For example, given $c_i \prec c_{i+1}$, it is possible for a schedule with $EST(c_i) \leq s(c_i) < e(c_i) \leq LCT(c_i)$, and $EST(c_{i+1}) \leq s(c_{i+1}) < e(c_{i+1}) \leq LCT(c_{i+1})$, but $s(c_{i+1}) \leq s(c_i)$, which violates the component dependencies and results in an invalid schedule. Such a schedule validity is handled by the component sequencing and runtime scheduling algorithm, which will be discussed later in task formation and runtime analysis.

Corollary 4.1. *The timing constraints derived using Algorithm 4.7 form a sufficient and necessary condition for any feasible schedule of a given transaction.*

Proof. This is a direct conclusion of Lemma 4.1, 4.2, 4.3, and Theorem 4.8. $\square$

The complexity of the timing assignment algorithm is linear. For each transaction t with n_t components and l_t links, the **assign_forward()** function visits each component and link once to assign EST and Δ, taking $O(n_t + l_t)$ steps. Similarly, the **assign_backward()** takes $O(n_t + l_t)$ to assign LCT for all components in t. For a system with n components and l links, we have $\sum_t n_t = n$ and $\sum_t l_t = l$. Therefore, the computation complexity of the timing assignment algorithm can be bounded by $O(n + l)$.

4.5 Task Formation

Task formation is a partition step to further divide the partitions in the partition graph generated from the component allocation step. The result is a refined partition graph with a set of smaller partitions to form the tasks as defined in Definition 2.8. This requires identifying the components in each task, and sequencing their executions to meet their timing constraints. The sequencing needs trade-offs between system overheads and schedule flexibility. The system overheads, introduced by the OS for managing tasks, increase along with the number of tasks. These overheads include both computation and storage (memory) resources. Conceptually, all components in the same partition run on the same computation device, thus can be grouped into one task. Thus-formed tasks result in the minimum system overhead. However, grouping all components in the same partition makes component sequencing difficult if not impossible, resulting in an inflexible schedule, because the components in the same partition may run with different timing properties including rates, release offsets, and deadlines. In contrast, it is also possible to run each component as a dedicated task. Such a policy can yield the most flexible schedule because the timing and scheduling parameters of the task can be derived consistently with the timing constraints of the components. However, as the number of components can be large in each partition, the system overheads can be high. In this section, we will therefore present a method to minimize the system overheads by reducing the number of tasks while still keeping the component sequencing easy and flexible, and meeting the timing constraints.

4.5.1 Problem formulation

At the time of task formation, we transform the structural model M_s into a partition graph M_{pn} with the timing constraints of the components assigned. To make this partition graph a valid runtime model, we need to further determine the task structure, i.e., components contained in a task, their execution sequences, and the task performance parameters. The derived task set should yield a *valid schedule* defined as follows:

Definition 4.15. *Given two components c_i and c_j with $c_i \prec c_j$ in a model. A schedule Ψ is called a* valid schedule *if and only if $e(c_i) \leq s(c_j)$ in Ψ.*

A valid schedule maintains the specified dependencies among the components in the structural model. It is essential to maintain the functional correctness of the ECSW after the task formation. Since the semantics of a task requires sequential and run-to-complete execution of all its components, we further restrict the tasks generated by using our method to contain no *internal idle time,* which is defined as follows:

Definition 4.16. *Given a component sequence Ψ of a task, an* internal idle time *of a task, denoted by θ, is a time duration inside a task between the completion of a component $e(a_i)$ and the release of its immediate successor $s(a_{i+1})$ in Ψ, i.e.,*

$$\theta = s(a_{i+1}) - e(a_i)$$

The internal idle time should be eliminated in the task formation. If there exists an internal idle time in a task, $max(\theta(a_i)) > 0$. Otherwise, $\theta(a_i) = 0$. Existence of internal idle time in a task requires a mechanism to control the component release time inside a task, which complicates the implementation and makes the scheduling difficult. The cause of the internal idle time includes strictly sequential and non-preemptable executions of components inside a task and the possible involvement of dependent components that run remotely.

Another requirement of task formation is to minimize the system overheads. The system overheads are associated with the OS service resource consumptions to manage the task executions. We call such overheads *service overheads*. In this work, we consider the timing service overhead and the scheduling service overhead. The timing service overhead is caused by generating and processing timer signals. It depends on the timer resolution and the number of timers used in the system, which consequently depend on the number of tasks in the system. The scheduling service overhead includes both the overhead of selecting a ready task for execution, and the context switches, which also depend on the number of tasks. Therefore, reducing the number of tasks can effectively reduce the system overhead.

With these requirements, our task formation problem can be formally stated as

> For every partition PN_i in the partition graph with the timing constraints of component a_c assigned, $(r_i(a_c), EST_i(a_c), LCT_i(a_c))$, divide PN_i into new groups $\tau = \{\tau_i\}$ that satisfy:
>
> 1. components $a_c \in \tau_i$ execute sequentially in a valid schedule Ψ with their executions satisfying $(r_i(a_c), EST_i(a_c), LCT_i(a_c))$;
> 2. $\tau_i \bigcap \tau_j = \emptyset$;
> 3. τ_i executes continuously with $\theta(a_c) = 0$;
> 4. the total number of groups $|\tau|$ is minimized.

To find a solution to this problem, we need to address two issues: (1) forming the groups in each partition and (2) sequencing the components in each group with their timing constraints met. The resultant groups then form the tasks.

Theorem 4.9. *The task formation problem is NP-hard.*

Proof. The proof is by reduction. The task formation problem contains a subproblem of sequencing the components with arbitrary execution times in a group so that all of their timing constraints of release offsets and completion times are met. Suppose we have determined the groups with their constituent components, determining their execution order with constraints met in problem statement is equivalent to the problem of sequencing with release times and deadlines on one processor [25]. Since the problem of sequencing with release times and deadlines is NP-complete, our task formation problem is NP-hard. $\square$

Since the task formation problem is NP-hard, we need a heuristic algorithm to find a solution that scales to large systems.

4.5.2 Task formation algorithm

Our task formation process works as follows. Given a partition TN_i with components $a_1, \ldots, a_n$, we start with an initial task set of n tasks $\tau = \{\tau_1, \ldots, \tau_n\}$, each of which contains one component, modeled as $\tau_i =< c_i, 1/r(c_i), LCT(c_i), EST(c_i), F(c_i), loc(c_i) >$. These tasks are then merged iteratively to minimize the total number of tasks in the final system. The components are sequenced as the tasks are merged. Their timing constraints derived in the timing assignment are used to verify the validity of results during the sequencing. Two issues must be addressed in the task formation: *selecting tasks for a merge*, and *sequencing the components* in the merged task.

Algorithm 4.10 shows our task formation algorithm.

Selecting tasks for a merge.

The algorithm merges the tasks with single components first according to their invocation rates and belonged transactions. This is based on the fact that the dependent components in a transaction run at the same rate and their executions are continuous. The components in such a group satisfy

$$EST(a_i) < EST(a_j) \Rightarrow LCT(a_i) < LCT(a_j)$$

Therefore, the components in the group can be sequenced according to their ESTs. The ties of EST for concurrent components in the same transaction are resolved by their LCT. The algorithm

Algorithm 4.10 *Task formation algorithm.*

```
INPUT: partition graph $M_{pn}$;
    timing assignment $(r_i, EST_i, LCT_i)$ for every component $a_i$;
OUTPUT: task set $\tau$.
BEGIN
 1  foreach partition $PN_i \in M_{pn}$ do
 2    foreach rate $r_i$ do
 3      foreach $T_j \in \Phi(m_k)$ of $M_s$ do
 4        create an ordered group $\tau_{i,T_j} \leftarrow \emptyset$;
 5        $s(\tau_{i,T_j}) = e(\tau_{i,T_i}) = 0$;
 6        $wcet(\tau_{i,T_j}) = 0$;
 7      end-foreach
 8    end-foreach
 9    sort $a_c$ in $PN_i$ in ascending order of $EST(a_c)$;
10     for $a_c$ with same $EST(a_c)$ do
11       sort $a_c$ in ascending order of $LCT(a_c)$;
12     foreach $a_c \in PN_i$ with rate $r_i$ in transaction $T_j$ do
13       append $a_c$ to $\tau_i i, T_j$;
14       $s(a_c) \leftarrow max(e(\tau_{i,T_j}, EST(a_c))$;
15       $e(a_c) \leftarrow s(a_c) + w_c$;
16       $e(\tau_{i,T_j}) \leftarrow e(a_c)$:
17     end-foreach
18     foreach group $\tau_{i,T_j}$ do
19       foreach $a_i, a_{i+1} \in \tau_{i,T_j}$ with $e(a_c) < s(a_{c+1})$ do
20         $\tau_{i,T_{j,c}} \leftarrow (a_1, ..., a_c)$;
21         $\tau_{i,T_{j,c+1}} \leftarrow (a_{c+1}, ..., a_n)$;
22         update $s(\tau_{i,T_{jc}}), e(\tau_{i,T_{j,c}}), wcet(\tau_{i,T_{j,c}}), s(\tau_{i,T_{j,c+1}}), e(\tau_{i,T_{j,c+1}}), wcet(\tau_{i,T_{j,c+1}})$;
23       end-foreach
24     check groups in ascending order of $s(\tau)$;
25     while exist two group in order $\tau_{i,x}$, $\tau_{i,z}$ with $s(\tau_{i,z}) \leq e(\tau_{i,x})$ do
26       comp_sequence($\tau_{i,x}, \tau_{i,z}$);
27       if returned schedule is valid then
28         $\tau_{i,x} \leftarrow$ returned schedule that merges $\tau_{i,x}, \tau_{i,z}$;
29         update $s(\tau_{i,x}), e(\tau_{i,x}), wcet(\tau_{i,x})$;
30       end-if
31     end-while
32     while exist $\tau_{i,x}$ and $\tau_{j,y}$ with $r_i = \lfloor \frac{r_i}{r_j} \rfloor r_j$ do
33       if $\tau_{i,x}$ and $\tau_{j,y}$ satisfy Eq. (4.12) and (4.13) then
34         duplicate component sequence in $\tau_{i,x}$ $N = \lfloor \frac{r_i}{r_j} \rfloor$ times;
35         create $\tau_{j,x}$ with $N$ instance with $s(\tau^{k+1}_{i,x}) = s(\tau^k_{i,x}) + 1/r_i i$ and $e(\tau^k_{i,x}) = s(\tau^k_{i,x}) + wcet(\tau_{i,x}$;
36         modify $\tau_{j,x}$ invocation rate to $r_j$;
37         comp_sequence($\tau_{j,x}, \tau_{j,y}$);
38         if returned schedule is valid then
39           $\tau_{j,y} \leftarrow$ returned schedule that merges $\tau_{j,x}, \tau_{j,y}$;
40           update $s(\tau_{j,y}), e(\tau_{j,y}), wcet(\tau_{j,y})$;
41         end-if
42       end-if
43     end-while
44   end-foreach;
45   return $\tau$;
END.
```

then determines the start and completion time of each component in the sequence as:

$$s(a_c) = max(e(\tau_i), EST(a_c))$$

$$e(a_c) = s(a_c) + w_c$$

$$e(\tau_i) = e(a_c)$$

where $s(a_c)$ and $e(a_c)$ are the start and completion time of component a_c in the task, and $e(\tau_)$ is the group execution time with up-to-date components. Each group after the first step contains scheduled components from the same transaction with the invocation rate. Since the resources are sufficient for the executions of these components, it is guaranteed the generated schedule of components in each group is feasible, i.e., all their assigned timing constraints are met.

The algorithm then verifies that no $\tau_i \in \tau$ contains internal idle time by comparing a component's completion time $e(a_c)$ and the start time of its immediate successor in the schedule $s(a_{c+1})$. If an internal idle time between a_c and a_{c+1} is found, τ_i is split into two groups with one group $\tau_{i,T_{j,c}}$ containing sequenced components before a_c (including a_C), and the other $\tau_{i,T_{j,c+1}}$ containing sequenced components after $a_c c + 1$. The completion time of $\tau_{i,T_{j,c}}$ is therefore adjusted to be $e(a_c)$, while the start time of $\tau_{i,T_{j,c+1}}$ is set to $s(a_{c+1})$. The process repeats to the group $\tau_{i,T_{j,c+1}}$ recursively until all components are checked.

After the initial groups are constructed, our algorithm chooses two groups to merge at each iteration. The two groups are selected according to the *rate similarity* and *execution overlap*. The *rate similarity* constrains the selected tasks to have harmonic invocation rates. The *execution overlap* requires the selected tasks with overlapped executions. We first find any two groups that satisfy the following conditions for a merge:

1. $r_i = r_j$;

2. $s(\tau_i) \leq s(\tau_j) \leq e(\tau_i)$;

3. any component a_c in the schedule of the new merged group has $EST(a_c) \leq s(a_c)$ and $e(a_c) \leq LCT(a_c)$.

The first condition guarantees the components in the merged group have a one-to-one invocation relationship at runtime. The second condition ensures the groups contains overlapped execution so the components in the merged group can form continuous execution. The last condition ensures that every component in the merged group has its execution within its assigned execution window to meet the end-to-end timing constraints.

After merging the groups with the same rate, the groups with harmonic rates are examined for a merge. The selected tasks τ_i and τ_j must satisfy the following conditions in order to be merged:

1. P_i is a divisor of P_j ($P_j = NP_i$, where N is an integer);

2. $$max(o_i, o_j) - min(o_i, o_j) \leq \begin{cases} w_i & \text{if } o_i \leq o_j; \\ w_j & \text{otherwise.} \end{cases} \tag{4.12}$$

3. $$w_j + (N-1)w_i \geq \begin{cases} (N-1)P_i & \text{if } o_i \leq o_j; \\ (N-1)P_i + o_i - o_j & \text{otherwise.} \end{cases} \tag{4.13}$$

   ($N = \lceil \frac{P_j}{P_i} \rceil$).

where P_i, o_i, w_i and P_j, o_j, w_j are the period, release offset, and computation resource demands of group τ_i and τ_j, respectively. The merge of two tasks meeting the above condition is done by duplicating the short period group τ_s N times and merging them with the longer period group τ_l, considering each instance of τ_i as an independent task with different release offsets.

Component sequencing.

During the merging of two groups in Algorithm 4.10, the components in the two groups need to be sequenced to maintain run-to-complete semantics of the task. Since the problem of sequencing components with all their timing constraints met is NP-hard [25], we use the minimum-EST-first as a heuristic to sequence the components. The minimum-EST-first heuristic ensures that the generated component sequence obeys the component dependency. The algorithm is shown in Algorithm 4.11

Algorithm 4.11 *Component sequencing algorithm:comp_sequence().*

```
INPUT: $\tau_i$ with sequences $\Psi_i =< a_1, \dots, a_n >$ and $\tau_j$ with $\Psi_j =< b_1, \dots, b_m >$;
OUTPUT: new execution sequence $\Psi_{ij}$ on success.
BEGIN
1   initialize the search with start point $p \leftarrow 1$ and end point $q \leftarrow n$;
2   for $j = 1$ to $m$ do
3     find $i$ between $[p, q]$ in $\Psi_i$ with $EST(a_i) < EST(b_j) \leq EST(a_{i+1})$;
4     if no such $i$ then exit-loop;
5     find $k$ between $[i, q]$ in $\Psi_i$ with $s_i(a_k) \leq LCT(b_j) - F(b_j) < s_i(a_{k+1})$;
6     if no such $k$ then exit-loop;
7     find a position $l$ between $[i, k]$ in $\Psi_i$ to insert $b_j$ such that component timing constraints met;
8     if no such $l$ can be found then return $fail$;
9     else
10        insert $b_j$ to $\Psi_i$ before $l$;
11        update $s_j(b_k)$ for $b_{j+1}, \dots, b_m$;
12        update $s_i(a_k)$ for $a_{l+1}, \dots, a_n$;
13        if any $a_{l+1}, \dots, a_n$ with $e_i(a_p) > LCT(a_p)$ or any $b_{j+1}, \dots, b_m$ with $e_j(b_q) > LCT(b_q)$
14            then return $fail$;
15        $p \leftarrow l$;
16    end-if-else
17  end-for
18  if $j < m$ then
19    add the rest components $b_j \in \Psi_j$ to the end of $\Psi_i$;
20    revise $s_{ij}(b_j), e_{ij}(b_j)$ for these components;
21  end-if
22  $\Psi_{ij} \leftarrow \Psi_i$;  23   return $succ$, $\Psi_{ij}$;
END.
```

The algorithm works as follows. Given $\Psi_i =< a_1, \dots, a_n >$ and $\Psi_j =< b_1, \dots, b_m >$ are the component sequences of the two groups τ_i and τ_j with $o_i < o_j$, we insert $b_1, \dots, b_m$ one-by-one to the Ψ_i in their execution order in Ψ_j to form Ψ_{ij}. A eligible position in Ψ_{ij} for b_j should satisfy: (i) $EST(a_i) < EST(b_j) \leq EST(a_{i+1})$, (ii) $max\{e_{ij}(a_i), e_{ij}(b_{j-1})\} \leq s_j(b_j)$, and (iii) not violating any timing constraints of $b_{j+1}, \ldots, b_m$ and $a_{i+1}, \ldots, a_n$ after the insertion of b_j. The start time of $b_{j+1}, \ldots, b_m$ in Ψ_j and $a_{i+1}, \ldots, a_n$ in Ψ_i should then be updated using

$$s_i(a_{i+1}) = e_{ij}(b_j), s_i(a_k) = s_i(a_{k-1}) + F(a_k)(i + 2 \leq k \leq n)$$

$$s_j(b_{j+1}) = e_{ij}(b_j), s_j(b_l) = s_j(b_{l-1}) + F(b_l)(j + 2 \leq l \leq m)$$

In case the components cannot be sequenced with the condition satisfied, the algorithm gives up with the assumption that no non-preemptive feasible schedule exists for the merged group, instead of performing an exhaustive search. This prevents partially merged groups as such a merge does not reduce the number of groups, hence does not reduce the system overhead, while sacrificing the schedulability as it results in a long task execution time.

After each merge, the performance constraints of the merged groups τ_{ij}=(Ψ_{ij},P_{ij},d_{ij},o_{ij},w_{ij}, loc_{ij}) can be determined as follows:

$$P_{ij} = P_j = P_j;$$

$$d_{ij} = \begin{cases} d_i + w_j & \text{if } d_i \leq d_j \\ d_j + w_i & \text{if } d_i > d_j \end{cases}$$

$$o_{ij} = min(o_i, o_j);$$

$$w_{ij} = w_i + w_j;$$

$$loc_{ij} = loc_i = loc_j$$

Upon completion of Algorithm 4.10, each partition in the partition graph M_{pn} has been further divided into a set of groups with the sequentially executing components. Each group can then be implemented as a task whose execution is managed by underlying supporting software on a computation device. The final runtime model can then be constructed with these resultant tasks.

4.5.3 Algorithm analysis

The correctness of the algorithm relies on the rate similarity and execution overlap used for selection of the merging task, and the earliest-EST-first sequencing. These policies ensure the generated groups with the properties defined in the task model. We first prove that the merged task using rate similarity and execution overlap results in no internal idle time.

Lemma 4.4. *Given two groups $\tau_i =< \Psi_i, P_i, d_i, o_i, w_i, loc_i >$ and $\tau_j =< \Psi_j, P_j, d_j, o_j, w_j, loc_j >$ ($P_i \leq P_j$) with continuous component sequences $\Psi_i = \{a_1, a_2, ..., a_m\}$ and $\Psi_j = \{b_1, b_2, \ldots, b_n\}$, merging τ_i and τ_j into τ_{ij} without internal idle time in Ψ_{ij} can happen if and only if*

1. *P_i is a divisor of P_j ($P_j = NP_i$, where N is an integer);*

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

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

   ($N = \lceil \frac{P_j}{P_i} \rceil$).

Proof. We first prove sufficiency. We need to prove that if the two tasks satisfy conditions 1,2, and 3, then $\theta(c_i) = 0$ for all $c_i \in \Psi_{ij}$. Since $P_j = NP_i$, we only need to consider one period of P_j. We denote the start and completion times of component a_k in Ψ_i as $s_i(a_k)$ and $e_i(a_k)$, those of component b_l in Ψ_j as $s_j(b_l)$ and $e_j(b_l)$, and those of component c_i ($c_i = a_k$ or b_l) in merged Ψ_{ij} as $s_{ij}(c_i)$ and $e_{ij}(c_i)$.

First, suppose $o_i \leq o_j$. We have

$$o_j - o_i \leq w_i, o_j = s_j(b_1), o_i = s_i(a_1)$$

$$\Rightarrow s_j(b_1) - s_i(a_1) \leq w_i$$

$$\Rightarrow s_i(a_1) \leq s_j(b_1) \leq s_i(a_1) + w_i = e_i(a_m)$$

This indicates that the execution of Ψ_i and Ψ_j overlaps. According to condition 3, we have

$$w_j + (N-1)w_i \geq (N-1)P_i$$

$$\Rightarrow w_j \geq (N-1)(P_i - w_i)$$

Since the maximum idle time for each P_i period is $P_i - w_i$, the maximum idle time during $P_j - P_i$ is $(N-1)(P_i - w_i)$. So, $w_j \geq (N-1)(P_i - w_i)$ implies the idle time between the completion of the current Ψ_i and the start of next Ψ_i during $P_j - P_i$ can be filled by Ψ_j.

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

So, for any component c_i and c_{i+1} in Ψ_{ij}, if $c_i = a_k, c_{i+1} = a_{k+1}$, then $\theta(c_i) = \theta(a_k) = 0$ since Ψ_i has no internal idle time. Similarly, if $c_i = b_l, c_{i+1} = b_{l+1}$, then $\theta(c_i) = \theta(b_l) = 0$ since Ψ_j has no internal idle time. If $c_i = a_k, c_{i+1} = b_l$, according to conditions 2 and 3, $s_{ij}(c_i) < s_j(b_l) < e_{ij}(c_i)$. This results in $s_{ij}(b_j) = e_{ij}(c_i)$. Thus $\theta(c_i) = 0$. Similarly, we can prove $\theta(c_i) = 0$ for $c_i = b_k, c_{i+1} = a_l$. Therefore, satisfying the three conditions can lead to $\theta(c_i) = 0$ for all components in the merged component sequence Ψ_{ij}.

We now prove necessity. First, we prove $P_j = NP_i$ (N is an integer) is necessary. Suppose P_i is for Ψ_i starting with a_1 and P_j is for Ψ_j starting with b_1 ($o_i \leq o_j$), the release phase between Ψ_i and Ψ_j, $\Delta(b_1)$, can be computed as

$$\Delta^k(b_1) = (kP_j + o_j) - min(kP_j + o_j, \lfloor \frac{kP_j}{P_i} \rfloor P_i + o_i)$$
$$k = 0, 1, 2, \dots, \frac{LCM(P_i, P_j)}{P_j} - 1$$

Suppose a_i in Ψ_i is the immediate predecessor of b_1 in Ψ_{ij}, then $\theta_{ij}(b_1) = s_{ij}(b_1) - e_{ij}(a_i) \sim \Delta^k(b_1)$. In order to make $\theta_{ij}(b_1) = 0$ for any case, $\Delta^k(b_1) = 0$ must be true for any k. It is obvious that $\Delta^k(b_1) = 0$ is true if and only if P_i is a divisor of P_j, i.e., $P_j = NP_i$. The case $o_i > o_j$ can be proved the same way.

For condition 2, let's assume $o_{ij} = o_i$ (a_1 is the first component in Ψ_{ij}). This implies $o_i \leq o_j$. Since $\theta_{ij}(c_i) = 0$ for any c_i in Ψ_{ij}, b_1 must satisfy $\theta_{ij}(b_1) = 0$. This implies that there exists a_i in Ψ_i such that $e_{ij}(a_i) = s_{ij}(b_1)$. Therefore,

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

We can prove that condition 2 is true for the case $o_i > o_j$ in the same way.

For condition 3, let's assume $o_{ij} = o_i$ first. Ψ_{ij} has $\theta(c_i) = 0$ for any component c_i. Given

$P_j = NP_i$, the completion time of Ψ_{ij} can be computed as

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

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

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

□

Lemma 4.4 shows that selection and merging of tasks implemented in Algorithm 4.10 is correct. Now we prove the component sequencing implemented in Algorithm 4.11 is correct in these terms: (1) the original dependencies among components is maintained, (2) its execution is continuous, and (3) no cycle is introduced.

Lemma 4.5. *Given a group τ_{i,T_j} consisting of components from the same transaction T_j running at the same rate r_i, a schedule Ψ generated by totally ordering the components using earliest EST first maintains the components' dependencies in T_j.*

Proof. The proof is straightforward. For component a_i and a_j in the same transaction, the following implication must hold:

$$a_i \prec a_j \Leftrightarrow EST(a_i) < EST(a_j)$$

according to the timing assignment algorithm. For a group τ_{i,T_j} containing a_i and a_j, we have $a_i \prec a_j \in \Psi$ since they are ordered according to EST. Thus, we can conclude

$$a_i \prec a_j \in T_j \Leftrightarrow a_i \prec a_j \in \Psi$$

$\square$

Lemma 4.6. *Given two tasks, τ_i and τ_j, with the same rate,*

1. *if $a_i \prec a_j$ in the schedule Ψ_x (or Ψ_y) of τ_i (or τ_j), then $a_i \prec a_j$ in the schedule Ψ_{ij} of the merged group τ_{ij};*

2. *if both Ψ_i and Ψ_j are with $\theta_i(a_i) = 0$ and $\theta_j(b_j) = 0$, and $o_i < o_j < o_i + w_i$ or $o_j < o_i < o_j + w_j$, the merged sequence Ψ_{ij} has $\theta_{ij}(c_i) = 0$ for every c_i.*

Proof. We prove these assertions by contradiction. To prove the first assertion, assume $a_i \prec a_j$ in one schedule before the merge, and $a_j \prec a_i$ in the schedule Ψ_{ij} after the merge. If a_i and a_j are in τ_i, it is obvious that $a_j \prec a_i$ in Ψ_{ij} must be the result of $a_j \prec a_i$ in Ψ_i since the relative positions of all components in τ_i never change during the merge. If a_i and a_j are in τ_j, $a_j \prec a_i$ in Ψ_{ij} implies $a_j \prec a_i$ in Ψ_j. This is because of the fact that the components in τ_j must be inserted to τ_i in the order of Ψ_j during the merge. Component a_j is never inserted before a_i if $a_i \prec a_j$ in Ψ_j. Therefore, $a_j \prec a_i$ in Ψ_{ij} contradicts $a_i \prec a_j$ in the schedule of a partition before the merge. So $a_i \prec a_j$ in Ψ_{ij} must be true for the merged groups if $a_i \prec a_j$ exists in any original group.

To prove the second assertion, assume there exists a c_i in Ψ_{ij} such that $\theta_{ij}(c_i) > 0$ in Ψ_{ij} of the merged task τ_{ij}, then

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

We examine two following cases:

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

case 2. c_i and c_{i+1} are from different tasks. Without losing generality, assume $c_i \in \Psi_i$ and $c_{i+1} \in \Psi_j$. According to the algorithm, c_i and c_{i+1} are next to each other in Ψ_{ij} after the merge only if there exists $< c_i, c_j > \in \Psi_i$ between which c_{i+1} is inserted. Suppose Ψ_{ij}^k and Ψ_{ij}^{k+1} are the sequences before and after inserting c_{i+1}, and there are k components in Ψ_j that have been merged with Ψ_i.

$$\Psi_{ij}^k = c_1, \ldots, c_i, c_j, \ldots, c_{n+k}$$

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

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

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

Ψ_{ij}^0 is the sequence of Ψ_i without any component in Ψ_j inserted. So $\Psi_{ij}^0 = \Psi_i$. According to the above derivation, we must have $\theta_i(c_i) = s_i(c_j) - e_i(c_i) > 0$. This contradicts the fact that Ψ_i is continuous.

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

Lemma 4.6 gives the conditions for a correct task formation. The first assertion ensures the new merged group maintains the components' dependencies in the original group, which results in the functional equivalence between the transformed group and its origins. The second assertion ensures the merging process does not introduce any internal idle time to the execution, thus minimizing the system overhead without sacrificing the end-to-end performance.

Theorem 4.10. *The component sequence Ψ of a group is a valid schedule if and only if the components in Ψ_i are sequenced using minimum-EST-first.*

Proof. Prove sufficiency. If the components in a group are sequenced using minimum-EST-first, according to Lemma 4.5 and the first assertion of Lemma 4.6, the components' dependencies are maintained in the schedule Ψ_i. Therefore, the component schedule of the group is valid.

Prove necessity. If Ψ_i is a valid component schedule of a group, given $c_i \prec c_j$ in the original model, there must be $e(c_i) \leq s(c_j)$. This implies $s(c_i) < s(c_j)$, which is consistent with $EST(c_i) < EST(c_j)$. Therefore, it is the same as being scheduled using minimum-EST-first. □

Theorem 4.10 implies that the component sequencing algorithm results in a valid schedule for each group. Note such validity is limited to a group. In other words, the schedule only guarantees that the dependencies of the components within a group are maintained, but not the dependencies among the components in different groups.

Corollary 4.2. *The component sequence generated in task formation is both valid and feasible for each group.*

Proof. This is a direct conclusion according to Theorem 4.8 and 4.10. □

Lemma 4.7. *There is no dependency cycle among the components in the schedule Ψ_{ij} of the merged group τ_{ij} if Ψ_i of τ_i and Ψ_j of τ_j contain no cycle.*

Proof. We prove this by contradiction. Suppose there exists a dependency cycle between a_i and a_j Ψ_{ij}, then $a_i \prec a_j$ and $a_j \prec a_i$. If $a_i, a_j \in \tau_i$ (or $a_i, a_j \in \tau_j$), it implies the Ψ_i (or Ψ_j) contains a cycle. This contradicts no cycles in Ψ_j (or Ψ_j). Therefore, a cycle in Ψ_{ij} can only be formed while merging τ_i and τ_j. Without losing generality, we assume $a_i \in \tau_i$ and $a_j \in \tau_j$, and assume $s(\tau_i) < s(\tau_j)$. Given both $a_i \prec a_j$ and $a_j \prec a_i$, there must be at least two instances of either a_j or a_i in Ψ_{ij}. This contradicts any component being in one and only one partition. Therefore, no cycle can be generated during the merging of two initial groups. □

Theorem 4.11. *The groups generated by Algorithm 4.10 meet all properties of tasks.*

Proof. This is a direct conclusion of Corollary 4.2, Lemma 4.4, 4.6, and 4.7. □

Theorem 4.11 shows that the partitions generated by Algorithm 4.10 are tasks in which end-to-end performance constraints can be met if those constraints for the tasks and components can be met.

Algorithm 4.10 is a polynomial time algorithm. Creation of initial groups with components in the same transaction and running at the same rate takes linear time $O(n)$ given n components in M_s. Sorting the components in each group and determining their initial schedule based on earliest EST take $O(n + nlogn)$ time. Given partition PN_i is divided into k groups, it takes $O(klogk)$ to sort the groups and $O(k)$ time to find and merge the groups. Given each group contains u components, each merging takes $O(u + logu)$ to find a position to insert a component into the schedule, and $O(u)$ to verify and adjust the start and end time for the rest of the components. Therefore, the total time to merge the two groups takes $O(u^2)$. Merging all the groups takes $O(ku^2)$ time. As the groups are merged, the number of groups k decreases, while the number of components in each group u increases. Since $\sum_{i=1}^{m} u_i = n_i$ where n_i is the number of components in PN_i, the time taken for merging groups can be bounded by $O(n_i^2)$. Suppose there are p devices, since PN_i has a one-to-one relationship with $PROC_i$ in the platform model, we have $\sum_{i=1}^{p} n_i = n$. As $\sum_{i=1}^{p} n_i^2 < (\sum_{i=1}^{p} n_i)^2 = n^2$, the total time for merging all partitions is bound by $O(n^3)$. The computation complexity of Algorithm 4.10 is therefore $O(n^3)$.

4.6 Task Graph Generation

The task graph generation is the final step of our model transformation. After the task formation, the partition graph is refined with each partition consisting of tasks, which are component groups. The task graph generation then determines the communication and dependencies among the formed tasks, assigning the resource demands and timing attributes for both the tasks and their communications. The resultant model, in the form of a task graph, provides detailed information for implementing the software on a target platform and supports formal timing and scheduling analysis using real-time theories.

4.6.1 Task graph generation algorithm

The task graph generation algorithm constructs the graph by tracing the links in the transactions and aggregating them to form task communications in the task graph. The transactions in a structural model is therefore modeled as task chains in the task graph. The functions of the timing constraints and locations of the tasks and communications are assigned after the task graph is generated according to the component sequence and the timing constraints of the components in each task. The algorithm is shown in Algorithm 4.12.

The algorithm first constructs the task set using the groups obtained from the task formation, and assigns the task attributes accordingly. The invocation period, release offset, workload and execution location of τ_i are assigned as the corresponding invocation period, start time, total workload of all its components, and execution location of the group. The task communications and dependencies can be derived according to the communications and dependencies among the components in different tasks. Given a link $a_i \rightarrow a_j$ exists in the original transaction, and a_i, a_j belong to different

Algorithm 4.12 *Task graph generation.*

```
Input: the models $M_s$ for transactions;
    the partition set $\tau$ obtained from task formation.
Output: a task graph $TG =< \tau, L, P, o, w, loc >$.
BEGIN
 1  foreach component group $\tau_i \in \tau$ do
 2    create a task for each $\tau_i$;
 3    $P(\tau_i) \leftarrow 1/r_i$;
 4    $w(\tau_i) \leftarrow \sum_{a_i \in \tau_i} F_c(a_i)$;
 5    $loc(\tau_i) \leftarrow PROC_i$;
 6    $o(\tau_i) \leftarrow s(\tau_i)$;
 7    $D(\tau_i) \leftarrow d(\tau_i)$;
 8  end-foreach
 9  foreach transaction $T_i$ do
10    foreach link $l(a_i, a_j)\ inT_i$ do
11      if $a_i \in \tau_i, a_j \in \tau_j, i \neq j$ then
12        add directed link $\tau_i \rightarrow \tau_j$ to $L$;
13        adjust link weight $w_l(\tau_i, \tau_j) \leftarrow w_l(\tau_i, \tau_j) + F_l(a_i, a_j)$;
14      end-if
15    end-foreach
16  end-foreach
17  return $TG$;
END.
```

tasks τ_i and τ_j after the task formation, a directed link $\tau_i \rightarrow \tau_j$ is added to the task graph link set. Note there can be multiple pairs of communicating components between two tasks. These links are aggregated to one directed link in the task graph. The weight of the link is therefore the sum of the resource demands of all communicated components in the two tasks. The thus-formed task graph might contain cycles between some tasks. We can prove that such cycles are neither deadlocks nor livelocks, hence will not affect the application of schedulability analysis. The resultant task graph is a valid runtime model. The performance analysis of such a runtime model can be done using the schedulability methods. Details of the runtime model analysis method will be presented in Chapter 5.

4.6.2 Algorithm analysis

Since we merge the components from different transactions and with harmonic periods, the constructed task graph may contain cyclic links among tasks. Such cycles among tasks are resolved at the component level, as shown in the following theorem.

Theorem 4.12. *A task graph generated by Algorithm 4.12 does not contain any cycle among the components.*

Proof. We prove this by contradiction. According to Lemma 4.7, there is no component cycle within a task. We only need to prove that there is no component cycle between the components in different tasks. Suppose there exists a component cycle between components in different tasks:

$$a_i \in \tau_i, a_j \in \tau_j, a_i \prec a_j \land a_j \prec a_i$$

If the above equation is true, then

$$e(a_i) < s(a_j) < e(a_j) < s(a_i)$$

which must be false for any a_i, a_j. Thus there is no component cycle between tasks. □

Theorem 4.12 indicates that the task graph maintains the acyclic properties of the component model at a level with finer granularity. In other words, the task graph at the component level is an acyclic graph. This property satisfies the requirements of all existing schedulability and timing analysis methods. Therefore any existing effective schedulability and timing analysis algorithm can be used for analysis of the task graph.

Theorem 4.13. *A task graph generated using Algorithm 4.12 is a runtime model.*

Proof. The proof is straightforward. According to Definition 4.14, the resultant model is a partition graph. According to Theorem 4.4, the resultant model is a valid runtime model. □

The complexity of Algorithm 4.12 is linear. Given t tasks in the system, the algorithm takes $O(t)$ time to assign the properties to each task. To create the links among tasks, the algorithm needs to examine every link in the structural model, which takes $O(l)$ with l links in the structural model. Given a structural model with n components and l links, there will be at most n tasks (one component in each task). Hence, the computation complexity of Algorithm 4.12 is bounded by $O(n + l)$.

4.7 Evaluations

Our goal of model transformation is to provide a scalable method to generate a runtime model given an ECSW structural model and platform with both performance and resource constraints met. Hence, our evaluations focus on the method's scalability and effectiveness. We are only interested in the algorithms themselves, and do not evaluate the quality or optimality of the generated results as the metrics for these properties are difficult to define. On the other hand, according to the assumptions of our model transformation method, any generated system is assumed to be equivalent to the optimal one, from an end-to-end control perspective, if it meets all timing and resource constraints.

To justify the algorithms in our runtime model generation method, we need to choose a baseline method for comparisons. Unfortunately, no method exists to automatically generate a runtime model from the given structural and platform models in both current research and software tools. All methods for the runtime generation in current practices depend heavily on manual error-and-trial processes. To this end, we divided our method into two consecutive parts for evaluations. The first consists of the rate assignment and component allocation, and the second consists of timing assignment, task formation, and task graph generation. The scalability and effectiveness depends mainly on the component allocation for the first part, and on the task formation for the second part.

So our evaluations are based on the performance of these two algorithms.

The metrics in the scalability evaluation include the number of nodes visited in the design space for the component allocation, and the number of steps performed to generate the results for the task formation. The number of nodes visited was obtained by counting every explored allocation the algorithm tried, regardless of success or fail. The number of steps of task formation was obtained by counting every component merge the algorithm tried, regardless of success or fail. A better algorithm would explore fewer nodes in the design space or perform fewer steps to find a solution. The metrics in the effectiveness evaluation included the failure ratios for the component allocation and the number of tasks generated for the task formation. The failure ratio was computed as the percentage of the number of experiments in which the algorithm fails to find a solution over the total number of experiments conducted. The number of tasks was obtained directly from the result of the algorithm. A better algorithm would cause a lower failure ratio and generate a fewer number of tasks.

4.7.1 Experiment setup

All evaluations were conducted on a desktop machine with an Intel P3-based 500 MHz processor and 256 MB memory. To ensure all experiments are completed within a reasonable time, we limit the algorithm's exploration to be one million nodes. This is equivalent to about 30 minutes of wall-clock execution time on the desktop machine.

The experiments used a set of randomly-generated models. Using randomly-generated models is essential as it is very difficult to manually construct software models with a large number of components. We implemented a graph-generation tool to automatically generate both software and platform models with user-defined characteristics, including the number of components, the node degree of components, weight distributions for nodes and links, and the number of computation devices with their resource availability. The parameters used to randomly generate the models for the experiments are given in Table 4.1.

parameter	value
component graph	
component	$100 \sim 1000$
in/out links	$1 \sim 5$
resource demands	
computation/component	$0.005 \sim 0.05$
memory/component	$0.005 \sim 0.05$
communication/link	$10 \sim 100$
platform model	
device	$5 \sim 50$
link	1 shared
resource bound	
computation	$0.6 \sim 1$
link	$100 \sim 10000$
memory	$0.3 \sim 1$

Table 4.1: Parameters for random system model generation.

In each generated component graph, the number of components was selected from $100 \sim 1000$ in increments of 100. Similarly, the platform models were also generated randomly with the number of computation devices from $5 \sim 50$ in increments of 5.

To make fair comparisons, the algorithms for each evaluation took identical inputs. For component allocation, both the developed algorithm and the baseline algorithm took the same randomly generated graph. For task formation, both the developed algorithm and the baseline algorithm took the same partition graph and timing assignments generated by our component allocation algorithm.

4.7.2 Evaluation results of component allocation

In the evaluation of component allocation, the standard branch-and-bound (BB) algorithm was chosen as a baseline in the experiments since the algorithm has been widely used for component allocation, and its performance has been well studied and understood. In each experiment, the component graph was partitioned and allocated to the processors in the platform using the standard BB, informed BB with component ordering (IBB+O), and informed BB with both ordering and forward checking (IBB+O+FC). We chose the load-balancing strategy for $C_{ideal}(PN_i)$ and $M_{ideal}(PN_i)$.

Performance with different model sizes

We first evaluated the scalability of the algorithm while varying the size of the structural model. The number of components in the model is used to represent the size of a structural model. In the experiments, we fixed the number of processors in the platform model to be five. Since the comparison of the visited nodes under different algorithms is meaningful only when a solution exists, we reduced the resource consumption by each component as the model size increases, to ensure that the software can be fit in the five-processor platform. The generation was tuned so not every allocation meets all resource constraints.

Figure 4.2 shows the number of nodes visited in the design space before a solution was found. The algorithm IBB+O+FC is found to outperform IBB+O slightly, while both outperform the standard BB algorithm significantly. The figure shows that the BB algorithm has been explored an average of two nodes in a design space to find a proper allocation of each component, while IBB+O+FC has been explored exactly one node in all experiments. This implies that IBB+O+FC has chosen a proper allocation at its first trial, and hence, does not need to backtrack. Since IBB+O and IBB+O+FC have both performed significantly better than BB but with minimal differences between them, one can attribute the competence function to this. The small difference between IBB+O and IBB+O+FC is due to the forward checking, which eliminates constraint-violating partitions from the candidate list. Since the platform contains a small number of processors, the design choices eliminated by the forward checking have only limited effects on the overall search space, and result in a small difference between the two algorithms. Moreover, the difference between IBB+O+FC and IBB+O decreases as the size of the structural model increases. The reason for this can be the reduced resource consumption by each component that we set during the graph generation. As the ratio of component resource consumption to the platform resource availability decreases, the number of solutions may increase, so it is more likely to take fewer steps in the design space to find a solution.

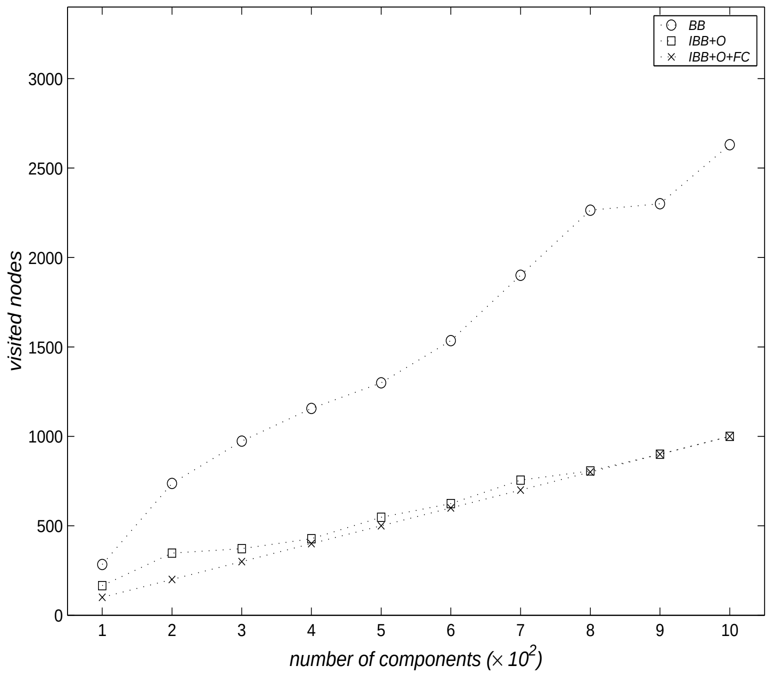


Figure 4.2: The number of nodes visited for different model sizes.

Figure 4.3 shows different algorithms' failure ratios for different structural model sizes. We changed the resource consumption parameters during system generation to make it more likely to violate constraints.

Figure 4.3 shows that BB fails to find a solution in more cases than IBB+O. IBB+O is also slightly more likely to fail to find a solution than IBB+O+FC. These results indicate that IBB+O+FC can find a solution faster than IBB+O, which finds a solution faster than BB. Since we applied all three algorithms to the same structural model in each experiment, the failure ratio with an unscalable algorithm may encounter more false positives, meaning that a solution exists but the algorithm cannot find it within a pre-defined number of steps. We also observed that the difference between failure ratios of different algorithms increases as the number of components increases. This may

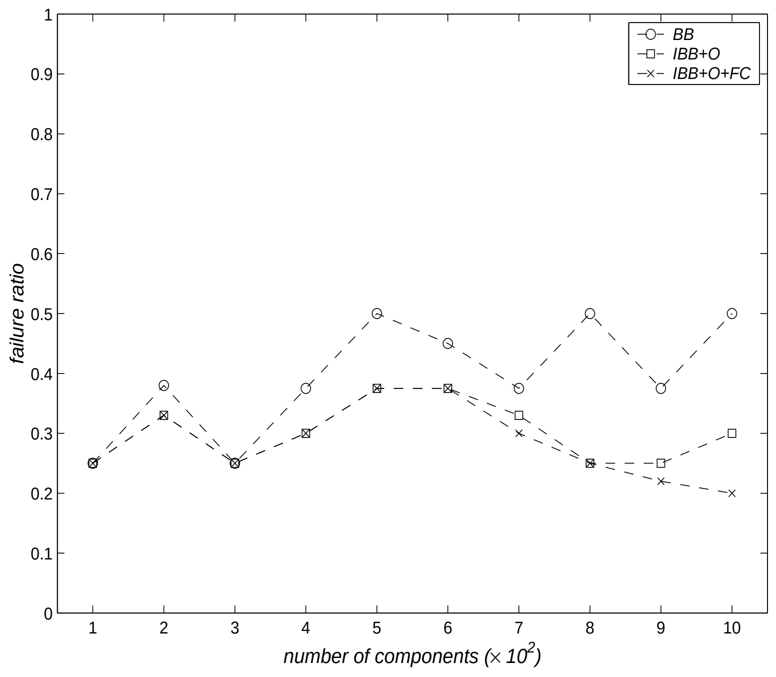


Figure 4.3: The failure ratios of different algorithms.

result from exploring only a fixed number of design nodes in an expanding design space. The swings of the failure ratios for different graph sizes may also result from the randomness of the resource consumptions generated for each experiment.

Performance with different partition numbers

The scalability of our algorithm is also affected by the number of devices in the platform. Since the devices and the component partitions are one-to-one mappings, we used the number of partitions to represent the number of devices in the platform. The principles of the experiment design are the same as those used for the experiments of different model sizes. In this case, we fixed the number of components in the structural model to be 100. Similarly, we set different graph-generation parameters for both visited-node and failure-ratio experiments so that a solution can be found before reaching the node-exploration limit for the former case and unlikely to be found within the exploration limit for the latter. The results of these experiments are plotted in Figures 4.4 and 4.5

The results shown in Figures 4.4 and 4.5 are similar to those for different model sizes. IBB+O+FC

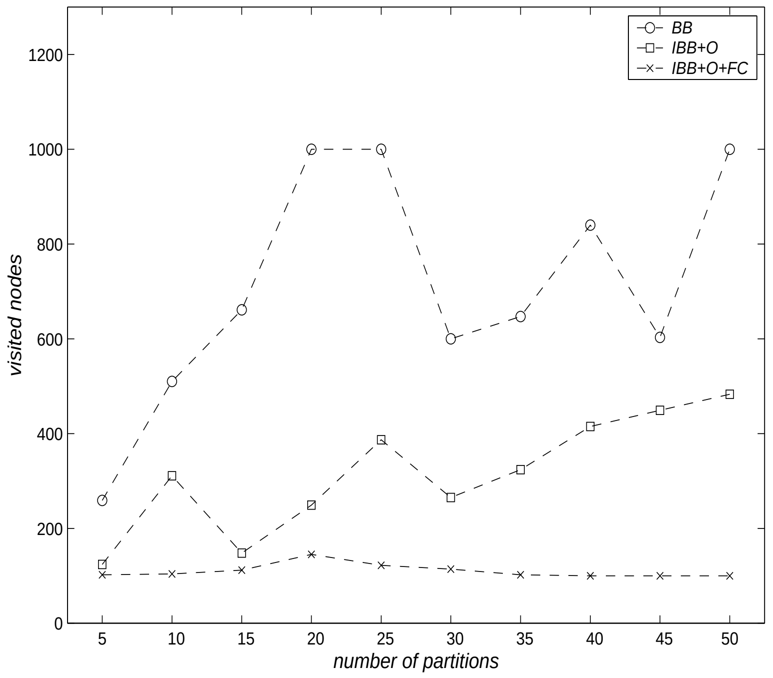


Figure 4.4: The number of nodes visited with a different number of devices.

explored the fewest number of nodes to find a solution, while BB explored the most. There was a significant difference between the visited nodes of IBB+O and IBB+O+FC. In this case, forward checking improved the solution search by removing partitions from a component's domain as the number of partitions increases. Similarly, BB experienced the highest failure ratio, and IBB+O+FC experienced the lowest. The results also show that different failure ratios appear only beyond a certain number of partitions (10 for BB, and 25 for IBB+O). This implies that the competence function and forward checking become helpful only when there are a large number of partition domains.

4.7.3 Evaluation results of task formation

In the evaluation of task formation, the baseline algorithm was selected to group tasks based on the synchronous dependencies, invocation rate, and execution overlap. With the baseline algorithm, only components allocated in the same partition (on the same device), belonging to the same transaction, with the same invocation rate, and immediately next to each other in the transaction were grouped. All concurrent paths were grouped in different tasks to avoid the component sequencing

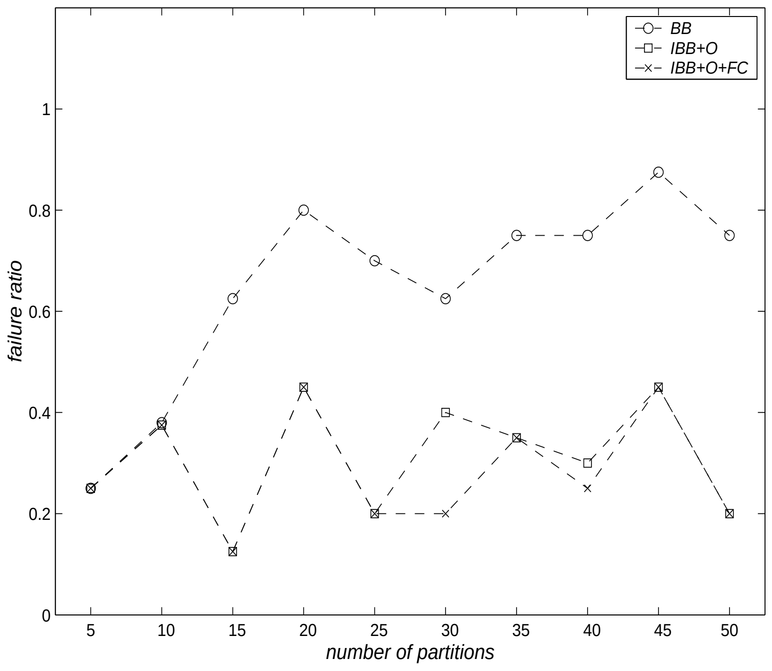


Figure 4.5: The failure ratios of different algorithms.

in a task. Such a method has been commonly used in practices, and is a key assumption for many timing analysis and model transformation algorithms [70, 69, 26, 40, 11]. The input models used in this evaluation were obtained from the above component allocation with the timing assignment algorithm applied. Therefore, the input models contained both execution location information and timing constraints.

In the task formation experiments, we fixed the number of partitions (devices in the platform) to five to make the algorithm complete in a short amount of time. Figure 4.6 shows the number of steps for the task formation of our algorithm (TFA) and the baseline algorithm (baseline). We also plotted the n^3 with n being the number of components to show the trend of the algorithm. To make the result fit in the diagram, we took $ln(s)$ for all results in the diagram presentation.

As can be seen from Figure 4.6, the baseline algorithm took the minimum steps for each case. The results show it is linear, which is consistent with its implementation. For each case, the baseline algorithm simply takes a component, either merges it with an existing group of the same rate, same

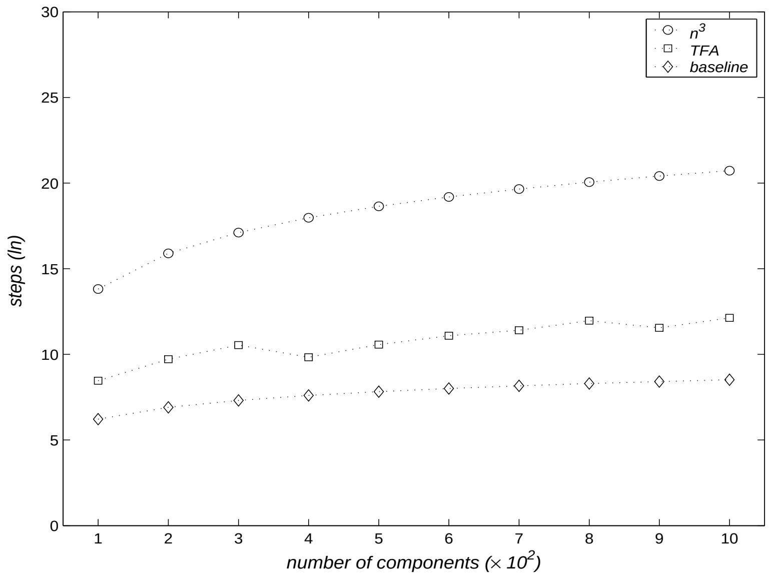
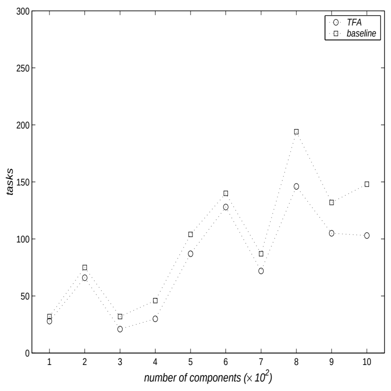


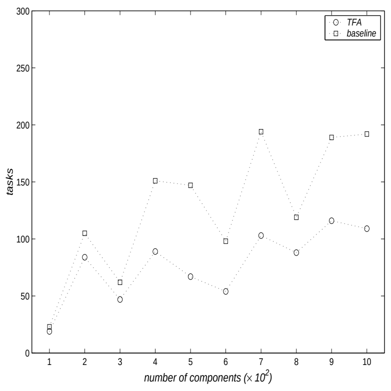
Figure 4.6: Number of steps taken to form tasks.

transaction, and overlapped execution, or creates a new group for it. Therefore, the total number of steps should be linear to the number of components. Our algorithm TFA shows a trend of some high power function. However, the number of steps taken by TFA in this experiment is much smaller than the bound of n^3. This might result from the randomly-generated components' resource demands yielding fewer groups with overlapping executions. Thus less component sequencing is required, resulting in fewer steps. In an extreme case in which very few components can be grouped, the TFA algorithm becomes a linear algorithm too. On the other hand, if most of the components can be grouped, the TFA algorithm may take the number of steps close to n^3 or higher. In any case, the algorithm demonstrated a non-exponential complexity in the experiments, although it was not as scalable as the baseline algorithm.

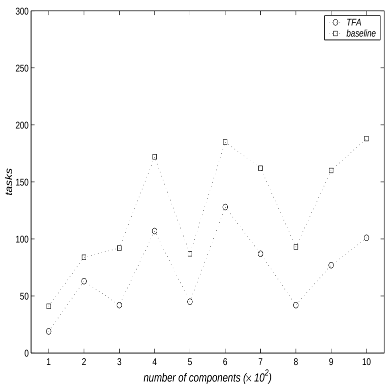
We further conducted the experiments to investigate the quality of the task formation algorithm in terms of total number of tasks generated. As the baseline algorithm generates a task with components with the same rate, transaction, and overlapping executions, the number of tasks generated by the baseline algorithm varies along with the number of partitions, invocation rates, and transactions.



(a) Light workload.



(b) Medium workload.



(c) Heavy workload.

Figure 4.7: Number of resultant tasks under different configurations.

To simplify the experiments, we fixed the number of partitions to five the number of invocation rates to five too. The number of transactions, on the other hand, was randomly generated between $5 \sim 10$, which allowed for multiple transactions running at the same rate. Further, since the TFA algorithm behaves differently with different components' resource demands, we altered the system utilization constraints so the components' resource demands can cause different execution overlaps. Specifically, we constrained the utilization to 0.2 as light workload, which possibly results in fewer execution overlaps of the components, 0.5 as medium workload, which possibly causes more execution overlaps, and 0.7 as heavy workload, which possibly causes execution overlaps for most of the components. The results are shown in Figure 4.7.

As can be seen from Figure 4.7, the number of tasks generated by the TFA was smaller than that generated by the baseline in every case. As the workload increased, the differences between the tasks numbers generated by the TFA and baseline algorithm increased. This indicates that the TFA is more effective on reducing system overhead when the system tends to be busy. In such a case, more components' executions overlap, therefore can be merged. Although the resultant task numbers were close for a light workload, there were still differences because the TFA at least can merge components from different transactions, even their invocation rates were the same. This resulted in TFA generating fewer tasks for any cases in the experiments. This indicates our TFA algorithm is more effective than the baseline algorithm in terms of reducing resource consumptions of the system. The large inconsistent variances between different numbers of components were probably caused by the randomness in the model generation, as different experiments were based on different models.

Based on the above evaluations, we can conclude that our algorithms used in the runtime model generation are both scalable and effective. This is justified by (1) our allocation algorithm outperforming the traditional branch-and-bound algorithm on both the visited design space and failure ratios, and (2) our task formation algorithm producing fewer tasks than the commonly-used task formation method while maintaining the scalability. Besides the performance gained with each algorithm at each transformation step, the most significant contribution of our approach is that the transformation can now be a fully automated process, and hence, it can both eliminate the errors and model conversions introduced at different steps and it can accelerate the design.

4.8 Related Work

Several solutions based on the object-oriented modeling paradigm have been developed to address the task construction problem in support of model-based real-time embedded application software design. One of these is based on HRT-HOOD model [13, 16]. HRT-HOOD is an object-based modeling paradigm. The task construction with the HRT-HOOD model is done through transaction specification and timing assignments. Since the final implementation uses Ada programming language, many model restrictions were imposed to suite Ada-specific features. Saksena, Karvelas and Wang [69] developed a method for automatic task construction based on the software design models in ROOM [74], which is an object-oriented real-time modeling language. This task construction method uses a block waiting thread architecture and models the computation as event streams with each event triggering an object action. During the task construction, the priority of each event is determined according to the schedulability analysis. Then, the actions of the components are assigned to the threads using a branch-and-bound technique. The method is simplified in implementation by imposing architectural restrictions, such as mapping all events for the same component (or in the same transaction, or with the same priority) to the same thread. A similar scenario-based task construction approach was developed in [44]. In this method, the thread is constructed based on capsule instances. However, it is up to the designer to determine how many threads will be in the system, and which capsule instances are merged with this method. Both of the above methods need to deal with mutually exclusive data access in a shared object, which is difficult to automatically detect. There are some task construction methods based on process-oriented component models, such as timed multitasking system [58], Matlab/RTW, and ETAS ASCET. The tasks in timed multitasking systems can be derived automatically but require all tasks deliver output only at the end of the timing frames. The task constructions in RTW and ASCET are also based on the manual process.

Other research tools, such as VEST (Virginia Embedded Systems Toolkit) [79], Cadena [30], MetaH [10], and Time Weaver [18], have also been developed to assist embedded and real-time software development and analysis. However, the model transformation still depends on the manual process without systematic method support.

Our approach, on the other hand, uses a process-oriented model, and constructs tasks by assigning the timing constraints and grouping components. The process-oriented model naturally fits many modeling environments for control design, including Simulink/Stateflow, ETAS ASCET, Teja, and Synchronous Dataflow Diagram. Our timing assignment is different from traditional timing derivation [26] and deadline distribution [38] in that our derived constraints can be used to form a sufficient necessary condition for a valid feasible schedule of a group of components, and thus, can be used to construct the tasks in a runtime model. Our task construction can also be fully automated with the number of final tasks minimized, thus fitting in a resource-limited target.

4.9 Summary

Generating the runtime model from a given structural model and a platform model is a key step in the ECSW development. Both the timing and schedulability analysis results and the final system implementation depend on such a runtime model. In this chapter, we presented a method to support performance-aware runtime model generation of ECSW. The generation is achieved through a multi-step process of partitions and assignments. In this process, the invocation rates specified for inputs of transactions are first propagated to the components along the synchronous links. Then the components in a structural model are partitioned into groups according to computation devices in the platform model. The allocation considers multiple resource constraints and uses the branch-and-bound techniques with forward checking and component ordering to accelerate the allocation algorithm. The resultant partition graph meets the resource constraints of the platform. The process then distributes the end-to-end timing constraints over the components as components' constraints. These components' timing constraints form a sufficient necessary condition for any feasible schedule, and are used to build a valid component sequence in the task formation. The task formation step takes the partition graph with the components' timing constraints assigned, refining each partition with a set of component groups. The groups are formed using the rate similarity and the execution overlap so as to minimize the total number of partitions and avoid internal idle times. The components in each group are sequenced according to their ESTs. Finally, the task graph is constructed in which the timing attributes for the tasks are derived. Our preliminary evaluation results based on a set of randomly generated structural and platform models have shown the scalability and effectiveness of our runtime model generation method.

CHAPTER 5
Performance Analysis with Runtime Models

The runtime model generated in Chapter 4 contains a set of communicating tasks with their performance constraints and characteristics assigned. These constraints and characteristics include invocation periods, release offsets, relative deadlines, and worst-case resource demands. These tasks and communications have also been assigned to physical computation devices and communication links in the platform model. Before the tasks are coded and deployed on the modeled target, it is essential to analyze the ECSW performance using the runtime model to ensure the implementation meets all the system-level performance and resource constraints of the original design. In this chapter, we present the methods for such an analysis using our generated models and existing real-time schedulability analysis techniques. Different from the performance analysis of the structural model, which deals only with the software and analyzes the design based on estimations, the runtime model performance analysis considers the effects of both the software and its execution environment.

The runtime model performance analysis is necessary for the following reasons. First, the generated runtime model only specifies the task constraints for meeting the system-level end-to-end performance constraints. However, such task constraints do not directly indicate the schedule of these tasks to meet these constraints, and consequently to meet the end-to-end constraints. During the runtime model generation, we schedule only the components within each task. Such a schedule is a partial schedule at the system level, meaning that it only specifies the execution order in each task. As the ECSW contains multiple concurrent tasks, the components in different tasks need to be scheduled to meet the end-to-end system performance constraints. To keep system flexibility, the task schedule is usually left for the OS scheduler to determine dynamically at runtime. The runtime model analysis is therefore needed to ensure that the selected scheduling policies and parameters will yield a system-level valid feasible schedule. Second, the execution environment introduces variances to the task execution, which are ignored during the runtime model generation. One type of variance comes from the interference of other concurrent activities, such as concurrent components in different transactions and tasks. These activities may introduce the system overhead for managing the executions, such as context switch overhead for preemptions and scheduling overhead for dynamic scheduling. Therefore, meeting the tasks' constraints considering only tasks' workloads may not naturally result in meeting the system-level performance constraints with given resources. Another type of variance is from the invocation services, such as timer signal processing and event processing. These services may introduce both resource consumptions and jitter into the task executions. With different types and configurations of these underlying system services, the performance of the runtime model can be dramatically different. Finally, we can achieve better system performance through system-level performance tuning, which is only possible with runtime model analysis. Such tuning can be done by selecting suitable scheduling and invocation mechanisms. This is particularly important for automatic design refinement when some performance constraints are violated.

Our runtime model performance analysis is based on the timing and schedulability analysis theories for real-time systems, which are specific to the scheduling algorithms. Using a different scheduling algorithm for the execution of the same runtime model may result in a different performance. To our knowledge, the timing information specified in our runtime model is sufficient to generate a schedule for any existing analysis algorithm. For our analysis in this work, we have adopted the fixed priority preemptive scheduling policy. This scheduling policy has been extensively studied in the real-time research community and is implemented as a standard scheduling mechanism in most commercial real-time and embedded operating systems. We focus on applying and modifying existing methods for our runtime model performance analysis instead of developing a new analysis algorithm specifically for our runtime model. This will extend the usage of our runtime model and relative modeling techniques to systems using various types of scheduling algorithms. As a result, an analysis using other scheduling algorithms can be done in a similar way.

In this work, we assume that a unique fixed priority is assigned to each task. The tasks are scheduled according to their priorities at runtime and execute in a run-to-complete manner, meaning that a new invocation of a tasks will not start its execution before the current invocation is completed. This implies that there exists at most one instance of a task at any given time. Further, the components in a task inherit the task's priority, i.e., all components in a task share the same priority of the task. Since the priority of a task is statically assigned, the priorities of the components will not change during the execution. For simplicity, the mutually exclusive resource accesses are not considered in this chapter, although they can be easily addressed combining this method with synchronization techniques such as priority ceiling protocols. All tasks in the system are assumed using the same scheduling policies. With these assumptions, we can treat the tasks as building blocks and ignore their constituent components in the performance analysis. Finally, the platform is assumed to use an event-triggered mechanism for the services including timing, scheduling, and IPC. This allows the inclusion of a non-zero delay and variation in OS service performance characteristics in an analysis. An OS with time-triggered services can be considered as a special case of event-triggered services with a zero service delay/variation.

The specific performance analysis discussed in this chapter includes the schedulability of the task set on each device, the end-to-end response times of the task chains, and the resource consumptions of devices and communication links. In particular, our performance analysis of a runtime model addresses the following issues:

- Assignments of scheduler configuration parameters. These parameters are used to configure the system scheduler, and include scheduling policy, task invocation mechanism, and the priority for each task. The assignment should result in a feasible schedule of the system.

- Verification of tasks' timing constraints and devices' resource constraints. These constraints are verified based on the analysis of the task set on each device. The analysis reveals the resource consumption of a single device and link, and the responsiveness of individual tasks. The results help a designer to understand the performance of individual task and resource consumptions for each resource; therefore, optimization of individual task and device usage can be explored.

5.1 Assignments of Scheduler Configuration Parameters

Scheduler configuration parameters are used by the system software — OS in particular — to schedule the task executions. They include both system scheduler parameters and task scheduling parameters. System scheduler parameters are the policies shared by all tasks in the system, including scheduling policies and invocation mechanisms. The scheduling policies are built as part of the OS. Some examples of scheduling policies are first-in-first-out (FIFO), preemptive or non-preemptive, and static or dynamic scheduler. The invocation mechanisms determine how to invoke a process or thread. Typical invocation mechanisms are time-triggered or event-triggered. A time-triggered mechanism invokes the tasks based on nothing but time. Time-triggered architecture [46] uses this type of invocation mechanism. So does the Phase Modification (PM) in [9], where the dependencies of tasks are translated into their relative release times. An event-triggered mechanism invokes tasks based on nothing but events. Most of RTOS such as QNX, RTLinux, and VxWorks use this type of invocation mechanism. An example is the direct synchronization (DS) [82], which triggers the invocation of a task upon the completion of its predecessor. Other examples are combinational mechanisms, such as release guard (RG) [81], which support automatic selection of one of them. The task scheduling parameters, which are dedicated to each individual task, are used by the OS to schedule the task with a combination of system parameters. In an OS using a priority-based scheduler, the invocation period and release offset are used to determine the invocation of a task, and the priority is used for scheduling.

Assignments of the system scheduler parameters are fairly easy since there are only a small number of options in each OS. They are mainly determined by the simplicity of implementation, overheads of the selected mechanism, and desired response time. For example, while preemptive dynamic scheduling policy leads to simple implementation, it may introduce large system overheads as the number of tasks in the system is large. Therefore, it is only suitable when the number of tasks in the system is small. Similarly, the invocation mechanism of DS is simple, introduces low overhead, but results in a longer worst-case end-to-end response time compared to other mechanisms such as PM [81]. As the system scheduler parameters are shared and the task invocation rates are known, we focus on the priority assignments of the tasks in this section.

Every task must be assigned a priority before the runtime model performance can be analyzed. To guarantee the runtime model meets the system performance constraints, we must find a feasible schedule for all tasks in the system. An assignment of these scheduling parameters is called *feasible* if the task set meets all timing constraints under such assignment. A feasible priority assignment should be based on the timing constraints of the tasks in order to satisfy them. Finding a feasible task priority assignment for a system with multiple concurrent task chains is difficult. It has been proved that finding a feasible priority assignment for a system with any scheduling parameters is NP-hard [81]. Further, a feasible priority assignment for one scheduling policy may not be feasible for another. For example, a feasible priority assignment with preemptive scheduling and DS invocation mechanism may not be feasible for one with preemptive scheduling and PM invocation mechanism. Our priority assignment in this work is thus with assumptions of some given system scheduling parameters.

Our priority assignment is a heuristic assignment method. It works as follows. The priorities ($prio_i$) of the tasks (τ_i) in the task set are first assigned using a deadline-monotonic strategy. If the assignment causes any constraint violation of a task, the method iteratively searches for a feasible solution by altering the priority of one task at a time. Two issues should be addressed in this process: which task should be chosen for priority adjustment, and what priority level it should adjust to. Since every task contributes to the constraint violation, and we cannot condemn one or a group of tasks for the violation, our algorithm chooses the tasks in the constraint-violated task chain for adjustment in descending order of their resource demands. Given a priority of τ_i, $prio_i$, the adjustment is done as follows:

1. If $prio_i$ is not the highest priority, raise the priority $prio_i$ of τ_i to a new level $prio'_i$, such that

   $$prio_i < prio'_i \leq prio_{i+1}$$

   where $prio_{i+1}$ is the next higher-level used priority. If τ_i is the task with the highest used priority, we raise τ_i to the next unused priority higher than $prio_i$.

2. Otherwise, lower the priority $prio_i$ of τ_i to a new level, $prio'_i$ such that

   $$prio_{i-1} \leq prio'_i < prio_i$$

   where $prio_{i-1}$ is the next lower-level used priority. If τ_i is the task with the lowest priority, we lower τ_i to the next unused priority lower than $prio_i$.

The new assignment of $prio'_i$ is then evaluated using a technique based on the simulate annealing. This is done through an *energy function*, defined as:

$$E(prio_i) = k_1 * \sum \sigma(TC_i) + k_2 * n_{prio} + k_3 * n_{diff} \tag{5.1}$$

where $E(prio_i)$ is the energy value of assigning $prio_i$ to τ_i, $\sigma(TC_i)$ is the end-to-end timing effect of assigning τ_i's priority to $prio_i$ for the task chain TC_i, n_{prio} is the number of distinct priority levels after introducing $prio_i$, and n_{diff} is the number of communicating components of different priority levels with τ_i's priority being $prio_i$. k_1, k_2, and k_3 are constants, representing weights for each term in the equation.

$\sigma(TC_i)$ can be computed as

$$\sigma(TC_i) = \begin{cases} resp(TC_i) - D(TC_i) & \text{if } resp(TC_i) > D(TC_i) \\ 0 & \text{otherwise} \end{cases}$$

where $D(TC_i)$ is the end-to-end deadline associated with the task chain TC_i, and $resp(TC_i)$ is the response time TC_i under condition τ_i running at the priority $prio_i$.

The energy function $E(prio_i)$ evaluates the system-wide effect of the assignment of $prio_i$ to τ_i. σ represents the timing effects to each task chain. A small value of σ indicates a beneficial assignment. Different values of $resp(TC_i)$ have the same system timing effect when the task chain meets its deadline ($resp(TC_i) \leq D(TC_i)$). A system with no timing constraint violated should have $\sigma = 0$. n_{prior} indicates the system overhead for managing the executions. The overhead increases as the number of priority levels increases. Similarly, n_{diff} affects the execution delays because communicating tasks running at different priorities are more likely to be preempted by other tasks. Values of k_1 and k_2 should be chosen such that the first two terms of $E(prio_i)$ return similar values. Constant k_3 should be smaller than k_2 to minimize the number of priority levels used in the system, and consequently to minimize the context switch overhead. Selection of these constants to yield optimal solutions depends on the problem under consideration, and deserves further research.

The priority assignment iteratively adjusts priority assignment of a task to minimize the value of $E(prio_i)$. If the new assignment $prio'_i$ results in a smaller value of E, it indicates the new assignment is better. When a better assignment $prio_i$ is found, the algorithm starts from the new assignment and continues searching until it cannot find an assignment with a lower energy function value. The process repeats until (i) no task's priority can be adjusted, or (ii) the change of $E(prio_i)$ value is within a small bound in consecutive adjustments.

The priority assignment needs the result of the schedulability analysis in the computation of the energy function value. The schedulability analysis determines the end-to-end response time $resp(TC_i)$ of the task chain TC_i containing the task τ_i.

The tasks' priorities assigned using this method are not sensitive to the invocation mechanism, meaning they can be used along with any invocation mechanism in the final system implementation and performance analysis, although the end-to-end system performance will be different with different invocation mechanisms under the same priority assignment.

5.2 Timing and Schedulability Analysis

We now perform the schedulability analysis to the task set on the same computation device. Any two tasks, τ_i and τ_j, on the same processor can have one of the following relationships:

1. τ_i and τ_j are dependent. In this case, the invocation periods P_i and P_j must be harmonic, and both τ_i and τ_j must contain some components from the same transaction with precedent constraints.

2. τ_i and τ_j are independent but subject to release offset constraints. In this case, the invocation periods P_i and P_j must be harmonic, and there exists a release offset constraint between the concurrent components in τ_i and τ_j. This implies that either τ_i and τ_j contain the dependent components from the same transaction but not immediately next to each other, or they contain

the independent components with release difference constraints.

3. τ_i and τ_j are totally independent. In this case, the invocation periods of τ_i and τ_j, and P_i and P_j, may be either non-harmonic or harmonic but will contain no component from the same transaction.

The results of this analysis include the response time of each task, the resource consumption, and the utilization of the processor. Since the response time of a task depends on the release phase of all the tasks on the processor, we bound the response time of a task with its best-case response time and worst-case response time under different release phases.

5.2.1 Best-case response time

To compute the best-case response time of a task, we need to first determine the best-case phasing of the task set on a processor. According to [68], the best-case phasing for a set of tasks with fixed priorities is when a task τ_i completes at exactly the same time when all tasks with higher priorities are released. The best-case response time of a task τ_i can then be computed as:

$$resp_b(\tau_i) = w(\tau_i) + \sum_{j \in hp(i)} \lceil \frac{resp_b(\tau_i) - P_i}{P_j} \rceil \cdot w(\tau_j) \tag{5.2}$$

where $resp_b(\tau_i)$ is the best-case response time of the task τ_i, $w(\tau_i)$ is the computation resource demand of τ_i (in execution time), and P_i is the invocation period of τ_i. $hp(i)$ defines a set of tasks that run on the same processor and at higher priorities than τ_i's. The equation can be iteratively solved by using the worst-case response time of τ_i as initial $resp_b(\tau_i)$. It is shown in [68] that any initial value within the same busy period of τ_i will converge to the best-case response time using Eq. (5.2). The iterative computation of the best-case response time of τ_i is then:

$$\begin{aligned} &resp_b^{n+1}(\tau_i) = w(\tau_i) + \textstyle\sum_{\forall \tau_j \in hp(i)} \lceil \frac{resp_b^n(\tau_i) - P_i}{P_j} \rceil \cdot w(\tau_j) \\ &resp_b^0(\tau_i) = resp_w(\tau_i) \end{aligned} \tag{5.3}$$

In Eq. (5.3), the initial best-case response time, $resp_b^0(\tau_i)$ is set to be τ_i's worst-case response time $resp_w(\tau_i)$, whose computation will be discussed later in this section. The iterative process stops when two consecutive $resp_b(\tau_i)$ values are the same, i.e., $resp_b^{n+1}(\tau_i) = resp_b^n(\tau_i)$.

Eq. (5.3) need to be modified to be used for the task set on the same processor in our runtime model due to different system model assumptions. The system model used to derive Eq. (5.3) assumes

1. for any task, its relative deadline must be less than its period;
2. all tasks are independent with zero release offset;

The first assumption ensures that there is only one instance of a task at any given time, thus no self-interference exists. Such an assumption holds true in our runtime model. Although the end-to-end deadline of a task chain can be longer than the task chain's invocation period (multiple instances of task chain), the execution window of a task is less than its period ($d_i - o_i \leq P_i$). Therefore only one instance of a task can be active at any given time.[1] The second assumption, on the other hand, does not hold. In our runtime model, precedent constraints exist among the tasks, and independent tasks may also have non-zero release offsets, as Case 1 and 2 stated at the beginning of this section. In this work, we modeled both in non-zero released offsets of the tasks, which limit the number of tasks that can be invoked simultaneously. So the number of higher priority tasks that interfere with τ_i's execution may be smaller. With such consideration, the best-case response time of τ_i can be computed in the following way: Given τ_i with priority $prio_i$, the high-priority task set $hp(i)$ first consists of the tasks τ_j with priorities $prio_j > prio_i$ whose executions can overlap. This

[1]This is the same as pipeline executions.

is equivalent to the assumption that the busy period of τ_i contains only tasks that can be invoked simultaneously. Particularly, for a task τ_j with $prio_j > prio_i$, if there exists a different task τ_k with $prio_k > prio_i$ and it satisfies:

- $P_k = P_j$, and
- $o_i + w_i < o_j$ or $o_j + w_j < o_i$,

we add both τ_j and τ_k to $hp(i)$. Otherwise, we add the one with minimum resource demands to $hp^0(i)$:

$$hp^0(i) = \begin{cases} hp^0(i) \cup \tau_k & \text{if } w_k \leq w_j; \\ hp^0(i) \cup \tau_j & \text{otherwise.} \end{cases} \tag{5.4}$$

Thus-obtained $hp^0(i)$ contains only tasks that can be invoked simultaneously. We then compute $resp_b(\tau_i)$ using Eq. (5.3). If the computed $resp_b(\tau_i) > o_j - o_i - w_i$ for some higher-priority task τ_j that is not in $hp(i)$, it indicates that τ_j is invoked at least once during τ_i's busy period. Thus, τ_j should be included in $hp(i)$ in computation of $resp_b(\tau_i)$. The new $hp(i)$ is:

$$hp^{n+1}(i) = hp^n(i) \cup \{\tau_j | j : o_j - o_i - w_i > resp_b(\tau_i)\} \tag{5.5}$$

The computation of $resp_b(\tau_i)$ is then repeated with the new $hp^{n+1}(i)$ using Eq. (5.3). The process continues until either $hp(i)$ does not change or $resp_b(\tau_i)$ does not change. Thus-obtained $resp_b(\tau_i)$ is the best-case response time for the task τ_i.

5.2.2 Worst-case response time

The worst-case response time of τ_i is determined by finding the worst-case phasing of τ_i in presence of interferences from other tasks with higher priorities. We apply the technique in [67].

For a given task τ_i, we first need to determine the interferences caused by the tasks with priorities higher than $prio_i$. During the k-th busy period of τ_i, the interference can be computed as follows:

$$I(\tau_i, k) = \sum_{\forall \tau_j \in hp(i)} \lceil \frac{resp_w(\tau_i, k) - \Delta_{ij}(k)}{P_j} \rceil \cdot w(\tau_j) \tag{5.6}$$

where $I(\tau_i, k)$ is the interference caused by all higher priority tasks during k-th busy period of τ_i; $hp(i)$ is the task set containing the tasks with $prio_j > prio_i$; $resp_w(\tau_i, k)$ is the worst-case response time of τ_i during the k-th busy period; $\Delta_{ij}(k)$ is the release phasing between τ_i and τ_j during τ_i's k-th busy period; P_j is the invocation period of τ_j; and $w(\tau_j)$ is the resource demand of task τ_j in computation time. The worst-case release phasing between τ_i and a higher priority task τ_j during τ_i's k-th busy period can be computed as:

$$\Delta_{ij}(k) = \lceil \frac{a_i^k - o_j}{P_j} \rceil \cdot P_j - (a_i^k - o_j) \tag{5.7}$$

where a_i^k is the arrival time of the k-th invocation of τ_i, which can be computed as:

$$a_i^k = k \cdot P_i + o_i$$

With the inference computed using Eq. (5.6), the worst-case response time of the k-th invocation of τ_i can be computed as:

$$\begin{aligned} resp_w(\tau_i, k) &= w(\tau_i) + I(\tau_i, k) \\ &= w(\tau_i) + \textstyle\sum_{\forall \tau_j \in hp(i)} \lceil \frac{resp_w(\tau_i,k) - \Delta_{ij}(k)}{P_j} \rceil \cdot w(\tau_j) \end{aligned} \tag{5.8}$$

Same as the best-case response time, $resp_w(\tau_i, k)$ can be computed iteratively.

As Eq. (5.8) determines the worst-case response time of the k-th invocation of τ_i, the overall worst-case response time of τ_i is the maximal one among all instances. Since the process repeats for every duration of LCM, we only need to check for $k = 0, \ldots, LCM/P_i$. Thus, the worst-case response time of τ_i can be computed as:

$$resp_w(\tau_i) = max_{0 \le k \le LCM/P_i} resp(\tau_i, k) \tag{5.9}$$

5.2.3 Resource utilization

The computation of resource utilization is straightforward. The device utilization can be computed as

$$U_{PROC_p} = \sum_{\tau_i \in PROC_p} \frac{w(\tau_i)}{P_i} + oh \tag{5.10}$$

where $w(\tau_i)$ is the resource demand of task τ_i allocated on device $PROC_p$, P_i is τ_i's invocation period, and oh is the resource consumptions of system services, typically including the timing service and the scheduling service. The resource consumptions of the timing service depend on the number of timer signals generated during the execution, which consequently depend the number of timers (number of different invocation rates) and the timer resolution. The number of signals m depends on the implementation strategy of OS [43]. If the implementation strategy is time-driven, the overhead can be computed as

$$oh(time) = \frac{w_t}{res}$$

where res is the timer resolution. If the implementation strategy is event-driven, the overhead can be computed as

$$oh(time) = \frac{\sum_i \frac{LCM}{P_i} \cdot w_t}{LCM}$$

where LCM is the least-common-multiple of all invocation periods on the examined device. The scheduling overheads depend on the number of context switches during the worst-case busy period execution. For the worst-case response time $resp_w$ of the lowest priority task, the overhead can be computed as

$$oh(sched) = \frac{\sum_{\forall \tau_i} \lceil \frac{resp_w}{P_i} \rceil \cdot w_s}{resp_w}$$

where w_s is the resource demand of each context switch.

Similarly, the utilization of the communication device can be computed as

$$U_{L_{p,q}} = \sum_{msg_i \in L_{p,q}} \frac{w(msg_i)}{R(L_{p,q})} \tag{5.11}$$

where $w(msg_i)$ is the resource demand of message msg_i passing over link $L_{p,q}$, which connects device $PROC_p$ and $PROC_q$; $R(L_{p,q})$ is the capacity of the link, usually represented in bandwidth. $w(msg_i)$ reflects both the resource demand of the individual message in transferred data size and the message generation frequency, i.e., $w(msg_i) = f_i \cdot size(msg_i)$.

5.3 Reducing Analysis Complexity

As can be seen from the above discussion, deriving the response time of a task is computationally complex. It requires construction of the higher priority task set for all tasks on the same processor, and identification of the tasks with mutually exclusive executions. Moreover, the computation of the response times of a task, regardless of the best-case or worst-case, is expensive. A major factor causing the complex analysis is the existence of task dependencies. If these dependent tasks can be transformed into a set of independent tasks, the analysis can be significantly simplified using classic schedulability analysis algorithms. To this end, we propose a solution to reduce analysis complexity by eliminating task dependencies.

Our elimination method is based on the introduction of shared buffers between the communicated tasks. This in turn is based on the fact that the dependent tasks must communicate through data and/or events. Given a task chain TC with dependent task set τ, we transform TC to an equivalent system TC' with task set τ', where

1. $|\tau'| = |\tau|$;
2. for each $\tau_i \in \tau$, there exist a corresponding task $\tau_i' \in \tau'$;
3. all tasks in τ' are independent;

4. for each directed link $l_{ij} : \tau_i \rightarrow \tau_j$ in TC, there is a buffer B_{ij} between τ_i' and τ_j', which τ_i' writes its outputs to, and τ_j' reads its input from.

τ_j' periodically checks B_{ij} for updates from τ_i'. The period at which τ_j' checks for buffer updates is called the *polling period* of τ_j'. Such a transformation ensures that the function of TC is preserved in TC'.

The shared buffer approach is different from the release offsets used to solve task dependencies in the current runtime model. In the release offset approach, the tasks after transformation are still dependent, using the release offsets as the synchronization method instead of direct triggers. On the other hand, the tasks with shared buffers are truly independent, meaning τ_j' and τ_i' execute without knowing each other's existence. Therefore, their release offsets can all be zero.

A key issue in using shared buffers to solve the task dependencies is how to generate outputs using valid data with original timing constraints. The timing constraints of the original end-to-end transaction can be considered the lifetimes of the input and all intermediate derived data. They must be satisfied to ensure the outputs are generated using the valid unexpired data. This is implemented by letting the tasks check the shared buffers for updates at some fixed intervals after the transformation. Given a task τ_j' in TC', which depends on τ_i in original TC, the worst-case τ_j' detecting update of B_{ij} occurs when τ_i' updates B_{ij} immediately after τ_j' checks it. Since τ_j' can only detect the updates in the next polling cycle in such cases, the delay can be at most one polling period. Therefore, the condition under which every task in a transformed system meets the end-to-end timing constraints D can be expressed as:

$$poll(\tau_j') + resp_w(\tau_j') \leq d(\tau_j) - o(\tau_j) \tag{5.12}$$

where $poll(\tau_j')$ is τ_j''s polling period, $resp_w(\tau_j')$ is τ_j''s response time, $d(\tau_j)$ is τ_j's deadline in TC, and $o(\tau_j)$ is τ_j's release offset time in TC. The term $d(\tau_j) - o(\tau_j)$ defines the maximum allowable processing time for τ'_j.

If some task in TC' does not satisfy Eq. (5.12), it indicates that the worst-case update-detection delay for the task is longer than required, and may cause violation of end-to-end timing constraints. To reduce the detection delay, we shorten the task polling period using Eq. (5.13):

$$\begin{gathered} excess(\tau'_j) = P(\tau'_j) + resp_w(\tau'_j) - d(\tau_j) + o(\tau_j) \\ poll'(\tau'_j) = poll(\tau'_j) - excess(\tau'_j)/2. \end{gathered} \tag{5.13}$$

where $poll'(\tau'_j)$ is the new polling period of τ'_j, and $excess(\tau'_j)$ is defined as the amount of time by which τ'_j exceeds its maximum allowable execution time. Since shortening the polling periods increases the system workload in each iteration and will eventually lead to an unschedulable task set, the method is guaranteed to terminate in a finite number of iterations.

After transformation, schedulability analysis is required to ensure the system-wide timing correctness. Because polling operation and data processing have different execution times, we replace a polling task by two tasks in the analysis: one dedicated to the polling behavior and the other to data processing. The one for data processing is only executed when the shared buffer is updated, and is released with a fixed offset equal to the polling period to ensure the data availability. These two tasks can then be considered as independent tasks and schedulability analysis can be performed using classic techniques.

Note that the reduction method comes with a price of scheduling overheads, which result in some systems becoming unschedulable after the transformation. This is because shortening the polling period of a task makes the task run unnecessarily frequently, and hence consumes more resources. One way to avoid the schedulability overheads is to use shared buffers with more-than-one element entries. The size of the buffer can be derived according to the generation frequency, consuming frequency, and data lifetime. Although increased shared buffer sizes solves the timing issues, it can only achieve this by requiring more storage spaces. It is the designer's responsibility to make the trade-offs among accurate analysis, scheduling overheads, and storage overheads.

5.4 Related Work

Methods for schedulability and timing analysis have been studied extensively in real-time system research. Many solutions have been developed and evaluated for various situations. Except for systems using time-triggered mechanisms [46, 31], the runtime system analyses are all based on priority-based scheduling. Traditional priority assignment methods include rate-based and deadline-based assignments [81]. To find a feasible assignment more effectively, Tindell *et al* proposed a solution based on simulated annealing [86]. Garcia and Harbour used another approach in HOPA (Heuristic Optimized Priority Assignment) algorithm [24] based on computation of "excess" response time or computing time to iteratively adjust the priority assignment.

After the priorities of the tasks are determined, many priority-based schedulability analysis techniques can be applied. Some widely-used techniques include rate-monotonic analysis [57], deadline-monotonic analysis [5], and HKL analysis [29]. These analytic methods usually solve only the uniprocessor schedulability analysis problem, and only with assumptions such as periodic independent task set and deadline less than or equal to tasks' periods. These methods have been extended in many ways to deal with dependent tasks, arbitrary timing attributes, multiple processors, and various access and execution control mechanisms. Lehoczkey first introduced the analysis method based on busy period [54]. The busy period analysis is used in [9] to analyze dependent task sets in a distributed environment using phase modification mechanism for execution control. Palencia *et al* developed techniques based on busy period analysis to derive the best-case and worst-case of distributed task sets [62, 63]. Redell further improved the algorithms to analyze the tasks with precedent constraints [68, 67]. Other analytical methods such as holistic analysis [87] and end-to-end analysis with different execution controls [81] have also been developed.

5.5 Summary

Schedulability analysis of the runtime model is an essential step in ECSW design in order to verify that timing and resource constraints are all met at system-level. In this work, we have attempted to apply existing schedulability and timing analysis techniques to our runtime model for the analysis instead of developing a new one dedicated to our model alone. This justifies the fact that our runtime model contains sufficient information for such a runtime analysis, and hence, can be used with other analysis algorithms to investigate different performance issues. Due to different modeling assumptions, applying existing analysis algorithms to our runtime model is not straightforward. Specifically, the runtime model generated with our method contains dependent tasks with their dependencies modeled in their release offsets. We must fit our runtime model to the assumptions of selected analysis algorithms.

In this work, we have chosen fixed priority-based scheduling analysis techniques to analyze our runtime model. Such analysis requires the priority assignment of all tasks. As finding feasible priority assignments is NP-hard, we use a heuristic priority assignment based on simulated annealing to determine the priorities for the tasks in the runtime model. The analysis then focuses on determining the tasks schedulability and resource consumptions on individual processors, including best-case and worst-case response times of tasks using the method based on release phasing analysis presented in [68, 67]. We modify the algorithms to take into account the task dependencies so we can obtain tighter bounds. Since the tasks' timing attributes in our runtime model are assigned in such a way that meeting the deadlines of all tasks results in the end-to-end timing constraints being met, passing a local schedulability analysis guarantees the system performance constraints will be met. Other performance metrics such as output jitters and separations can also be determined using the best-case and worst-case response times. The resource consumptions of computation devices and communication links are computed as utilizations.

As the analysis based on the busy period analysis is computationally complex, we have developed a method using shared buffers to eliminate the task dependencies, and consequently reduce the analysis complexity. With shared buffers, the dependent tasks in the system can be transformed to independent tasks polling the shared buffers at predefined intervals. The polling intervals of the tasks are derived with consideration of their timing constraints. The analysis of the task set after the transformation is simplified because all the release offsets and task dependencies are eliminated. Many classic schedulability and timing methods can then be applied with better scalability. However, the simplification of schedulability analysis comes at the cost of low resource utilization for the application, which implies more resources are wasted. Other solutions such as increasing the size of the shared buffers can also be developed to balance the scheduling overheads against storage costs.

CHAPTER 6

Performance Measurement Methods

The performance measurement methods are essential for collecting required performance information to support quantitative analysis. In our modeling framework, the performance characteristics of both application components and system software services must be measured. This chapter presents a measurement method that allows a user to measure the performance characteristics of these components systematically. The method allows the measurements to be made without the source code of the system software and with only small perturbations. The method is an end-to-end measurement combining both synthetic workload and micro-benchmarking. As the measurement of the application components is straightforward and easy, we will focus on the system service measurements in this chapter. After a brief discussion of the measurement method itself, we will present the application of our method to measure selected real-time operating systems (RTOS) and will demonstrate effectiveness of the measurement method.

6.1 Measurement Method

Our measurement method is an end-to-end measurement. End-to-end (e2e) measurements obtain performance information as an external observer, and therefore, minimize interference during measurements. They also provide performance results very close to what the application will experience at runtime.

Almost all current measurement strategies [37, 56, 80] are based on monitoring events in a system. Methods for event monitoring can be classified as event-driven methods, tracing methods, sampling methods, and indirect measurement [56]. Event-driven methods record event occurrences. Tracing methods are similar to event-driven but they record more system information that can be used to uniquely identify the event occurrence. Sampling methods record the state of a system in a fixed interval instead of recording every event. Indirect measurement methods are *ad hoc* designed and used only when the metrics cannot be directly measured. Among these methods, event-driven and tracing methods require to instrument the examined services and thus perturb the measurements. Although such measurements can provide the details of the measured service, the nonlinear and non-additive perturbations make the results inaccurate, while the performance details of activities inside a measured service are normally unnecessary for the application performance analysis. Indirect measurements are usually avoided whenever a direct measurement is possible. Sampling methods, on the other hand, have the following advantages: (1) no kernel instrumentation is required; (2) the measured information close to an application may experience; and (3) the least perturbations are introduced (but can only provide statistical results). Further, the sampling method gathers information based on external observations, which is suitable for reuse in application performance analysis since applications are also external observers for RTOS services. Therefore, the method based on sampling event occurrence is a suitable tool for our measurements.

Our e2e measurement method is combined with micro-benchmark and synthetic workloads to obtain accurate performance measurements of a service. To measure performance of a service for use in multiple application analysis, the measurement method should consider both the effects of the underlying system software service and the interactions at the same functional level and above. This is because the performance depends on both the execution environments and the system workloads. The system software services provide an execution environment for the measured service, while other services at the same level or above introduce workloads to the system. Microbenchmarks [11, 12] are an effective method for measuring individual and independent services without instrumenting the kernel. However, as typically used, they exercise the system in a limited way that is not necessarily representative of an actual application workload. We can derive more realistic performance metrics by coupling the micro-benchmarks with representative, domain-specific synthetic workloads. Synthetic workloads [45] allow the measurements to be taken under conditions close to the real applications with representative resource usage and interaction patterns. Through such synthetic workloads, our measurements can cover the most frequently used application configurations and the interaction patterns so the results can be reused to analyze a family of applications.

The limitation of this measurement method is that measured performance values will be dedicated to a platform configuration, including hardware and system software. This requires that components and services should be measured on all possible combinations of hardware and system software. Such an approach is possible for ECSW due to the fact that only a handful of hardware and OS products are commonly used for a given family of applications. It is also possible to derive the performance values of the system based on the low-level system service performance obtained using other micro-benchmarks such as *lmbench* or *hbench-OS*. However, there is no algorithm in current practices that can derive the performance values of high level services accurately using values of low level services.

6.2 Measuring RTOS service overheads

As defined in the modeling and analysis framework, performance characteristics are service demands, which are derived from the measured execution times. Measuring performance characteristics of system software services are relatively complicated, as they depend on application-level workloads, hardware architecture, and the interactions among services at the same level. Therefore, we evaluate the effectiveness of our measurement method based on system software services measurements. For simplification, we only apply the method to OS-level services. Other level services such as middleware services can be measured in a similar way.

We measure two basic RTOS services: timing and scheduling services. Timing services include various clock and timer management mechanisms implemented in an RTOS, and the performance information for such services is critically important to time-based activities in ECSW, such as sampling at a given rate and generating periodic outputs. Scheduling services, on the other hand, are essential for software execution, and the performance of such services plays a critical role in assessing the quality of the entire system. Timing and scheduling services are the basic services required by all ECSW executions and supported in all RTOSs. Many other system services such as network and disk management usually depend on them.

For application performance analysis, the overheads and unpredictability introduced by the system software services are critical in meeting the system timing constraints and achieving stable control [100, 97]. Since timing and scheduling services normally run as part of the RTOS kernel at the highest priority in the system, applications can experience large overheads and unpredictability due to these services. Our measurements should, therefore, reveal the variations of these overheads and unpredictability under different system and application configurations, and store them with these services. When performance analysis needs to be done, such measured information can then be used to construct an accurate performance model based on the given application usage and system configuration.

To achieve such a measurement goal, we define a set of performance metrics for overheads and unpredictability which are reusable for the analysis of a family of applications. We then construct synthetic workloads to enumerate all possible ways of using timing and scheduling services in ECSW applications, and measure the service performance using micro-benchmarks. Our measurements were done on a set of selected hardware and RTOSs to demonstrate the general applicability of our measurement methods. Note that the performance of different hardware and RTOSs is not measured to compare and select targets. Instead, they are measured for constructing performance analysis models for a family of applications that will be executed on these targets.

6.2.1 Measurement environment

Since the hardware provides an execution environment for RTOSs, the measured performance of RTOS services is hardware-dependent. Here we do not assume the existence of methods for deriving RTOS service performance from the hardware performance and configuration. Instead, we have designed a set of experiments for each different combination of hardware and operating system to obtain the performance of RTOS services directly. Since RTOSs have different implementation strategies and provide different ways of using these services, it is important to learn the effects of such differences on performance. Thus, a set of RTOSs was selected to run on different hardware for our measurements. Table 6.1 lists the hardware we used in the measurements.

Hardware	Processor type	Processor speed (MHz)	Memory size (MB)	Cache (KB)	Bus speed (MHz)
P133	Pentium	133	32	128	66
P166	Pentium	166	32	128	66
ETAS2000	MPC 555	40	2	-	20

Table 6.1: Hardware configurations for OS service measurements.

The operating systems we measured include QNX 4.24, OSEKWorks 2.0, and RTLinux 3.0. We assume that the source codes of these kernels are not available for measurement, although the source code availability of RTLinux 3.0 will help us understand the relationships among the OS services and verify our analysis results.

The versions of selected RTOSs we used, cannot run on all the hardware platforms. The configurations used for measurements are given in Table 6.2.

Hardware	QNX	RTLinux	OSEKWorks
P133	QNX-P133	RTL-P133	-
P166	QNX-P166	RTL-P166	-
MPC555	-	-	OSEK-MPC

Table 6.2: Testbed configuration for measurements.

To minimize interference from non-essential services, the services such as drivers for disk, network, and display were all turned off during the measurement. Collected data were temporarily stored in the main memory and were dumped into the destination host for analysis after the services used for storage were turned on at the end of the measurement.

6.2.2 Performance metrics

The measured performance metrics of OS services are expected to be minimum in number, reusable for a family of applications, and independently measurable. A minimum set of metrics is desired to save the amount of experimental effort and reduce the amount of performance information to be stored. Finding a minimum set of performance metrics for a service requires an understanding of the analysis requirements, dependencies among the OS services, and the relationships between different metrics. For example, the timing service in many OS implementations depends on the signal mechanism, which enables the values of the timing service performance metrics to be derived from those of the signals and do not need to be measured. Reusability means that the metrics are measured once and reused whenever the same environment is used, thus eliminating duplicate measurements for the same environment for different applications with similar workloads and interaction patterns. Finally, the independently measurable metrics simplify the experiments and data analysis, and are more flexible when they are used in a performance model.

Our measured performance metrics in this work include clock overheads and interval jitters for timing services, and context switch overhead for scheduling services. Clock overheads are the CPU time to process each signal generated from the system clock, and are one of the overheads for application performance analysis. Interval jitters are the variance of time intervals and are used to represent the unpredictability in analysis. Context switch overheads are the time that a kernel spends for terminating one task and putting another ready task into execution. They are another set of overheads in analysis. Although the context switch is defined loosely as in [12], the measured values are closer to what a task will really experience at runtime.

The metrics represent a minimum set of performance metrics for ECSW analysis of our target real-time control, such as avionic mission control or automotive engine control. The analysis requirements of real-time controls include finding the response times and the used and/or spared resources, which can usually be done by using real-time analytic algorithms, such as algorithms to compute busy periods, time-demand functions, and processor utilizations. Among these metrics, clock overheads and context switch overheads specify the resource requirements of the services since they introduce additional computation workload into the system. Clock overheads also determine the completion time of an activity, therefore are necessary for computing the activity response time and the system's busy period. On the other hand, the interval jitter does not introduce additional workload, but is useful in determining the length of a busy period and completion time of an activity. Therefore, these metrics, along with application component performance characteristics, such as worst-case execution times, are sufficient for the analysis algorithms to generate the results required by the application. Clock overheads, interval jitters, and context switch overheads represent basic performance characteristics of timing and scheduling services. All analysis algorithms are either based directly on these metrics or on some measured information that can be derived from these metrics. Therefore, the metrics can be reused for different analyses. Furthermore, RTOSs usually provide interfaces to access timing and scheduling services, and the effects can be seen in an application. So our metrics can be measured individually without kernel instrumentation.

6.2.3 Measurement tool and analysis strategy

Our measurement tool is based on sampling the processor clock cycles for each measured case. Most modern processors are equipped with some registers dedicated to performance and timing measurements. In our work, we use the hardware Time-Stamp Register on the Pentium processor [35] and the Time-Base Register on MPC 555 [61] as a means of measurement. Both of them are 64-bit registers, initialized to 0 when the system powers up, and increased by 1 upon every hardware clock tick.

A statistical analysis method is used to process the measured data. For each experiment, $10,000$ samples are collected during the normal execution. Besides computing the average and standard deviation of the measured parameters, we also find the maximum and minimum values as performance bounds for a measured parameter on a given platform.

6.2.4 Experiment design

Measuring the performance of system services should consider the effects of both application and hardware in the layered model. In this work, we assume the 3-level system realization layers with the higher level performance depending on the service at the lower-level and the interferences of the same level activities, as shown in Figure 6.1. The hardware layer provides the environment that the measured services will be running in, and workloads at the application level can make measured services behave with different performance. Since each of our measurements in this work is dedicated to one combination of hardware and RTOS, the experiment design focuses only on generating application-level workloads. Such workloads are designed to cover all possible scenarios of the application usages so that the measured information can be reused for different analyses.

Experiments for timing service measurement. Measured metrics of timing services include clock overheads and interval jitters. Clock overheads depend mainly on the clock resolution, there-

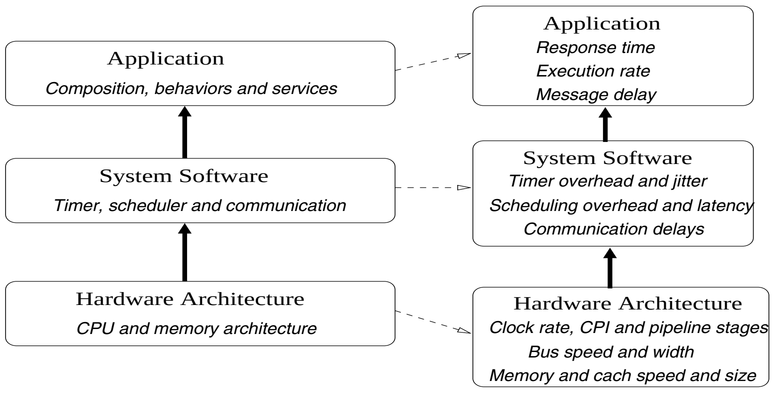

Application
Composition, behaviors and services
System Software
Timer, scheduler and communication
Hardware Architecture
CPU and memory architecture
Application
Response time
Execution rate
Message delay
System Software
Timer overhead and jitter
Scheduling overhead and latency
Communication delays
Hardware Architecture
Clock rate, CPI and pipeline stages
Bus speed and width
Memory and cach speed and size

Figure 6.1: System performance layered model for measurements.

fore can be measured by executing a test program under different resolutions. Specifically, given a program P with execution time e, the overhead of each clock tick can be computed using Eq. (6.1).

$$e_m = e + I_0 \cdot o + I_1 \cdot o + I_2 \cdot o...... \tag{6.1}$$

where e_m is the measured execution time of P, e is the real execution time of P, r_m is the clock resolution when the measurement is taken, and o is the overhead of processing each clock tick. Each item of $I_i \cdot o$ represents the overhead of processing the clock ticks happening during the time $I_{i-1} \cdot o$. I_i is the coefficient of the i-th order overhead, and can be calculated recursively as follows:

$$I_0 = \lfloor \frac{e}{r_m} \rfloor, \quad I_1 = \lfloor \frac{I_0 \cdot o}{r_m} \rfloor, \quad I_2 = \lfloor \frac{I_1 \cdot o}{r_m} \rfloor, \quad ...... \quad I_n = \lfloor \frac{I_{n-1} \cdot o}{r_m} \rfloor, ... \tag{6.2}$$

It can be seen from Eq. (6.2) that I_i decreases exponentially. Therefore, given any e, there exists a positive integer n so that for any $N > n$, $I_N = 0$, as the overhead introduced by its previous item will eventually be less than r_m. In OS measurements, the order of I_i seldom exceeds 2 as the e is normally tens of millisecond and given OS overheads are around microseconds. Therefore, the clock overhead can be computed as follows:

$$e_m = e + \lfloor \frac{e}{r_m} \rfloor \cdot o + \frac{\lfloor \frac{e}{r_m} \rfloor}{r_m} \cdot o^2 \tag{6.3}$$

$$e_m = e + \lfloor \frac{e}{r_m} \rfloor \cdot o, \tag{6.4}$$

Eq. (6.3) is used when the clock resolution is fine (normally less than $0.5ms$) and/or the execution time is long, while Eq. (6.4) is used for other cases.

It is essential to know the execution time e of P to compute clock overhead o. The measured execution time e_m is usually larger than e since the clock signal will be generated no matter whether it is used or not, and its overheads are included in the measured e_m. However, since e_m only includes the clock overheads that occur during the e_m measurements, we can set the clock resolution far larger than e to obtain a measurement e_0 that is close to real e, as given in Eq. (6.5).

$$e_0 = e, \qquad for \;\; r_m \gg e \tag{6.5}$$

In our experiment, the test program P is designed with an execution time of $10ms$. The resolutions used for the clock overhead measurements range from $100\mu s$ to $100ms$.[1] The selected clock resolutions and methods to set them are given in Table 6.3.

RTOS	method to set resolution	values (μs)
QNX	clock_setres()	100, 200, 400, 500, 600, 800, 1000, 10000, 100000
RTLinux	rtl_setschedmode()	100, 200, 400, 500, 600, 800, 1000, 10000, 100000
OSEKWorks	Rtcinit()	100, 200, 400, 500, 600, 800, 1000, 10000, 100000

Table 6.3: Resolution setting methods and values for examined OS.

Interval jitters, on the other hand, depend on the clock resolution and application configurations. The dependency between interval jitters and clock resolutions is caused by the process of a timer expiration signal has to be rounded up to a clock tick. The dependency between interval jitters and application configuration results from the usage of the service, such as the number of independent timers, the interval patterns of these timers, and the priorities of tasks using these timers. The

[1]The resolution range is chosen based on the capacity test of a platform and the usage in applications.

resolution values in our experiments are selected to be 500 μs and 1 ms.[2] The experiment workloads are designed with different patterns of timer intervals and different numbers of timers. Table 6.4 lists the values we used in our jitter measurements.

Factor	values
clock resolution	500 μs, 1 ms
timer patterns	harmonic, non-harmonic
task priority	highest, medium, lowest
number of timers	1, 2, 5, 10, 15, 20

Table 6.4: Factors and values for interval jitter measurements.

A set of test programs is designed to perform the jitter measurements with the listed system attributes. Since only one timer can be associated with a process or thread in all examined RTOSs, we need up to 20 tasks in the experiments. Table 6.5 lists the attributes of every experiment task,[3] while Table 6.6 shows the combinations of the tasks on measuring the performance with different numbers of timers.

task id	period (harmonic) (ms)	period (non-harmonic) (ms)	priority	task id	period (harmonic) (ms)	period (non-harmonic) (ms)	priority
1	1	2	1	11	80	83	11
2	5	5	2	12	90	89	12
3	10	9	3	13	100	103	13
4	15	17	4	14	150	145	14
5	20	19	5	15	200	211	15
6	30	31	6	16	250	239	16
7	40	43	7	17	300	217	17
8	50	49	8	18	400	395	18
9	60	61	9	19	500	513	19
10	70	71	10	20	600	613	20

Table 6.5: Attributes of experiment tasks.

Experiments for scheduling services measurements. The metrics for scheduling services is context switch overheads. We define the context switch time loosely as in [12], which is the kernel

[2]The values are chosen with the consideration that the timer overheads introduced by these values should be close but potentially have a signifi cant impact on jitters.

[3]In this table, a smaller number is used for a higher priority. For the RTOS with a larger number for a higher priority, the real priorities of these tasks should be converted.

test case	1	2	3	4	5	6
# of timers	1	2	5	10	15	20
task in set	{1} or {10} or {20}	{1, 10} or {1, 20}	{1, 5, 10, 15, 20}	{1, 3, 5, 7, 10, 11, 13, 15, 17, 20}	{1, 3, 4, 5, 7, 9, 10, 11, 13, 14, 15, 17, 18, 19, 20}	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}

Table 6.6: Task combinations for the test cases.

time spent between the termination of the current running task and the beginning of a new ready task. The overheads actually include the termination time to suspend the current running task and store its environment, the scheduling time to select a new ready task to run, and the activation time to restore the execution environment of the new task and give the control to it. However, these times are not individually measurable without kernel instrumentations, hence are usually not doable for end users. But the total elapsed time from the termination of one task to the start of a new task is measurable without a source code, and is closer to what applications will actually experience at runtime. Therefore, such measurements are more suitable and reusable for application performance analysis.

Context switch overheads depend on the scheduling algorithm, the number of tasks in the ready queue, and the organization of the ready queue (for example, sorted queue or unsorted queue). The priority-based preemptive scheduling algorithm is one supported by all current RTOSs and used most frequently in ECSW applications. Our context switch overhead measurements are therefore performed using this scheduling algorithm. The task set is with a range of 2 to 20 tasks. The measurements are taken between two specially designed tasks in the task set. All other tasks, called *interference tasks*, are introduced only to change the length of the ready queue to learn the effect of the queue length on context switch overheads. The task set is checked to be schedulable manually before the measurements. Table 6.7 shows these tasks and their attributes used for scheduling service performance measurements.

task id	priority	period (ms)
1	2	1
2	1	triggered by task 1
3-20	3	1

Table 6.7: Attributes of experiment tasks.

The measurement with the given task set is designed as follows: $Task1$ runs periodically with $1ms$ period, and will trigger $Task2$ when it is finished. The priority of $Task1$ is lower than the priority of $Task2$ but higher than all interference tasks. The values of Time-Stamp register or Time-Base register are logged at both the end of $Task1$ and the beginning of $Task2$. Interference tasks run at the same period as $Task1$. Therefore, at every $1ms$, all tasks are ready except $Task2$ and will be moved to the ready queue. Since $Task1$ has the highest priority in the ready queue, it executes first. After it completes, $Task2$ is triggered and will be in the ready queue. Similarly, $Task2$ is the highest priority task and will execute before any other task. Thus, $Task3 \sim 20$ only affect the ready queue length during the measurement, and do not contribute any overhead to the context switch time between $Task1$ and $Task2$. The context switch overheads can then be obtained from the difference between the pairs of sampled values of $Task1$ completes and $Task2$ starts. All interference tasks are assigned the same priority since their priorities have no effect on the measurements. Table 6.8 shows the number of tasks used for each measurement to learn the effect of the ready queue length.

test case	1	2	3	4	5
# of tasks	2	5	10	15	20
task in set	{1, 2}	{1, 2, 3-5}	{1, 2, 3-10}	{1, 2, 3-15}	{1, 2, 3-20}

Table 6.8: Task combinations for test cases.

6.3 Measurement Results

Given the measurement goal and designed experiments, we performed the measurements of timing and scheduling services, and now present the measured results in this section.

6.3.1 Results of clock overhead measurements

The measured execution times of the test program with different clock resolutions are presented in Table 6.9. Note that the original measured execution times were represented as the number of clock cycles, and we converted them to real wall clock times in Table 6.9.

platform	statistics	resolutions (ms)							
		0.1	0.2	0.4	0.5	0.8	1	10	100
QNX-P133	average	10.725	10.355	10.171	10.133	10.078	10.065	10.006	10.0005
	maximum	13.871	11.845	10.875	10.674	10.261	10.154	10.015	10.012
	minimum	10.702	10.339	10.157	10.119	10.069	10.057	10.005	10.0002
	std	0.085	0.043	0.018	0.014	0.008	0.006	0.002	0.0002
RTL-P133	average	12.732	11.287	10.676	10.548	10.353	10.301	10.099	10.078
	maximum	20.912	13.589	11.613	11.222	10.754	10.638	10.132	10.092
	minimum	12.714	11.298	10.671	10.544	10.359	10.298	10.098	10.078
	std	0.249	0.012	0.061	0.025	0.015	0.013	0.0012	0.0005
QNX-P166	average	10.876	10.397	10.188	10.144	10.082	10.063	10.006	10.004
	maximum	12.738	11.207	10.483	10.348	10.188	10.121	10.011	10.009
	minimum	10.755	10.362	10.177	10.139	10.077	10.061	10.005	10.0002
	std	0.063	0.055	0.008	0.008	0.004	0.004	0.004	0.0002
RTL-P166	average	12.695	11.223	10.544	10.408	10.251	10.197	10.032	10.018
	maximum	18.523	13.685	11.779	11.373	11.766	10.588	10.061	10.046
	minimum	11.901	10.889	10.443	10.337	10.212	10.169	10.030	10.015
	std	0.249	0.012	0.061	0.025	0.015	0.013	0.0012	0.0005
OSEK-MPC	average	23.755	14.156	11.782	11.398	10.871	10.705	10.149	10.097
	maximum	23.804	14.166	11.812	11.411	10.895	10.723	10.150	10.129
	minimum	23.689	14.108	11.754	11.353	10.836	10.664	10.146	10.090
	std	0.0203	0.0217	0.0285	0.0236	0.0282	0.0261	0.0069	0.0173

Table 6.9: Measured execution times with different clock resolutions.

Given the measured execution times, the clock overhead at each resolution can be computed by solving Eq. (6.3) and (6.4). In the clock overhead calculation, we used the minimum execution time when the resolution was set to 100 ms as e_0 for each case. The computed clock overheads for each test case are shown in Figure 6.2, 6.3 and 6.4.

The measurement results show that the clock overheads tend to decrease in general as the du-

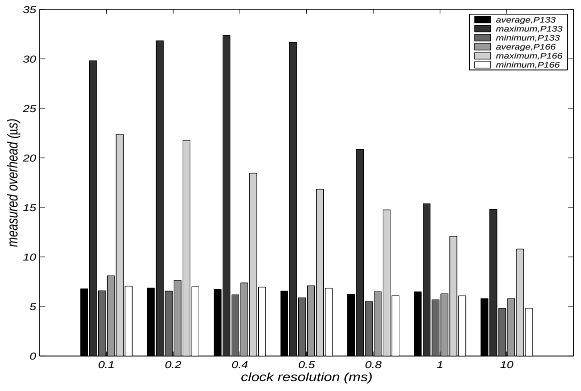


Figure 6.2: Timing service overheads of QNX.

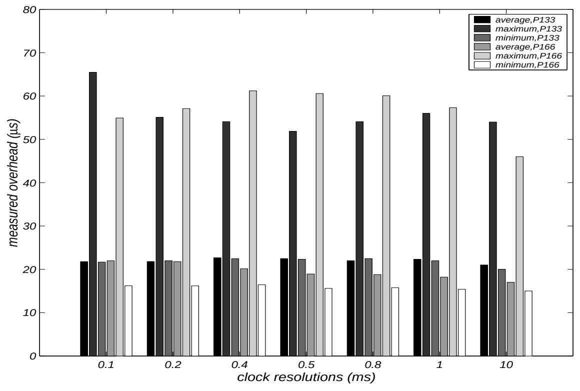


Figure 6.3: Timing service overheads of RTLinux.

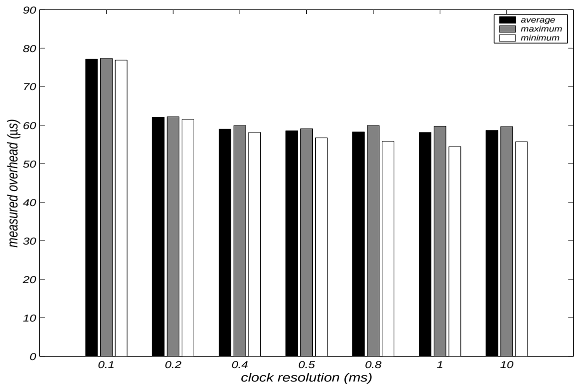


Figure 6.4: Timing service overheads of OSEKWorks.

ration between clock ticks increases. This indicates that a fine-resolution clock will consume more system resource and may cause schedulability problems although such a clock may make the system react swiftly. The quantitative effects of clock resolutions are OS-dependent. For QNX, the normal overheads are around $5 \sim 7\mu s$. But the maximum can be around $30\mu s$. RTLinux overheads are around $20\mu s$ though the maximum can be around $60\mu s$. The overheads for OSEKWorks are around $60\mu s$ except a higher overhead of $70\mu s$ is experienced when the clock resolution is set to $0.1ms$. We also experienced the system hang when setting the clock resolution smaller than $50\mu s$ for QNX-P133, QNX-P166, $70\mu s$ for RTLinux-P133 and RTLinux-P166, and $80\mu s$ for OSEKWorks-MPC, although the measured overheads for corresponding systems are much less than values that would make the system halt. This may be caused by some interrupt and timing related activities in the kernel.

The measurement results also show that a faster processor can generally reduce the clock overheads. As can be seen in Figure 6.2, using a faster processor reduces the maximum overheads although it does not improve the average and minimum cases. As for the results of RTLinux shown in Figure 6.3, both average and minimum overheads were reduced on the P166 platform. Further, comparing the overheads of QNX and RTLinux on both platforms with those of OSEKWorks-MPC, the clock overheads of OSEKWorks were almost constant for any given resolution, while the maximum overheads for both QNX and RTLinux were significantly larger than average and minimum for any given resolution. Less variant overheads for a given OSEKWorks can result from the simple functionality of OSEKWorks [93] flat memory structure of MPC555 [61]. Both help to reduce unpredictability during the executions.

6.3.2 Results of interval jitter measurement

We then measured the effects of clock resolutions, as well as the number and patterns of timers, and task priorities on interval jitters that a given task will experience. For clock resolutions, we are interested in how different clock resolutions effect the jitters of different interval lengths. The measured results of interval jitters for QNX and RTLinux under different clock resolutions are shown in Figure 6.5 and 6.6. For OSEKWorks, we did not observe any jitter for examined clock resolutions.

According to the measured results, interval jitters varied greatly from one OS to another. In QNX, interval jitters increased as the clock resolution became lower, while the jitters for RTLinux showed only slight change with different resolutions. The reason for this could be that QNX uses a clock-based scheduler while RTLinux and OSEKWorks use event-based schedulers. Predictable hardware architecture also contributed to the absence of jitters across all experiments in OSEKWorks.

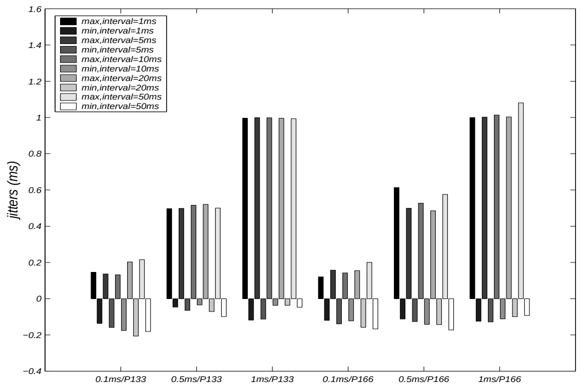


Figure 6.5: Interval jitters of QNX.

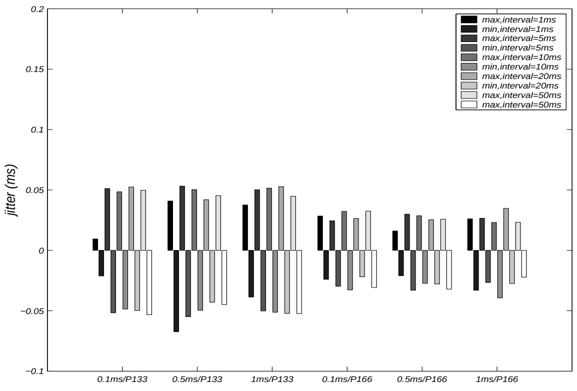


Figure 6.6: Interval jitters of RTLinux.

Moreover, the results showed that interval jitters are independent of the length of an interval. This is different from the traditional understanding that the interval should be some relatively large multiples of resolution to overcome jitters. Even for QNX with a clock-based scheduler, the interval jitters of 1 ms interval with 1 ms clock resolution were not different from jitters of $10ms$, $20ms$, and $50ms$ intervals. The resolution further determined the size of interval jitters. As can be seen in Figure 6.5, the jitters were always bounded with twice the clock resolution. Figure 6.5 and 6.6 also indicate that a faster processor did not help to reduce jitters for an OS using a clock-based scheduler, but did reduce jitters for an OS using event-based scheduler.

Next, we studied the effects of the number of timers and interval patterns on jitters. The measurements also included the jitters experienced by tasks with different priorities. Figure 6.7 and 6.8 show the results of harmonic and non-harmonic intervals on QNX, respectively. Figure 6.9 and 6.10 show the results of the same experiments on RTLinux, and Figure 6.11 and 6.12 show the results of OSEKWorks.

According to these results, we first observed that the interval jitters increased in proportion to the number of timers in the system for all measured cases. Such dependencies should be an OS property, independent of hardware. Both in the case of the same OS running on different hardware and of different OSs running on the same hardware the same tendency of interval jitter changes

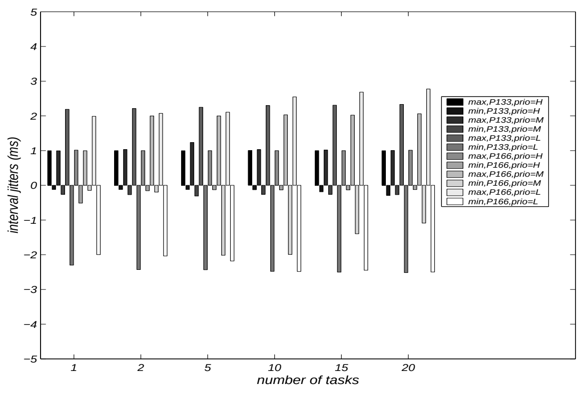


Figure 6.7: Interval jitters for harmonic intervals on QNX.

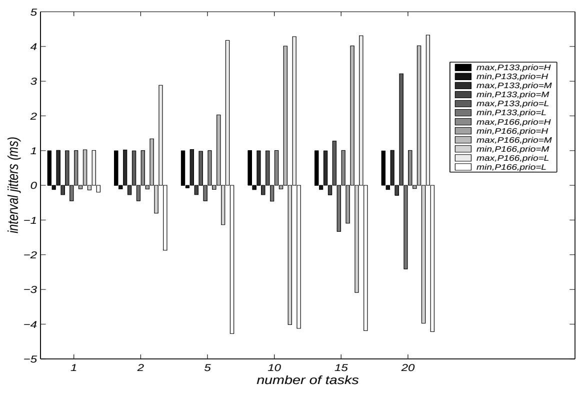


Figure 6.8: Interval jitters for non-harmonic intervals on QNX.

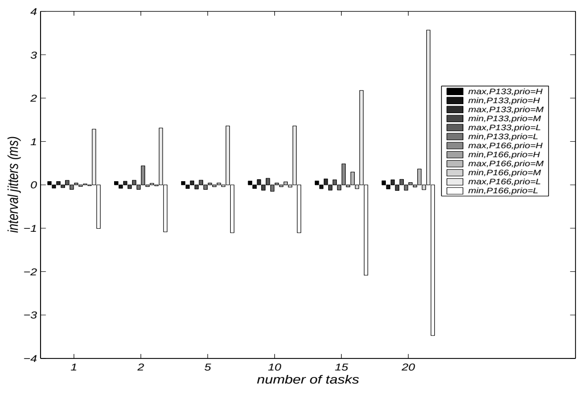


Figure 6.9: Interval jitters for harmonic intervals on RTLinux.

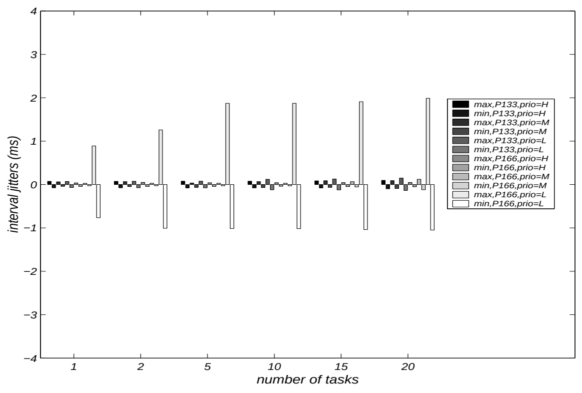


Figure 6.10: Interval jitters for non-harmonic intervals on RTLinux.

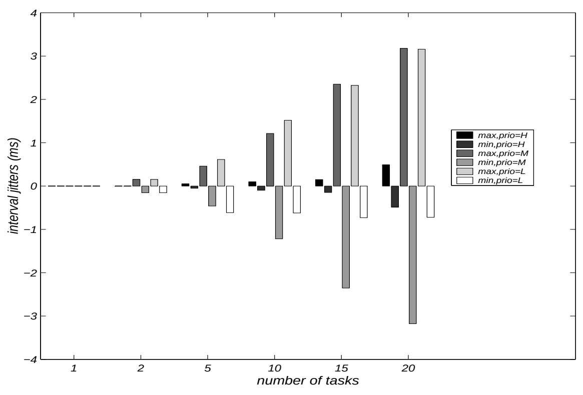


Figure 6.11: Interval jitters for harmonic intervals on OSEKWorks.

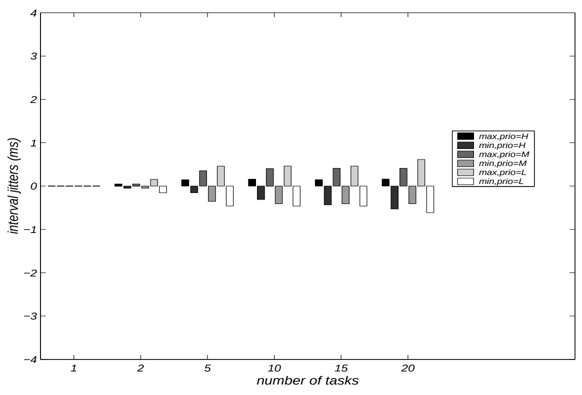


Figure 6.12: Interval jitters for non-harmonic intervals on OSEKWorks.

showed up, and using a faster processor did not reduce the jitter. These results suggest that reducing the number of timers by combining tasks with the same intervals would reduce the interval jitters, and consequently improve system performance.

Interval jitters experienced by tasks with different priorities were also significantly different. A higher priority task experienced smaller jitters, while a lower priority task experienced larger ones in all our measurements. Jitter differences for tasks with different priorities are independent of the number of timers and interval patterns, but depend on the type of RTOS. Larger jitters experienced by a lower priority task could be the cumulative effects of kernel activities that have a lesser effect on higher priority tasks.

The interval patterns had significant impacts on interval jitters too. These impacts depended on the implementation strategy of OS structure [43] and the number of timers in the system. Among the measured cases, jitters were almost the same for both harmonic and non-harmonic intervals when there were a relatively small number of timers (≤ 5). When the number of timers became larger, the jitters with non-harmonic intervals became larger than those with harmonic intervals for QNX, but showed the reverse for RTLinux and OSEKWorks. This can be explained by the clock-based and event-based OS implementation.

The "memory effects" of timer intervals found in our previous research [100] were also observed in this measurement. However, the effects showed differently in the way that some longer intervals were followed by a sequence of compensatory shorter intervals or vise versa instead of a longer interval followed immediately by a compensatory shorter interval in our previous observations.

We also observed that larger jitters were present in experiments on the P166 board with a faster processor for both QNX and RTLinux, which goes against the common view that a faster processor should yield a better performance. Our guess is this could be either the hardware itself (e.g., different manufacture of some chips and boards) or the implementation of RTOS (e.g., different rates of some

OS internal activities on different processors). The true reason may require further investigation.

6.3.3 Results of context switch measurement

The measurements of context switches were performed under system configurations with different numbers of tasks. To learn the relationships between clock resolutions and context switch times, we performed the measurements under different clock resolutions too, specifically $0.5ms$ and $1ms$, as the OS scheduler can be either clock-based or event-based.

The measured context switch times of QNX, RTLinux, and OSEKWorks are shown in Figure 6.13, 6.14, and 6.15 respectively. For QNX, the average and minimum context switch times were not sensitive to the number of tasks in the ready queue, but the maximum context switch time increased when the number of tasks in the ready queue increased under a finer clock resolution. Such a difference can be as high as 3 times between a system with 20 interference tasks and a system without any interference task. For RTLinux, both average and maximum context switch times increased as the number of tasks in the ready queue increased under any clock resolution, while the minimum times stayed the same. Both average and maximum context switch times of a system with 20 interference tasks can be twice as high as those for a system with no interface task. On the other hand, the context switch times for OSEKWorks showed hardly any variance. Such results imply that the context switch time heavily depends on the RTOS implementation, and at least will not increase if the number of tasks is reduced.

We also noticed that the clock resolution had significant impacts on context switch times. The context switch times (average, maximum and minimum) with a small clock resolution were higher than those with a larger resolution for QNX, while it was the opposite for RTLinux and OSEKWorks. This can be explained by the fact that the context switch times of a clock-based scheduler may be more sensitive to clock overheads, while the event-based scheduler may be more sensitive to the the resolution itself.

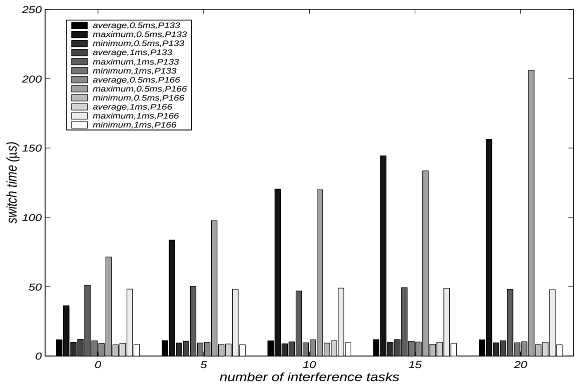


Figure 6.13: Measured context switch times for QNX.

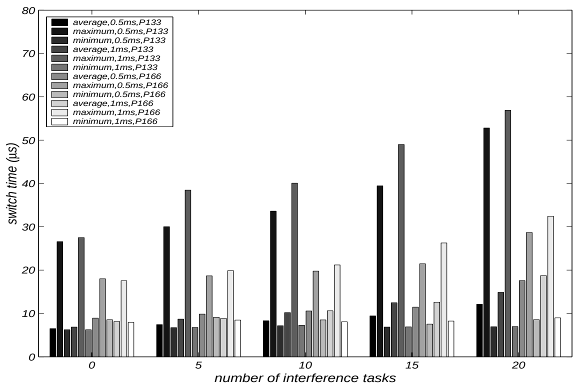


Figure 6.14: Measured context switch times for RTLinux.

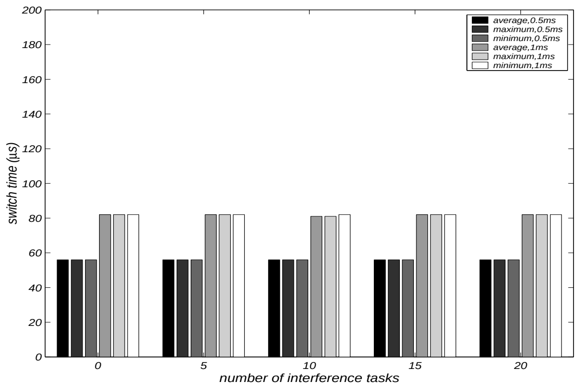


Figure 6.15: Measured context switch times for OSEKWorks.

6.4 Related Work

Recently, many models have been proposed for OS and integrated ECSW performance analysis, including hierarchical models of OS services [12] and performance models of the RTOS scheduler based on both the OS structure and the measured performance of the scheduler component [43]. Other performance models [11, 4] or integrated systems have also been developed, including a generic performance analysis framework [11], compositional performance modeling based on stochastic process algebra [32], model composition based on dependency graphs [1], and UML-based models [33, 4]. All these models suggested a modeling hierarchy with the OS performance model as a separate layer between hardware and the application, and required the measurement of OS services for model construction and performance analysis.

There have also been numerous general techniques for measuring the different aspects of computer systems [37, 56], and benchmarks for OS-level measurements, including workload design, system monitoring, measurement strategies and tools, and result presentations. These techniques must be carefully tailored to meet the need for measuring the OS service performance. Most current OS-level measurements are based on benchmarks, such as Rhealstone [41], Hartstone [91], *lmbench* [59], and *hbench-OS* [12]. Although these techniques are useful in measuring the performance of individual operations and determining performance bottlenecks, the effects of application structures and interactions—which may make significant performance differences of ECSW—are ignored. The information measured with these benchmarks is limited and inaccurate for ECSW performance analysis.

Measurements that include the effects of applications can be made with a synthetic workload. Methods of generating synthetic workloads include Hartstone [91], a synthetic workload specification language (SWSL) and generator [45], and DynBench [75]. All of these techniques consider application properties but performance metrics and frameworks are not defined for RTOS service measurements. Some *ad hoc* methods have also been used for performance measurements of OS services [72, 22, 88] with domain-specific applications. These measurements have generated RTOS performance information with domain-specific applications, although the measurements are usually restricted to a specific configuration used for the purpose of comparing different products.

Our measurement method focuses on how to obtain information on the performance of various RTOS services under representative application configurations that are necessary for the analysis of ECSW design and integration. This differs from measurements targeted for product comparisons in that (1) the measurements should be as non-intrusive as possible, (2) the measured information should be reusable for a family of applications, and (3) the measurements should encompass all possible configurations of the service in which they can be used. Our previous research on performance measurement [100] has shown that different measurement goals lead to different measured metrics and workloads.

6.5 Summary

Performance analysis has become essential for ECSW construction with reusable components. Such an analysis requires measuring and reusing the performance information of OS-level services. Most existing performance measurement methods consider either a fixed configuration with applications, such as macrobenchmarks and simulations, or ignore how applications will use these services, such as various microbenchmarks. Our proposed end-to-end measurement method is based on both microbenchmarks and synthetic workloads. The measured performance information can be used to construct the OS-level performance model given the application functional design. The measured performance can therefore be stored with OS service components, and can be reused in performance analysis of different applications along with reuse of the OS services and their execution environments. We have applied this method in measuring the timing and scheduling services of selected RTOSs, and have presented several findings of the performance dependencies among the measured services, the service performance metrics, and the application configurations. Such information can be used in both timing analysis and schedulability analysis of integrated ECSW.

CHAPTER 7

AIRES: A Software Toolkit for ECSW Design and Analysis

The techniques of the performance modeling, design model transformation, and performance analysis developed in this dissertation are to facilitate ECSW design and implementation. Since ECSW development is a multi-stage multi-iteration process involving multiple design groups from different disciplines, it is essential to have software design tool support for the designer to interact with the design model construction to finalize the selection of model parameters. The software tool supports the visualization of the modeling and analysis results, and guides the designer through a correct design process. We have, therefore, implemented such a tool, called *AIRES* (Automatic Integration of Real-time Embedded Software), with the performance modeling, estimation, transformation, and analysis techniques built into it.

The significant difference between the AIRES toolkit and current existing ECSW design and analysis tools is that the AIRES toolkit supports not only the performance analysis of the complete runtime model, but also the performance estimation with abstract models and the design model transformation. This fills the gap between the control design and software analysis. The current design tools for control software focus on functional model construction, and leave performance as an issue to be addressed at the implementation phase. Such an approach usually delays detection of performance errors, and hence incurs high development costs and a long development cycle. Furthermore, current analysis tools provide only the results of the given design, and cannot link the results with design decisions made in a design tool. To address any performance errors detected by the analysis tool, the designer must trace their causes in the models and correct them in the design tool. The designer may have to move back and forth between design and analysis tools until all performance constraints are met. Due to the large design space for a typical ECSW system, it is extremely difficult for the designer to decide proper changes that correct the current performance errors without introducing new errors. The AIRES toolkit is designed and implemented to integrate modeling and analysis in the same environment, and to directly feed analysis results back to the design models along with recommendations on how to correct errors. We have also embedded a design process in the AIRES tool to force the designer to follow a correct model construction and analysis procedure with the required modeling information provided at each step. With this built-in design process, the design model can be transformed automatically from one stage to another toward the final executable system.

The implementation of the AIRES tool consists of a graphic modeling environment, a modeling language for ECSW design and performance model construction, and a set of loadable algorithms implementing the performance analysis and performance-aware model transformation.

7.1 Graphic Modeling Environment

The AIRES toolkit uses the Generic Modeling Environment (GME) [34], developed by Vanderbilt University, as its graphic modeling environment. The GME is a configurable modeling tool supporting domain-specific modeling and synthesis. It can be configured with multiple meta-models, each of which defines a modeling paradigm (modeling language) of an application domain. The analysis and model transformation algorithms can be implemented as a loadable module based on MS COM. The model visualization is customizable through decorator interfaces.

The selection of the GME is based on the following requirements of the AIRES tool implementation:

- Modification of the modeling framework. Our developed performance modeling framework requires the annotation of performance parameters to functional model constructs. Given any modeling language used for an application domain, we may have to modify the language so that an application model constructed using the modeling language contains the information needed for the analysis. The customizable meta-model concept supported by the GME allows us to define a modeling language with the desired features.

- Integration of third-party algorithms. Our analysis and transformation algorithms need to interact with the modeling environment to extract the required information. It is essential to choose a modeling environment that provides hooks to allow the algorithms to access the models. The GME provides a set of interfaces, including interpreters, plug-ins, and add-ons, to integrate the analysis algorithms and their required meta-models into the modeling environment.

- Visualization of analysis results. It is essential for designers to visually verify the results generated by the analysis algorithm. This requires the modeling environment to support visualization of the analysis results. The analysis results, in particular, should be associated with the design modeling parameters so the designer can quickly learn the effects of a modeling parameter change. The GME provides interfaces for model manipulations so that the results of the analysis algorithm can be fed back into the modeling environment.

The architecture of GME is shown in Figure 7.1. All domain-specific meta-models are derived from the GME core meta-model. An application model is then constructed using the corresponding domain-specific meta-model. The model construction and visualization are supported by the GME editor, visualization mechanism (through decorator), and model browser. The analysis algorithms can be integrated into the modeling environment as interpreters, plug-ins, and add-ons. These mechanisms provide interfaces for the algorithms to extract the models in the environment and feed back the results to the modeling environment for visualization.

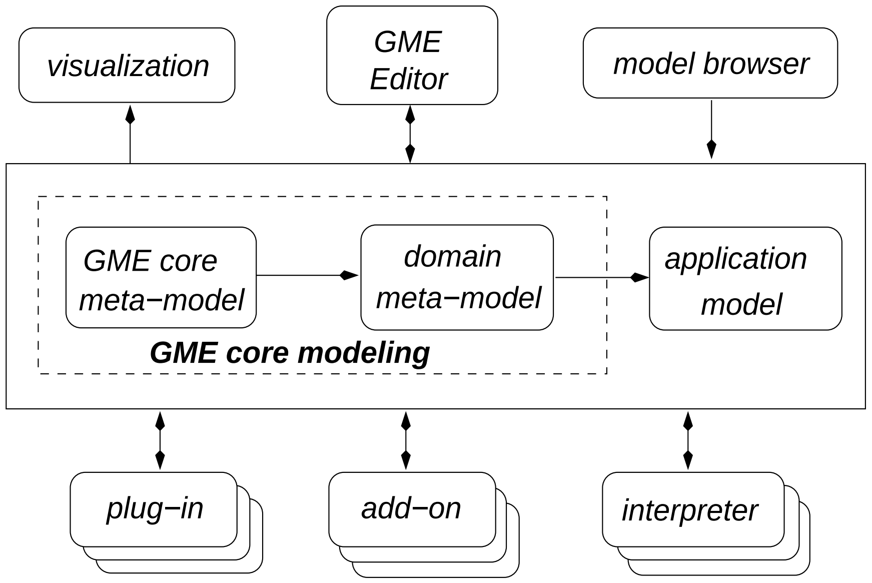


Figure 7.1: GME architecture and components.

7.2 AIRES Modeling Language

All AIRES models and analysis require the AIRES meta-model. The AIRES meta-model implements the modeling framework discussed in Chapter 2, which captures all required information for ECSW modeling and analysis. It defines four types of models used in different design stages: *component model*, *platform model*, *structural model*, and *runtime model*. Each type of models is stored in a separate model repository, called a *folder*.

The *component folder* stores the software component models with the structure defined in Chapter 2. They can be reused for application model construction in multiple applications designs. As shown in Figure 7.2, each component contains a set of actions and ports. In the AIRES meta-model, we distinguish component event ports from data ports to model different types of communications. Only ports of the same type can be connected. Communications through event ports must be synchronous, while the communications through data ports can be asynchronous. Knowing the communication types is necessary for performance estimation and analysis. Some performance parameters are annotated to both the component and its constituent actions as shown in Figure 7.2.

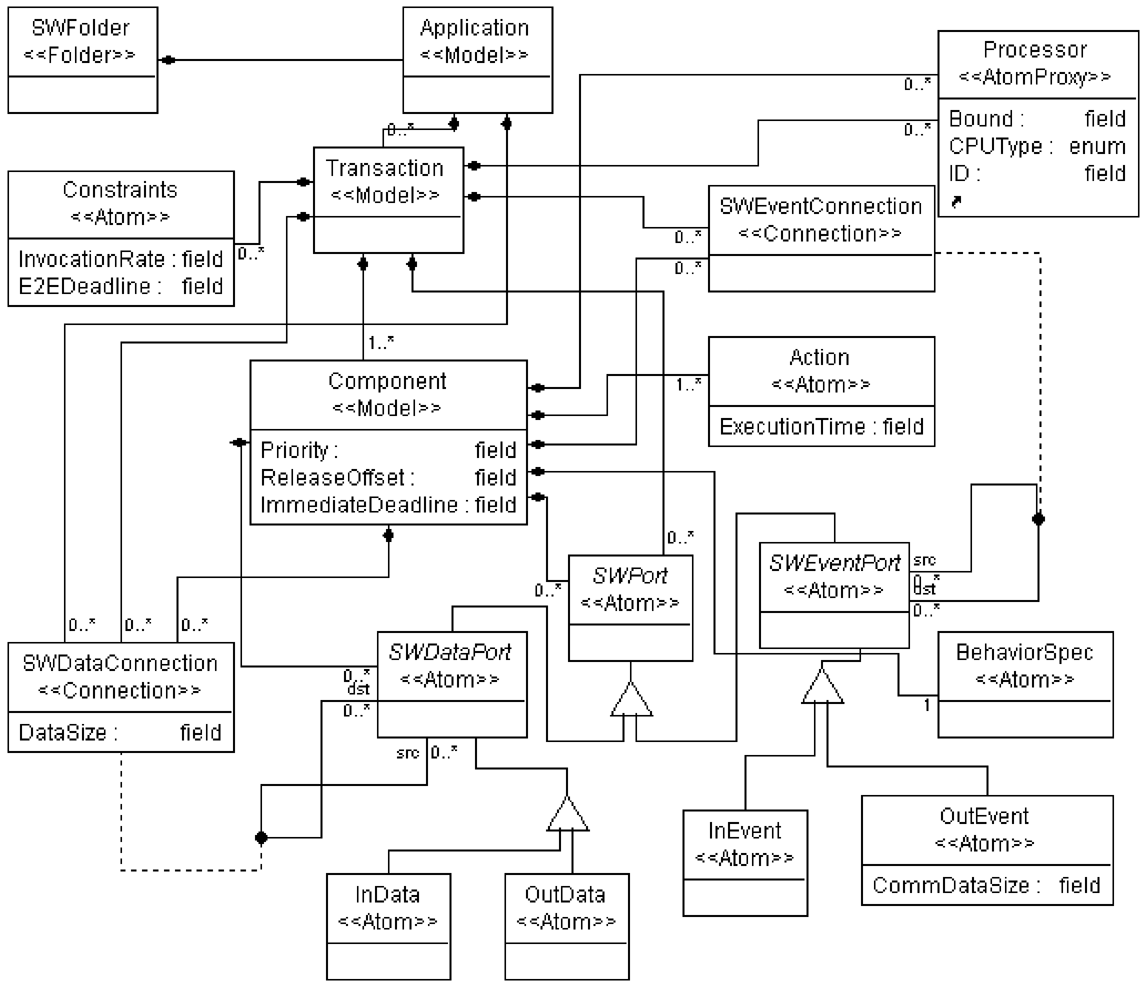


Figure 7.2: Meta-model for component modeling and structure modeling.

Figure 7.2 also includes the meta-model of software structural models. A structural model is the design model of a target application. It is stored in the *software folder*. The structural model contains a 3-level hierarchy. The top-level is an *application* model that contains multiple *transactions*. Each transaction contains a set of interconnected low-level *components*. A transaction is represented as a directed graph consisting of components and event connections (SWEventConnection). An event connection links an input event port (InEvent) and an output event port (OutEvent) of different components. Using event connections implies the component communications in a transaction must be synchronous. Because the message that arrives at an input port must be the same message generated by the connected output port over an event connection, we only need to specify the message size for one port—for example, the OutEvent—to avoid modeling inconsistency. A transaction starts from some input component(s) and ends at some output component(s). In an application model, different transactions can communicate through components' data ports. These communications are assumed to take place asynchronously. System-level constraints, such as end-to-end deadlines and invocation rates, are imposed on the transactions.

To construct a runtime model from a structural model, one must know the target platform configuration. As shown in Figure 7.3, the AIRES meta-model defines hardware components such as processors and communication links, and supporting software such as operating systems for platform design and modeling. The attributes of the hardware components are defined as resource capacity bounds, which vary along with the hardware (for example, clock speed, instruction set, and cache/memory size of a processor). The performance of the system software is defined as a set of overheads varying along with the selection of service parameters (for example, OS type and timer resolution). It is the platform designer's responsibility to construct a platform that provides sufficient resource for the design ECSW. The platform construction can be guided by the structural model performance estimations in order to meet other system constraints. In the AIRES tool, the constructed platform model is stored in the *platform folder*.

The AIRES tool models the runtime software as a task graph. Tasks are also modeled as port-based objects with the performance parameters including execution locations, invocation periods, priorities, release offsets, and deadlines. The transactions in the structural model are represented as task chains in the runtime model. A runtime model is also a hierarchical model, in which the

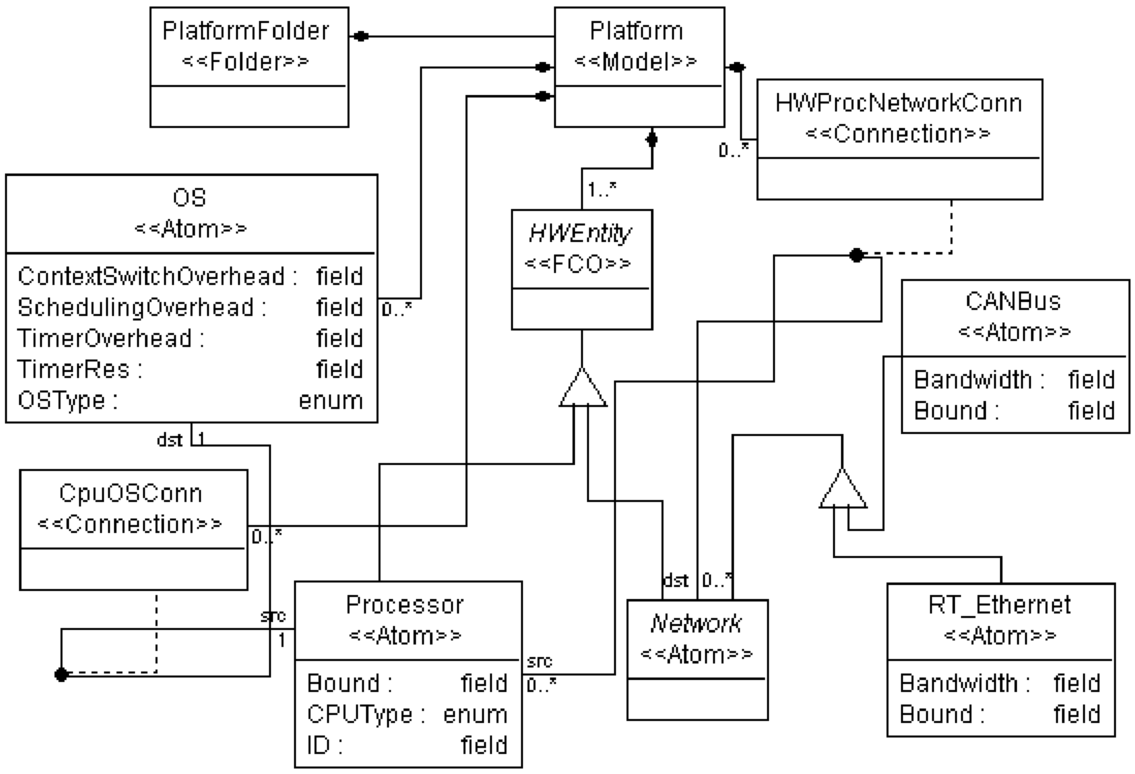


Figure 7.3: Meta-model for platform modeling.

top-level consists of *TaskChains*, each of which contains a set of communicating *Tasks*. Each task contains a set of sequentially executing *Components*. Figure 7.4 shows the AIRES meta-model for the runtime model. The construction of a runtime model with a given structural and platform model can be done either manually by the designer following the AIRES analysis results, or automatically by the AIRES tool with some predefined transformation policies (such as workload distribution policy). The constructed runtime model is stored in the *runtime folder*.

7.3 Built-in Design Process

The AIRES tool implements a built-in design process, as shown in Figure 7.5, which forces the designer to follow when constructing and analyzing a model. The process aims to avoid design errors caused by incomplete specifications in a model. It consists of a pre-defined sequence of design

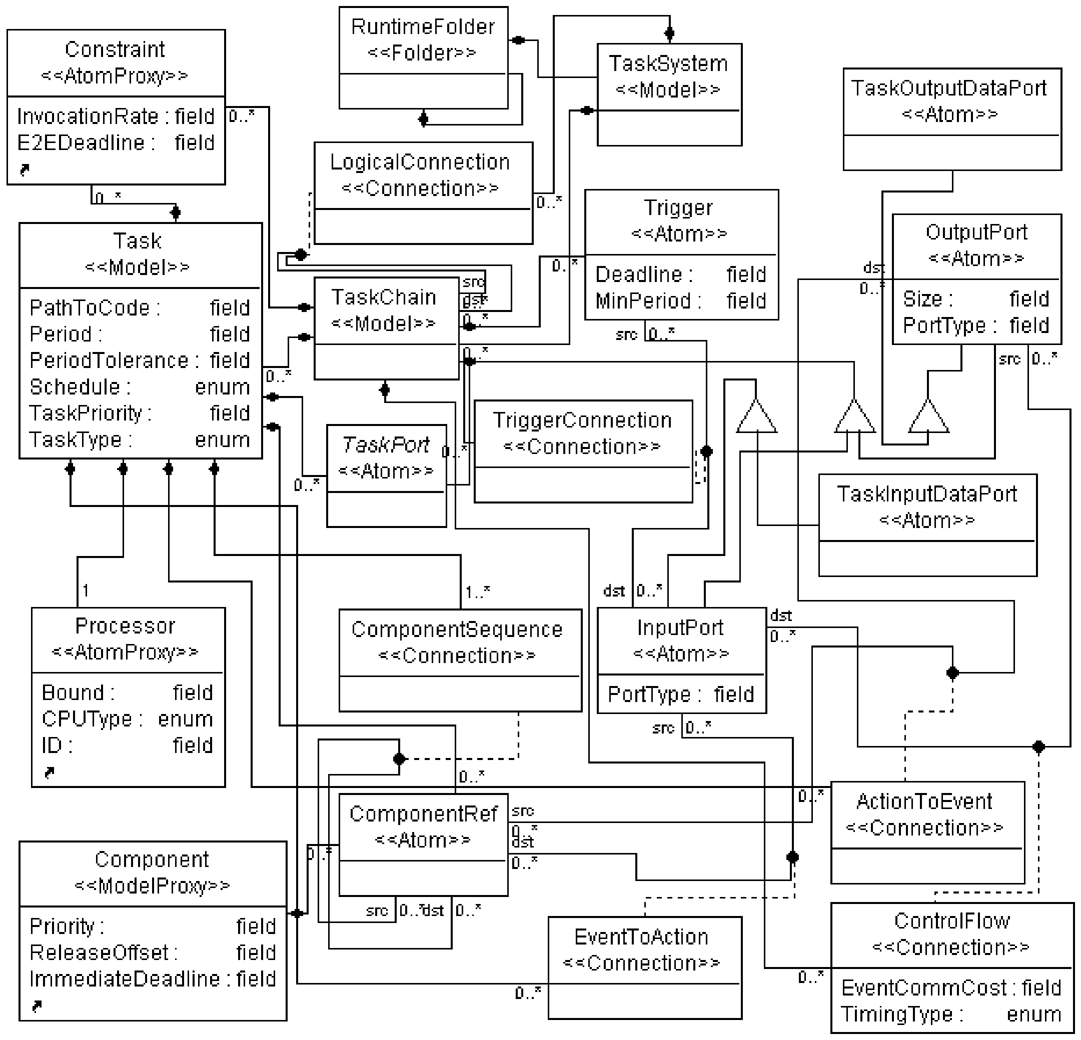


Figure 7.4: Meta-model for runtime system modeling.

steps in constructing and refining software models. At each step, the AIRES tool ensures that the information required for further modeling and analysis is provided by the designer or derived from other modeling information before moving to the next design step.

Since the AIRES tool focuses on software design and implementation, the process starts from component model construction and ends after runtime model analysis. Component model construction is performed after control discretization, which is the process of selecting and connecting software component models to implement the designed functionalities. The previous steps focus only on controllability and control performance, and are irrelevant to the software design. Similarly,

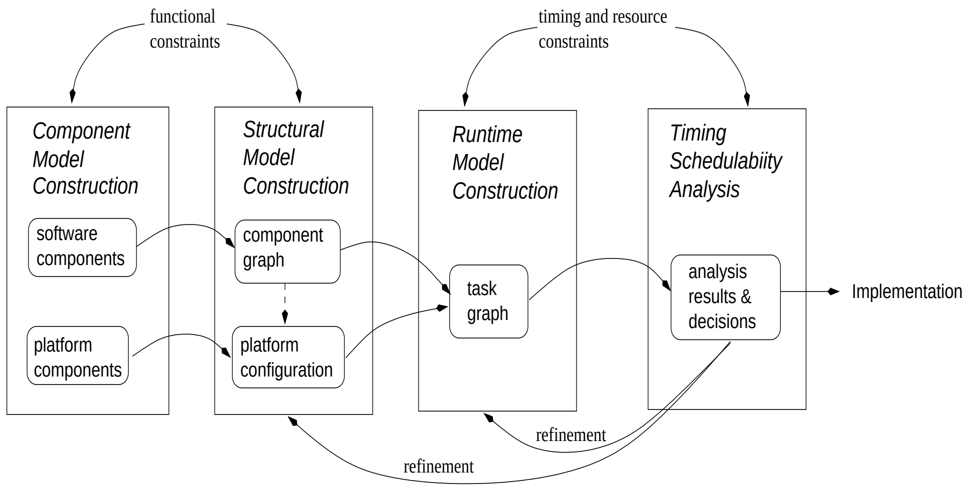

functional constraints
timing and resource constraints
Component Model Construction
Structural Model Construction
Runtime Model Construction
Timing Schedulabiity Analysis
software components
platform components
component graph
platform configuration
task graph
analysis results & decisions
Implementation
refinement
refinement

Figure 7.5: AIRES build-in design process.

the steps after runtime analysis deal with implementations, and can be done with the assistance of automatic code generation tools.

The design process may require multiple iterations to obtain a correct system configuration that meets both functional and performance constraints. To simplify the design process, our tool addresses the functional constraints, first by constructing a structural model. It then addresses the timing and resource constraints by model transformation. The functional model construction can be done either manually by the designer or automatically through synthesis using some design automation tool. The transformation that meets timing and resource constraints can be done automatically in the AIRES tool, or interactively between the AIRES and the designer. The result of the AIRES tool is a final implementable system with the specified functional, timing, and resource constraints met.

The AIRES tool can also be used with other system design, analysis, and generation tools to form an integrated design tool chain. The functionalities implemented in the AIRES can be used individually as long as the modeling language used in the tool chain is compatible with the AIRES meta-model. For example, the AIRES runtime model construction can be used as a model transformation tool to generate a runtime model, while a commercial real-time analysis tool, such as

RapidRMA [89] or Timewiz [85], can be used to analyze the performance of the generated runtime model.

7.4 Algorithm Implementation

The GME provides three interfaces to integrate customizable algorithms: *add-ons*, *plug-ins*, and *interpreters*. The *add-ons* and *plug-ins* are based on the OCL (Object Constraint Language), and are suitable for simple constraint checks. Our implementations use the interpreter mechanism. The interpreters are built as dynamic link libraries (DLL) that can perform complex operations. Specifically, our interpreters are built with the GME Builder Object Network (BON) interfaces. Upon its invocation, the interpreter creates a BON structure that contains a mirror object for every modeling element in the model. Other operations in the interpreter can then access the full model through the BON structure and its methods.

The AIRES interpreters include the following algorithms:

- Component allocation that assigns the components to the platform for execution;
- Runtime model construction that determines the timing attributes for components, group the components with the same execution location to form tasks, and assigns the execution attributes of the tasks;
- Schedulability analysis that performs schedulability, end-to-end timing, and resource consumption analysis.

The component allocation algorithm assigns the components in the structural model to a computation device in the platform. The corresponding communications across devices are assigned to the links between the devices. The designer needs to specify the allocation strategy for this algorithm. The implemented strategies include first-fit and load-balance for computation resources, and minimal communication for communication resources. The tool also allows the designer to choose certain combinations of these strategies. The algorithm is implemented with heuristics that take into account the computation, communication, and memory resources, as described in Chapter 4. For the purpose of comparison, we have implemented an exhaustive search algorithm that can explore the whole design space for a small-scale design. This helps us understand the optimality of the generated component allocation results.

The runtime generation algorithm described in Chapter 4 is also implemented. It forms a task graph of the system and determines the timing attributes for all tasks. The generated model is stored in the *task folder* for runtime schedulability analysis.

The schedulability analysis algorithm can provide results on task priority assignments, schedulability and end-to-end response times, consumptions of resources and individual activities, and system overheads. The schedulability analysis requires the designer's input on the scheduling policy. Specifically, for the priority assignment, we implemented the traditional rate-monotonic assignment, the deadline-monotonic assignment, and the heuristic assignment as described in Chapter 5. The heuristic assignment is also used for the automatic system refinement. To account for the system overheads in the analysis, we built functions to derive the system overheads based on the platform configuration. These functions are constructed using the RTOS measurement results.

7.5 Model Transformation and Analysis of ETC

We present the use of the AIRES tool for model transformation and analysis of an automotive electronic throttle control (ETC) design. The example was provided by the PATH group at UC Berkeley and Ford Motor Company in the MoBIES program, and was in the form of Simulink/Stateflow block diagrams with only control-related parameters. Our design assumes that the discretization has been done before the software design starts, including the system modeling granularity, sampling periods, and control steps. The application of the AIRES tool starts from the structural model with the end-to-end constraints.

Figure 7.6 shows the models in the GME environment. With AIRES, the system models, including component models, platform configuration, and runtime models, can be constructed or generated in the workspace. The right-hand browser lists the folders for modeling and analysis in AIRES. The components' icons on the right end of the tool bar are interpreters implementing the AIRES algorithms.

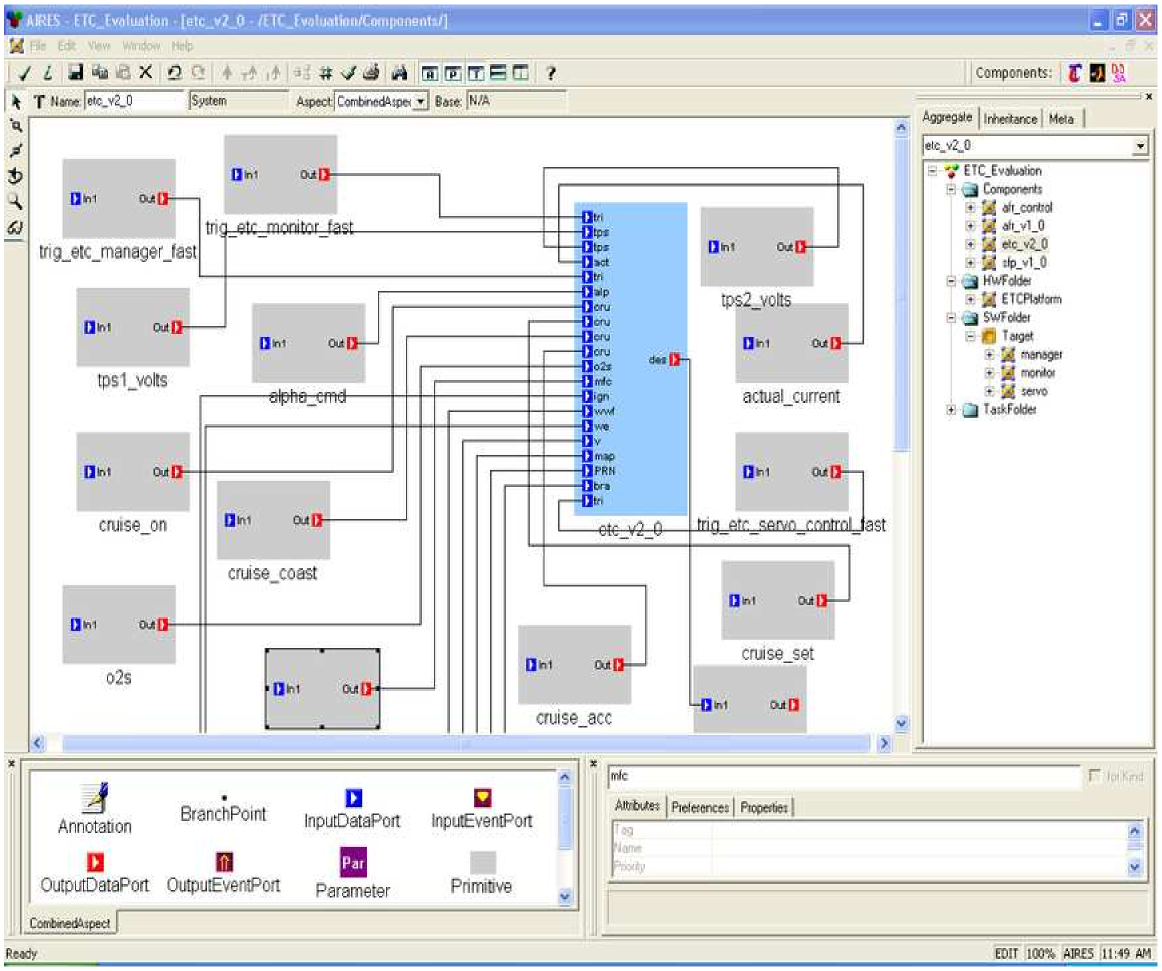
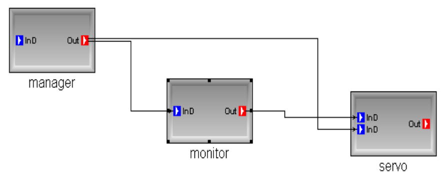


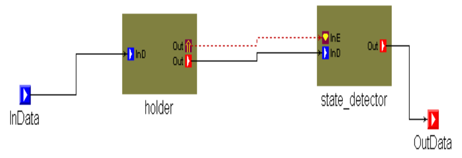
Figure 7.6: Models in GME environment.

7.5.1 Software structural model and platform model

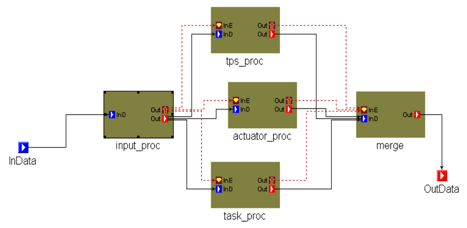
The ETC structural model contains three communicated transactions: *manager*, *monitor*, and *servo*. Each of these transactions contains a sequence of components. Figure 7.7 shows the ETC structural model with all its transactions. The performance parameters of the model are given in Table 7.1 and 7.2. In this example, we assume the end-to-end deadline of each transaction equals its invocation period.



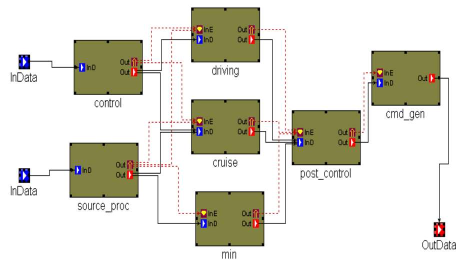
(a) ETC structural model



(b) Transaction: manager



(c) Transaction: monitor



(d) Transaction: servo

Figure 7.7: Models of manager, monitor, and servo transaction in ETC.

The platform for the ETC software consists of two MPC 555-based micro-controllers, connected by a CAN-bus, as shown in Figure 7.8. Both controllers are running with OSEKWorks (OS_1 and OS_2). The first controller P_1 is assigned with a resource bound 0.8 and $ID_{CAN} = 0$, while the second controller P_2 with a resource bound 0.7 and $ID_{CAN} = 1$.

transaction	component	parameter	type	value (ms)
manager		invocation period	constraint	10
	holder	wcet	resource demand	0.03
	state_detector	wcet	resource demand	0.5
monitor		invocation period	constraint	40
	input_proc	wcet	resource demand	0.3
	tps_proc	wcet	resource demand	0.5
	actuator_proc	wcet	resource demand	0.1
	task_proc	wcet	resource demand	0.3
	merge	wcet	resource demand	0.1
servo		invocation period	constraint	3
	control	wcet	resource demand	0.1
	source_proc	wcet	resource demand	0.3
	driving	wcet	resource demand	0.2
	cruise	wcet	resource demand	0.1
	min	wcet	resource demand	0.1
	post_control	wcet	resource demand	0.4
	cmd_gen	wcet	resource demand	0.2

Table 7.1: Modeling parameters of ETC components.

source	destination	data size (byte)
manager.holder	manager.state_detector	4
manager.state_detector	monitor.input_proc	8
monitor.input_proc	monitor.tps_proc	5
monitor.input_proc	monitor.actuator_proc	3
monitor.input_proc	monitor.task_proc	3
monitor.tps_proc	monitor.merge	4
monitor.actuator_proc	monitor.merge	4
monitor.task_proc	monitor.merge	4
manager.state_detector	servo.control	8
monitor.merge	servo.source_proc	4
servo.control	servo.driving	4
servo.control	servo.cruise	4
servo.source_proc	servo.cruise	2
servo.source_proc	servo.min	2
servo.driving	servo.post_control	2
servo.cruise	servo.post_control	2
servo.min	servo.post_control	2
servo.post_control	servo.cmd_gen	8

Table 7.2: Modeling parameters of ETC links.

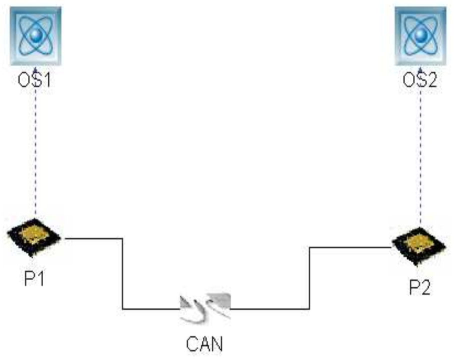
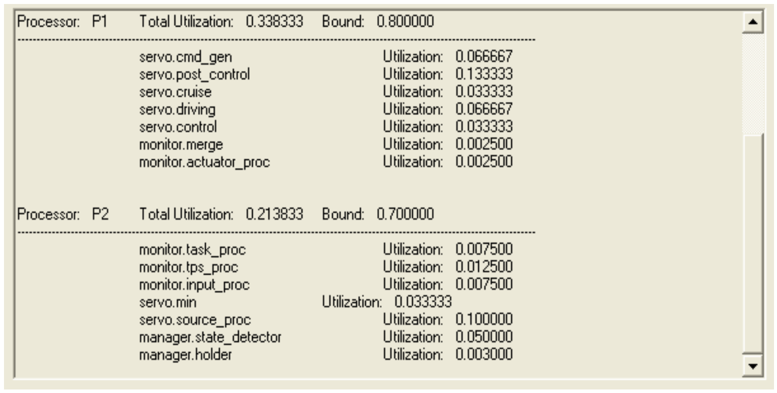


Figure 7.8: Platform configuration of ETC.



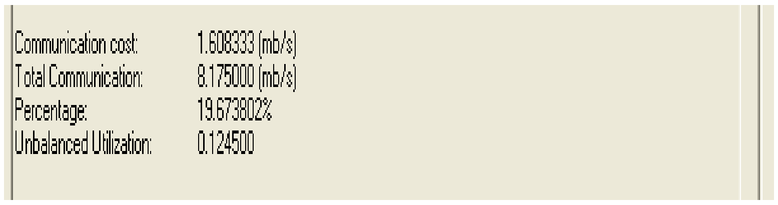
```
Processor:  P1      Total Utilization:   0.338333    Bound:   0.800000
--------------------------------------------------------------------------------
                    servo.cmd_gen                          Utilization:   0.066667
                    servo.post_control                     Utilization:   0.133333
                    servo.cruise                           Utilization:   0.033333
                    servo.driving                          Utilization:   0.066667
                    servo.control                          Utilization:   0.033333
                    monitor.merge                          Utilization:   0.002500
                    monitor.actuator_proc                  Utilization:   0.002500

Processor:  P2      Total Utilization:   0.213833    Bound:   0.700000
--------------------------------------------------------------------------------
                    monitor.task_proc                      Utilization:   0.007500
                    monitor.tps_proc                       Utilization:   0.012500
                    monitor.input_proc                     Utilization:   0.007500
                    servo.min                    Utilization:   0.033333
                    servo.source_proc                      Utilization:   0.100000
                    manager.state_detector                 Utilization:   0.050000
                    manager.holder                         Utilization:   0.003000
```

Figure 7.9: Allocation of the components of the ETC software.

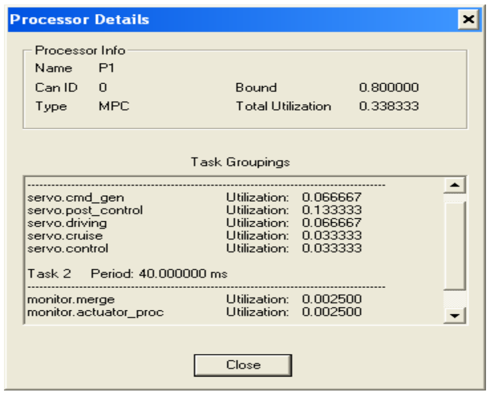
7.5.2 Component allocation and runtime model generation

The component allocation and runtime model generation was automatically done by the component-to-task mapping interpreter. The interpreter first translated the structural model with the three transactions into a partition graph that defines the component allocation. The interpreter had several allocation policies built into it, including the first-fit, load-balance, communication minimization, multiple constraints with any computation, communication, memory, and the combination of computation and communication. In this example, we have chosen the combination of the communication minimization with the load-balance allocation strategy. The allocation of the ETC structural model is given in Figure 7.9.



```
Communication cost:      1.608333 (mb/s)
Total Communication:     8.175000 (mb/s)
Percentage:              19.673802%
Unbalanced Utilization:  0.124500
```

Figure 7.10: Summary of allocation results.

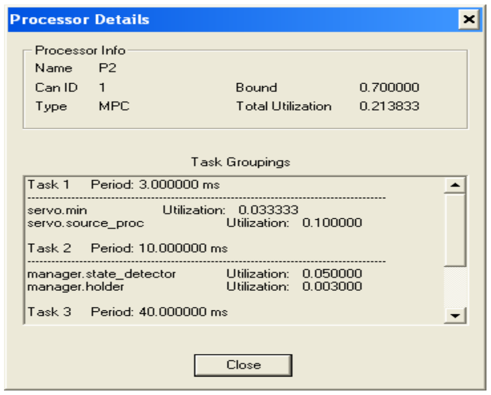


```
Processor Info
Name      P1
Can ID    0                    Bound                0.800000
Type      MPC                  Total Utilization    0.338333

                    Task Groupings
-----------------------------------------------------------------
servo.cmd_gen                  Utilization:   0.066667
servo.post_control             Utilization:   0.133333
servo.driving                  Utilization:   0.066667
servo.cruise                   Utilization:   0.033333
servo.control                  Utilization:   0.033333

Task 2    Period: 40.000000 ms
-----------------------------------------------------------------
monitor.merge                  Utilization:   0.002500
monitor.actuator_proc          Utilization:   0.002500
```

(a) Tasks on processor P_1.



```
Processor Info
Name      P2
Can ID    1                    Bound                0.700000
Type      MPC                  Total Utilization    0.213833

                    Task Groupings
Task 1    Period: 3.000000 ms
-----------------------------------------------------------------
servo.min                 Utilization:   0.033333
servo.source_proc              Utilization:   0.100000

Task 2    Period: 10.000000 ms
-----------------------------------------------------------------
manager.state_detector         Utilization:   0.050000
manager.holder                 Utilization:   0.003000

Task 3    Period: 40.000000 ms
```

(b) Tasks on processor P_2.

Figure 7.11: Generated tasks.

In this allocation, the algorithm uses the minimum communication after the allocation as its primary objective. Among multiple allocations with the same minimum communication, the algorithm chooses the one leading to balanced workloads on the two processors. The algorithm also gives the total communications of the resultant allocation, and the utilization difference between the two processors as unbalanced utilization, as shown in Figure 7.10.

The tasks on each processor are generated in the way discussed in Chapter 4. The generated tasks on each processor, with the components in each task, are shown in Figure 7.11.

As can be seen from Figure 7.11(a), P_1 contains two tasks with one for the servo control transaction and the other for the monitor transaction:

$Task1 : servo.cmd\_gen, servo.post\_control, servo.driving, servo.cruise, servo.control$

$Task2 : monitor.merge, monitor.actuator\_proc$

Similarly, P_2 contains three tasks for the transaction of manager, monitor, and servo. The components in these tasks are:

$Task1 : servo.min, servo.source\_proc$

$Task2 : manager.state\_detector, manager.holder$

$Task3 : monitor.tps\_proc, monitor.task\_proc, monitor.input\_proc$

Despite the harmonic rates of the components of transaction manager and monitor, the two tasks on P_2 cannot be merged because the total execution time of the monitor's components is not large enough to fill the gaps after rolling out the execution of the manager task.

7.5.3 Runtime analysis

After the runtime model of the ETC software is generated (with the timing attributes of tasks assigned), we can perform the performance analysis on the model stored in the *runtime folder*. Figure 7.12 shows the generated runtime model.

The analysis is done by invoking the runtime analysis interpreter. It first assigns the priorities to the tasks and the components in each task. The components' priorities in a task inherit the task priority level and are assigned according to their precedent constraints. Multiple priority assignment methods are implemented in the runtime analysis interpreter. In addition to the one discussed in Chapter 5, traditional priority assignment algorithms, including the rate-monotonic, deadline-monotonic, and ad hoc assignment, are implemented.

The analysis results are shown in Figure 7.13. The results include the parameters derived from the structural model such as invocation periods, deadlines, and worst-execution times, as well as

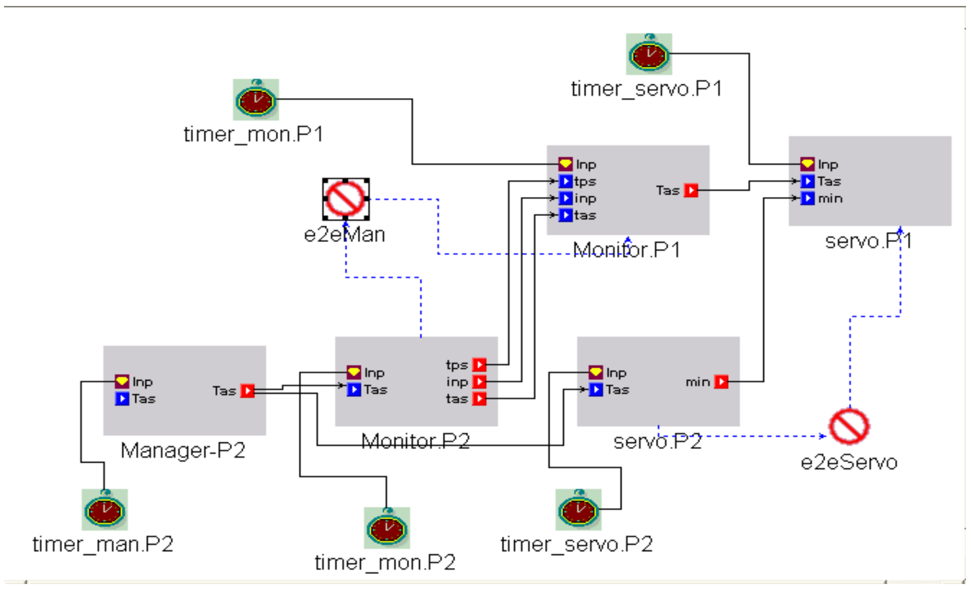
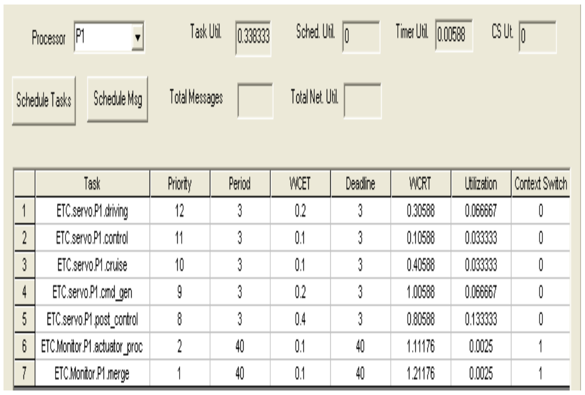


Figure 7.12: Runtime model of the ETC software.

the parameters derived by the analysis, such as priorities, worst-case response times (WCRT), utilizations consumed by the components, and the number of context switches a component/task may experience in the worst case. Note that the worst case for a component may not be experienced for all its invocations due to the mutual exclusion occurrence of the worst-case scenarios for different components. For example, each of ETC.Monitor.P1.actuator_proc and ETC.Monitor.P1.merge may experience one preemption in the worst case during the runtime. However, since they are both preempted by the same set of servo components, this implies that only one of them may experience the worst case at an invocation. This means the worst case for the ETC.Monitor task on P_1 includes only one preemption instead of two, thus the WCRT of the ETC.Monitor.P1 is less than the sum of the WCRTs of ETC.Monitor.P1.actuator_proc and ETC.Monitor.P1.merge.

The AIRES tool can also visualize the analysis results using the Gantt Chart. Here we present only the results for P_1 in Figure 7.14.

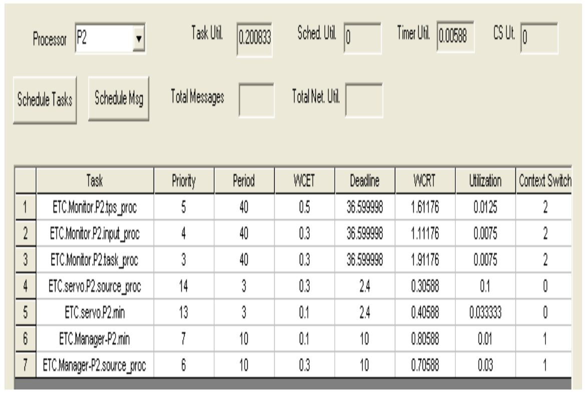
The ETC example shown here passes the schedulability analysis and test, and can then be implemented on the defined target platform.



Processor P1 | Task Util. 0.338333 | Sched. Util. 0 | Timer Util. 0.00588 | CS Ut. 0

	Task	Priority	Period	WCET	Deadline	WCRT	Utilization	Context Switch
1	ETC.servo.P1.driving	12	3	0.2	3	0.30588	0.066667	0
2	ETC.servo.P1.control	11	3	0.1	3	0.10588	0.033333	0
3	ETC.servo.P1.cruise	10	3	0.1	3	0.40588	0.033333	0
4	ETC.servo.P1.cmd_gen	9	3	0.2	3	1.00588	0.066667	0
5	ETC.servo.P1.post_control	8	3	0.4	3	0.80588	0.133333	0
6	ETC.Monitor.P1.actuator_proc	2	40	0.1	40	1.11176	0.0025	1
7	ETC.Monitor.P1.merge	1	40	0.1	40	1.21176	0.0025	1

(a) Runtime analysis of tasks on processor P_1.



Processor P2 | Task Util. 0.200833 | Sched. Util. 0 | Timer Util. 0.00588 | CS Ut. 0

	Task	Priority	Period	WCET	Deadline	WCRT	Utilization	Context Switch
1	ETC.Monitor.P2.tps_proc	5	40	0.5	36.599998	1.61176	0.0125	2
2	ETC.Monitor.P2.input_proc	4	40	0.3	36.599998	1.11176	0.0075	2
3	ETC.Monitor.P2.task_proc	3	40	0.3	36.599998	1.91176	0.0075	2
4	ETC.servo.P2.source_proc	14	3	0.3	2.4	0.30588	0.1	0
5	ETC.servo.P2.min	13	3	0.1	2.4	0.40588	0.033333	0
6	ETC.Manager-P2.min	7	10	0.1	10	0.80588	0.01	1
7	ETC.Manager-P2.source_proc	6	10	0.3	10	0.70588	0.03	1

(b) Runtime analysis of tasks on processor P_2.

Figure 7.13: Runtime analysis results.

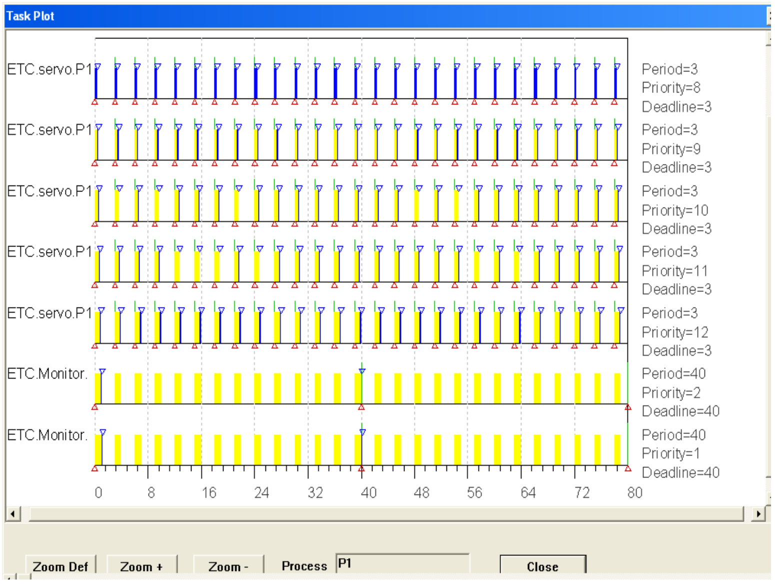


Figure 7.14: Runtime analysis plot for tasks on P_1.

7.6 Summary

The software design tool is essential to support performance analysis with design models and automatic model transformation. In this chapter, we have presented an implementation of a software design tool, called *AIRES*, under the sponsorship of the DARPA IXO MoBIES Program. AIRES allows the designer to interact with the modeling and analysis process, and visualize the results. The AIRES tool is based on a generic modeling environment (GME), and contains a modeling language that implements the modeling framework discussed in Chapter 2. Along with the modeling language, a software design process is built into AIRES to "force" the designer to complete the model information before performing further analysis.

The model transformation and analysis algorithms are all implemented as interpreters, which are loadable dynamic link libraries based on GME BON interface. These algorithms can be used individually or integrally in a development tool chain to automate the design process and model generation. The design techniques described in this dissertation are built into the AIRES tool along with some traditional ECSW design and analysis policies.

Some preliminary evaluations conducted under the MoBIES Program showed that the AIRES tool fills the gap between current design and modeling tools and analysis tools. The flexible implementation that allows the algorithms to be used together to form a tool chain or used as individual tools to integrate with other tools makes it adaptable in customizable design environment with different design, modeling, and analysis requirements.

CHAPTER 8
Conclusions and Future Work

8.1 Conclusions

In this dissertation, we have developed performance modeling techniques to support performance analysis and performance-aware system design for ECSW. These techniques include a performance modeling framework and a set of related methods for modeling, analysis, and transformation. Our framework can be used with functional design models and processes now common in ECSW design. In our framework, we have modeled the software components and architecture, the platform with hardware and system software, and the runtime system with tasks and their communications. Performance-related parameters, classified into constraints and characteristics, have been annotated to the models to support system performance modeling and analysis. All the analysis and transformation methods have been developed to operate using different models defined in the framework. Specifically, we investigated three performance methods: early design performance estimation, performance-aware runtime model generation, and performance analysis of runtime models.

ECSW design begins with the construction of a structural model. This structural model consists of a set of transactions that model the control information processing flows in the ECSW. For each transaction, end-to-end timing constraints are defined. In addition, each transaction contains a set of interconnected software components that are modeled as port-based objects. These software components are annotated with both the performance characteristics (in resource demand) and the performance constraints (in rate and deadline). The performance characteristics of the components can be obtained through measurements. We developed a method based on the virtual service concept to transfer the platform-dependent measurements to platform-independent resource demands for early design analysis and reuse. With this components' performance model, the workloads of the transactions and the system can be computed. The platform where the designed ECSW runs is modeled as a set of hardware components (including processors, network links, and storage devices) and system software (including operating system and middleware services). The performance of the platform is modeled as resource consumptions (for supporting software) and resource capacities (for hardware components). As the design advances, the ECSW design in the structural model evolves to that in a runtime model with more implementation details. We have modeled the ECSW runtime model as a set of intercommunicating tasks, each of which can be implemented directly as a process or a thread in an OS. In a runtime model, the tasks are modeled as a sequence of the components in the structural model. Consequently, the transactions in a structural model are transformed into task chains in a runtime model. The tasks and links in the runtime model have been assigned to the devices and specified with the execution information (invocation rate, release offset, and deadline). This runtime model can then be directly implemented on the given platform.

In current ECSW design practices, performance analysis and tuning is considered as an implementation issue and thus is postponed until the detailed design is complete. For a large ECSW containing thousands of components with complex interactions among them, the software architecture that is determined during the early design phase has a greater fundamental impact on meeting the performance constraints. Early detection of architecture-level design flaws is usually more beneficial. We have therefore developed an early design phase performance estimation method to address this performance analysis issue. The method is based on bound estimations. All analyses use the components' resource demands based on the virtual service rate. To find the performance of the best-case, our method explores the beneficial parallelism, and computes the performance under such a best configuration. In Chapter 3, we have proved that finding the case with the maximum parallelism is NP-hard. Thus, our algorithm uses a heuristic of iterative component grouping to find a sub-optimal solution. The results from this can then be used to guide the platform design to ensure the constructed platform providing sufficient resources to achieve better performance under given costs.

We have further shown how the performance-aware design can be achieved using the performance analysis. In Chapter 4, the methods supporting automatic transformation of a structural model to a runtime model were discussed. The objective of the transformation was to obtain a runtime model that meets the platform resource constraints and the structural model performance constraints. Our transformation is a multi-step process. First, the invocation rates of the transactions were propagated to the individual components. Then, the components in the structural model are allocated to the devices that meet the resource constraints according to the design-specified allocation strategies. To improve the scalability of component allocation, we combined the techniques of branch-and-bound, component ordering with combinational resource consumptions, and forward checking. The result of the allocation is a partition graph with each partition containing components allocated on the same computation device. Then, we distributed the end-to-end system timing constraints over the components as components' timing constraints. These constraints include invocation periods, earliest start time, and latest completion time. The distribution is based on forward and backward tracking of the transaction, and accounts for both computation and communication delays. With the components' execution locations known, the resource consumptions are now represented in real execution times instead of virtual resource demands. We then proved that thus-derived components' timing constraints are sufficient necessary for any feasible schedule.

According to this, the task in the runtime model can be constructed by merging and sequencing the components on the same device. We proved that sequencing the components with all timing constraints met is NP-hard, and used a heuristics of rate similarity and earliest-start-time-first to merge and sequence the components. We further proved that such an approach yields a valid and feasible schedule of the components in a task, with only a minimum total number of tasks in the system, while maintaining schedule flexibility.

In Chapter 5, we presented methods for analyzing the schedulability and performance of the generated runtime model. Since the generated runtime model only specifies the constructs and timing properties, and leaves the scheduling policies to system software, it is required to analyze the generated runtime model under the given scheduling support. We assumed fixed preemptive scheduling in this work. With such system scheduling services, we first need to assign priorities to the tasks in the runtime model in a way such that the tasks could meet their constraints. The assignment is adjusted iteratively until the constraints are met. Then our analysis checks the schedulability of the task set and computes the best-case and worst-case response times of each task. If all tasks meet their performance constraints, the end-to-end performance can surely be met according to the construction of tasks. To reduce the computation complexity of the algorithm for distributed dependent task set schedulability analysis, we introduced a technique based on shared buffers to eliminate task dependencies so that a scalable analysis algorithm can be applied.

We also developed and evaluated a method to improve the systematic measurements of resource consumptions of both application components and system services. The method uses an end-to-end sampling-based technique with a combination of synthetic workloads and micro-benchmarks. This allows the measurement results to be realistic and reusable without requiring the source code. This is particularly effective for system service performance measurements. The evaluations of a set of selected RTOS services validated the effectiveness of the method. The results obtained using this method have also been built as functions in our software design tool to support better performance analysis.

Finally, we have implemented the modeling framework and the related methods in an ECSW design toolkit called AIRES. The AIRES tool uses a generic modeling environment (GME) as its modeling environment, defining the framework in a meta-model with different design models organized in different folders, and implementing the analysis and transformation algorithms as interpreters. We have applied the AIRES tool to both automotive applications such as engine control and vehicle-to-vehicle control, and avionics applications such as mission computing in the DARPA MoBIES Program. The results showed that the AIRES tool filled the gap of integrating the performance analysis in the design process and performance-aware model transformation. Our framework and methods have also been shown to be effective and scalable.

8.2 Future Directions

The first area that deserves further research is to find methods of transforming the control design to the software structural model. Embedded control system design usually starts from the control design by control engineers in the form of control models, such as continuous models in differential equations. In the control design, the designer assumes the system will be working in an ideal environment with sufficient resources. Such a control design must then be transformed into a discrete model for software implementation. This discretization includes determining the granularity for software implementation (component granularity) and assigning system-level end-to-end performance constraints to each control flow. How the control is discretized has a great impact on the construction of the structural model, and hence, is crucial for constructing low-cost platforms and meeting system performance constraints. However, the current methods which are based on engineers' experiences are neither formally verifiable nor free of errors.

A second area needing further research is the effect on control performance of a design choice or strategy. As can be seen throughout this dissertation, design strategies are used at every step in the ECSW design process to help make design decisions. Examples include the strategies used in allocating components (e.g., first-fit, communication minimization), in forming tasks (e.g., rate similarity, earliest-start-time-first), and in assigning priorities. Understanding quantitatively the effects of these strategies on final system resource consumption and control performance will be very valuable. Such results can be used to guide ECSW design and optimization. They can also be used to refine the design by switching automatically between design strategies to better meet resource and performance constraints.

Further, research into methods that can track the causes of performance or resource constraint failures is highly desirable. In current ECSW design practices, violation of performance and/or resource constraints can be detected by simulations or analysis. However, finding the cause of such a failure is difficult and usually depends on the designer's experience and intuition. Methods that link a performance failure with its causes in the design can help the designer find design errors quickly. With such support, it would also be possible for the design tool to refine the design models automatically and thus achieve the resource and performance goals. Such methods can be developed by examining quantitatively the effects of each design parameter.

Finally, a further exploration is required to figure out how to use the techniques developed to support final system generation. Final system generation involves generating a code both for individual components and for system configuration (known as the "glue code"). This model compilation problem is currently an active research area. The solution of such problem would enable full automation of the ECSW design process, and support fast low-cost development of embedded software.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] V. S. Adve, R. Bagrodia, J. C. Browne, E. Deelman, A. Dube, E. N. Houstis, J. R. Rice, R. Sakellariou, D. J. Sundaram-Stukel, P. J. Teller, and M. K. Vernon. POEMS: end-to-end performance design of large parallel adaptive computational systems. *IEEE Transactions on Software Engineering*, 26(11):1027–1048, november 2000.

[2] G. A. Agha and W. Kim. Actors: A unifying model for parallel and distributed computing. *Journal of Systems Architecture*, 45(15):1263–1277, 1999.

[3] F. Andolfi, F. Aquilani, S.Balsamo, and P. Inverardi. Deriving performance models of software architecture from message sequence charts. In *Proceedings of the 2nd International Workshop on Software and Performance*, pages 47–57, Ottawa, Ont, Canada., 2000.

[4] ARTiSAN Software Tools, Inc., I-Logix,Inc., Rational Software Corp., Telelogic AB, TimeSys Corp., and Tri-Pacific Software Inc. Response to the OMG RFP for schedulability, performance, and time. (revised submission). ftp://ftp.omg.org/pub/docs/ad/01-06-14.pdf, June 2001.

[5] N. C. Audsley, A. Burns, M. F. Richard, and A. J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, Atalanta, GA., 1991.

[6] I. J. Bate. *Scheduling and timing analysis for safety critical real-time systems*. PhD thesis, Department of and Computer Science, University of York, York, United Kingdom, November 1998.

[7] D. Batory, R. Cardone, and Y. Smaragdakis. Object-oriented frameworks and product-lines. In *Proceedings of the 1st Software Product-Line Conference*, pages 227–247, Denver, Co., August 1999.

[8] D. Batory, J. Liu, and J. N. Sarvela. Refinements and multi-dimensional separation of concerns. In *Proceedings of the 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 48–57, Helsinki, Finland, September 2003.

[9] R. Bettati. *End-to-end scheduling to meet deadlines in distributed systems*. PhD thesis, Department of and Computer Science, University of Illinois, Urbana-Champaign, IL, 1994.

[10] P. Binns, M. Englehart, M. Jackson, and S. Vestal. Domain-specific software architectures for guidance, navigation and control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):201–227, June 1996.

[11] K. Bradley. *A framework for incorporating real-time analysis into system design processes*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1998.

[12] A. B. Brown and M. I. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of netbsd on the intel x86 architecture. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214–224, June 1997.

[13] A. Burns and A. J. Wellings. A structured design method for hard real-time systems. Technical Report YCS-93-199, University of York, 1993.

[14] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, 1996.

[15] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1989.

[16] P. Cornwell and A. Wellings. Transaction specification for object-oriented real-time systems in HRT-HOOD. In *Lecture Notes in Computer Science (LNCS 1031)*, pages 365–378, 1996.

[17] DARPA ITO. Model-based integration of embedded software. http://www.darpa.mil/ito/Solicitations/CBD_00-19.html, 1999.

[18] D. de Niz and R. Rajkumar. Time weaver: a software-through-models framwork for embedded real-tiem systems. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, pages 144–152, San Diego, CA., June 2003.

[19] M. J. Dent and R. E. Mercer. Minimal forward checking. In *Proceedings of the 6th IEEE International Conference on Tools with Artificial Intelligence*, pages 306–311, 1994.

[20] B. S. Doerr and D. C. Sharp. Freeing product line architectures from execution dependencies. In *Proceedings of Software Technology Conference*, May 1999.

[21] Ford Motor Company, General Motors Corporation, and Motorola Automotive and Industrial Electronics Group. SmartVehicle challenge problems. http://vehicle.me.berkeley.edu/mobies/, 2000.

[22] N. Frampton, J. Tsao, and J. Yen. Windows CE evaluation report: test plan and preliminary results. General Motors PowerTrain Group, April 1998.

[23] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, 1994.

[24] J. J. G. Garcia and M. G. Harbour. Optimized priority assignment for tasks and messages in distributed hard real-time systems. In *Proceedings of the 3rd Workshop on Parallel and Distributed Real-Time Systems*, pages 124–132, April 1995.

[25] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theorey of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[26] R. Gerber, S. Hong, and M. Saksena. Guaranteering real-time requirements with resource-based calibration of periodic processes. *IEEE Transactions on Software Engineering*, 21(7):107–131, July 1995.

[27] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742–760, 1999.

[28] H. Gomaa and D. A. Menasce. Design and performance modeling of component interaction patterns for distributed software architecture. In *Proceedings of the 2nd ACM International Workshop on Software and Performance (WOSP'00)*, pages 117–126, Ottawa, Ont, Canada., 2000.

[29] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transactions on Software Engineering*, 20(1):13–28, January 1994.

[30] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad. Cadena: an integrated development, analysis, and verification environment for component-based systems. In *Proceeding of the 25th IEEE International Conference on Software Engineering*, pages 160–172, Portland, OR., May 2003.

[31] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proceedings of the 1st International Workshop on (EMSOFT 2001)*, volume 2211, pages 166–184, Tahoe City, Ca., Octorber 2001.

[32] H. Hermanns, U. Herzog, U. Klehmer, V. Mertsiotakis, and M. Siegle. Computational performance modeling with the TTIPPtool. In *Computer Perforamnce Evaluation: Modeling Techniques and Tools (LNCS 1469)*, pages 51–62. Springer-Verlag, 1998.

[33] F. Hoeben. Using UML models for performance calculation. In *Proceedings of the second international workshop on software and performance (WOPS)*, Ottwa, Canada, September 2000.

[34] Institute for Software Integrated Systems. The generic modeling environment. http://www.isis.vanderbilt.edu/Projects/gme.

[35] Intel Corporation. Pentium processor family developer's manual, 1997.

[36] F. Jahanian and A. K. Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, December 1994.

[37] R. Jain. *The art of computer systems performance analysis: Techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons, 1991.

[38] J. Jonsson and K. G. Shin. Deadline assignments in distributed hard real-time systems with relaxed locality constraints. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, pages 432–440, Baltimore, MD, May 27-30 1997.

[39] P. Kahkipuro. Performance modeling framework for CORBA based distributed systems. Technical Report A-2000-3, Department of Computer Science, University of Helsinki, 2000.

[40] D.-I. Kang, R. Gerber, and M. Saksena. Performance-based design of distributed real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Application Symposium RTAS'97*, pages 2–13, Montreal, Canada, June 1997.

[41] R. Kar. Implementingthe rhealstone real-time benchmark. *Dr Dobb's Journal*, April 1990.

[42] G. Karsa, *et al.* Model-integrated system development: models, architecture and process. In *Proceedings of the 21st Annual International Computer Software and Application Conference (COMPSAC'97)*, Bethesda, MD, August 1997.

[43] K. A. Kettler, D. I. Katcher, and J. K. Strosnider. A modeling methodology for real-time/multimedia operating systems. In *Proceedings of the Real-Time Technology and Applications Symposium*, pages 15–26, May 1995.

[44] S. Kim, S. Cho, and S. Hong. Schedulability-aware mapping of real-time object-oriented models to multi-threaded implementations. In *Proceedings of the IEEE Real-Time Computing Systems and Applications Symposium*, pages 7–14, 2000.

[45] D. L. Kiskis and K. G. Shin. SWSL: A synthetic workload specification langauge for real time systems. *IEEE Transations on Software Engineering*, 20(10):798–811, October 1994.

[46] H. Kopetz. The time-triggered model of computation. In *Proceedings of the IEEE Real-Time System Symposium*, pages 168–177, Madrid, Spain, December 1998.

[47] C. M. Krishna and K. G. Shin. *Real-Time Systems*. The McGraw-Hill Companies, Inc, 1997.

[48] S. Kumar, J. H. Aylor, B. W. Johnson, and W. A. Wulf. *The codesign of embedded systems: A unified hardware/software representation*. Kluwer Academic Publishers, 1996.

[49] V. Kumar. Algorithms for constraint satisfaction problems: a survey. *A.I. Magazine*, 13(1):32–44, 1992.

[50] S.-K. Kweon and K. G. Shin. Achieving real-time communication over Ethernet with adaptive traffic smoothing. In *Proceedings of IEEE Real-Time Technology and Applications Symposium (RTAS'2000)*, pages 90–100, June 2000.

[51] Y. K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, December 1999.

[52] E. A. Lee. What 's ahead for embedded software? *IEEE Computer*, 33(9):18–26, September 2000.

[53] E. A. Lee. Overview of the Ptolemy project. Technical Memorandum UCB/ERL M01/11, March 6 2001.

[54] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitary deadlines. In *Proceedings of the 11th IEEE Real-Time System Symposium*, pages 201–209, December 1990.

[55] J. P. Lehoczky. Real-time queueing network theory. In *Proceedings of the 18th IEEE Real-Time System Symposium (RTSS'97)*, pages 58–67, San Francisco, Ca., December 1997.

[56] D. J. Lilja. *Measuring computer performance: A practitioner's guide*. Cambridge University Press, 2000.

[57] C. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of ACM*, 20(1):40–61, 1973.

[58] J. Liu and E. A. Lee. Timed multitasking for real-time embedded software. *IEEE Control Systems Magazine*, 23(1):65–75, February 2003.

[59] L. McVoy and C. Staelin. *lmbench*: Protable tools for performance analysis. In *Proceedings of the USENIX 1996 Annual TEchnical Conference*, San Diego, CA, January 1996.

[60] A. K. Mok and A. X. Feng. Real-time virtual resource: A timely abstraction for embedded systems. In *Proceedings of the 2nd International Workshop on Embedded Software (EMSOFT 2002)*, volume 2491, pages 182–196, Grenoble, France, Octorber 2002.

[61] Motorola. MPC555 user's manual, revised 15, September 1999.

[62] J. C. Palencia, J. Garcia, and M. G. Harbour. Best-case analysis for improving the worst-case schedulability test for distributed hard real-time systems. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, pages 35–44, Berlin, Germany, June 1998.

[63] J. C. Palencia and M. G. Harbour. Schedulability analysis for task with static and dynamic offsets. In *Proceedings of the 19th IEEE Real-Time System Symposium*, pages 26–37, Madrid, Spain, December 1998.

[64] D. Peng, K. G. Shin, and T. F. Abdelzaher. Assignment and scheduling communicating periodic tasks in distributed real-time systems. *IEEE Transactions on Software Engineering*, 23(12):745–757, December 1997.

[65] D. Petriu, H. Amer, S. Majumdar, and I. Abdull-Fatah. Using analytic models for predicting middleware performance. In *Proceedings of the 2nd ACM International Workshop on Software and Performance (WOSP'00)*, pages 117–126, Ottawa, Ont, Canada., 2000.

[66] T. Quatrani. *Visual Modeling with Rational Rose and UML*. Addison-Wesley, 1998.

[67] O. Redell and M. Sanfridson. Calculating exact worst case response times for static fixed priority scheduled tasks with offsets and jitters. In *Proceedings of the 8th Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, pages 164–172, San Jose, CA., September 2002.

[68] O. Redell and M. Sanfridson. Exact best-case response time analysis of fixed priority scheduled tasks. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 165–172, Vienna, Austria, June 2002.

[69] M. Saksena, P. Karvelas, and Y. Wang. Automatic synthesis of multi-tasking implementations from real-time object-oriented models. In *Proceedings of the IEEE Symposium on Object-Oriented Real-Time Distributed Computing*, pages 360–367, Los Alamitos, CA, March 2000.

[70] M. Saksena, A. Ptak, P. Freedman, and pawel Rodziewicz. Schedulability analysis for automated implementations of real-time object-oriented models. In *Proceedings of the IEEE Real-Time System Symposium*, pages 31–41, Madrid, Spain, December 1998.

[71] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. The MIT Press, Cambridge, Ma., 1989.

[72] D. C. Schmidt, M. Deshpande, and C. O'Ryan. Operating system performance in support of real-time middleware. In *Proceedings of the 7th IEEE Workshop on object-oriented real-time dependable systems*, San Diego, CA, January 2002.

[73] S. Schneider. *Concurrent And Real-Time Systems: The CSP approach*. John Wiley & Sons, Ltd., 2000.

[74] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., 1994.

[75] B. Shirazi, L. Welch, B. Ravindran, C. Cavanaugh, B. Yanamula, R. Brucks, and E. nam Huh. DynBench: a dynamic benchmark suite for distributed real-time systems. In *Parallel and Distributed Processing: IPPS/SPDP Workshops*, pages 1335–1349, Berlin, Germany, April 1999.

[76] C. Shousha, D. Petriu, A. Jalnapurkar, and K. Ngo. Applying performance modeling to a telecommunication system. In *Proceedings of the 1st International Workshop on Software and Performance*, pages 1–6, Santa Fe, NM., Octorber 1998.

[77] C. U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley Publishing Company, 1990.

[78] C. U. Smith and M. Woodside. Chapter 26: Performance validation at early stages of software development. In E. Gelenbe, editor, *System Performance Evaluation: Methodologies and Applications*, pages 383–396. CRC Press LLC, Boca Raton, FL., 2000.

[79] J. Stankovic. VEST: a toolset for constructing and analyzing component based operating systems for embedded and real-time systems. Technical report, University of Virginia, 2000.

[80] D. B. Stewart. Measuring execution time and real-time performance. In *Embedded Systems Conference*, San Francisco, CA, April 2001.

[81] J. Sun. *Fixed-priority end-to-end scheduling in distributed real-time systems*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1997.

[82] J. Sun and J. W. S. Liu. Synchronization protocols in distributed real-time systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 38–45, May 1996.

[83] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Publishing Company, 1997.

[84] The Boeing Company. Challenge problems for model-based integration of embedded software: Weapon system open experimental platform, 2001.

[85] Timesys Corp. Using timewiz to understand systemtiming — before you build or buy. http://www.timesys.com.

[86] K. Tindell, A. Burns, and A. Wellings. Allocating real-time tasks: An NP-hard problem made easy. *Real-Time System Journal*, 4(2), May 1992.

[87] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing & Microprogramming*, 50(2-3):117–134, April 1994.

[88] S. Toeppe and S. Ranville. RTOS evaluation and selection criteria for embedded automotive powertrain applications, 1999.

[89] Tri-Pacific Software Inc. RapidRMA: the art of modeling real-time systems. http://www.tripac.com/html/prod-fact-rrm.html.

[90] S. Wang, S. Kodas, K. G. Shin, and D. L. Kiskis. Measurement of OS services and its application to performance modeling and analysis of integrated embedded software. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'2002)*, pages 78–87, San Jose, California, September 2002.

[91] N. Welderman. Hartstone: synthetic benchmark requirements for hard real-time applications. Technical Report CMU/SEI-89-TR-23, Software Engineering Institute, Carnegie Mellon University, 1989.

[92] L. G. Williams and C. U. Smith. Performance evaluation of software architecture. In *Proceedings of the 1st International Workshop on Software and Performance*, pages 164–177, Santa Fe, NM., October 1998.

[93] Wind River System, Inc. Osekworks 4.0 calls references, 2000.

[94] C. Woodside, J. Neilson, D. Petriu, and S.Majumdar. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Transactions on Computer*, 1(44):20–34, January 1995.

[95] M. Woodside, C. Hrischuk, B. Selic, and S. Bayarov. A wideband approach to integrating performance prediction into a software design environment. In *Proceedings of the 1st International Conference on Software Engineering*, pages 31–41, Santa Fe, NM., October 2002.

[96] M. Woodside, D. Petriu, and K. Siddiqui. Performance-related completions for software specifications. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 22–32, Orlando, FL., May 2002.

[97] J. Xu and D. Parnas. On satisfying timing constraints in hard real-time systems. *IEEE Transactions on Software Engineering*, 19(1):70–86, January 1993.

[98] H. Zeltwanger. An inside look at teh fundamentals of CAN. *Control Engineering*, 42(1):81–87, January 1995.

[99] Q. Zheng and K. G. Shin. On the ability of establishing real-time channels in point-to-point packet-switched networks. *IEEE Transactions on Communications*, 42(234):1096–1105, February-April 1994.

[100] L. Zhou. *Real-time performance guarantees in manufacturing systems*. PhD thesis, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan, May 1999.

[101] K. M. Zuberi and K. G. Shin. Scheduling messages on Controller Area Network for real-time CIM applications. *IEEE Transactions on Robotics and Automation*, 13(2):310–314, April 1997.