# Early-Stage Performance Modeling and Its Application for Integrated Embedded Control Software Design *

Shige Wang and Kang G. Shin
Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
1301 Beal Avenue
Ann Arbor, MI 48109-2122
{wangsg,kgshin}@eecs.umich.edu

## ABSTRACT

Most of current embedded control software (ECSW) development techniques deal only with performance specifications during the early software design phase and delay the modeling and analysis until the detail design has been completed. In this paper, we propose a new approach to modeling and analysis of the performance of the designed ECSW without knowing the platform configuration and the software deployment. The functional model is assumed — as is commonly the case in practice — to be constructed by integrating existing (reusable) software components. The performance of components and connections in the model are modeled using annotated requirements and virtual resource demands. Our algorithm then computes the system performance such as end-to-end delays of transactions and workloads by traversing the model. The results can be applied to the platform design and runtime architecture generation. To demonstrate the usefulness of the proposed method in real world applications, we present the analysis of automotive vehicle-to-vehicle control software as an example.

## Keywords

performance modeling, performance-aware design, embedded software, integrated system

## 1. INTRODUCTION

Performance modeling and analysis at early design stages is crucial to the runtime correctness of embedded control software (ECSW), which typically runs in a resource-limited environment and must meet stringent performance constraints. At an early design stage, the implementation details of a designed ECSW and its execution decisions are not fully determined. The early-stage performance modeling and analysis can serve the purposes of: (1) evaluating the completion and satisfiability of performance specifications for the designed software architecture, (2) verifying and assisting platform design/configuration for sufficient resources for the ECSW execution, and (3) guiding the runtime model design such as the allocation and scheduling of components. Existing research results of both performance modeling [5, 7, 1] and real-time scheduling [3, 6] are not applicable to the early design stage since implementation details such as components' execution locations and scheduling policies are essential to the application of these techniques.

In this paper, we present a new method for early design-stage performance modeling and analysis without the implementation details. In this method, the designed ECSW is modeled as a graph of software components and their connections, called *structural model*. The software components in the structural model have annotated domain-specific performance parameters, whose values represent the resource demands and performance requirements of the components. We introduce virtual resources to map the measured platform-dependent resource demands of the components to platform-independent virtual resource demands. The performance of the structural model can then be derived by tracing the graph with performance annotated modeling elements. The designer can use the obtained performance to design and verify the platform, and generate a runtime model for implementation.

The rest of this paper is organized as follows. Section 2 describes the software structural model, based on which we develop our performance modeling and analysis method. Section 3 presents our performance modeling method, including performance annotations of modeling elements, and algorithms to derive the system performance. Section 4 discusses the application of the modeling results, especially to platform design and runtime model generation. Section 5 shows an example of the performance modeling and analysis of automotive vehicle control software using the proposed method. The paper concludes with Section 6.

## 2. SOFTWARE STRUCTURAL MODEL

A software structural model defines the software architecture that implements the system functional behaviors. In current ECSW development, the software is usually constructed by synthesizing software components, including device drivers, data processing algorithms, and control algorithms.

A *software component* in a structural model is formally defined as a pair $c = < B_c, P_c >$. $B_c$ is a set of actions to transform the component's inputs to outputs under control of a behavior specification. $P_c$ defines a set of ports for inputs ($I_c$) and outputs ($O_c$) with $P_c = I_c \bigcup O_c$ and $I_c \bigcap O_c = \emptyset$. A *port* specifies an external interface through which a component communicates with other software

*The work reported in this paper was supported in part by DARPA under AFRL contracts F3615-00-1706 and F30602-01-02-0527.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*WOSP'04* January 14-16, 2004, Redwood City, California.
Copyright 2004 ACM 1-58113-673-0/04/0001 ...$5.00.

component(s). Here we assume that the components are *process-oriented*, each containing a single operation to react its inputs for each mode.[1] A model constructed with *object-oriented* components like UML classes can be converted to the one with process-oriented components using port dependency graphs [2].

The interactions among components are defined as *connections*. A connection specifies the information dependency between two components. A *connection* is formally defined as a partial relationship $l = < x,y >$ between components $c_1$ and $c_2$ that satisfies $x \in O_{c_1} \wedge y \in I_{c_2} \wedge Type(x) = Type(y)$. $Type(x)$ and $Type(y)$ define the types of port $x$ and $y$. A connection can be either *synchronous* or *asynchronous*. A connection $< x,y >$ is *synchronous* if the output from port $x$ directly triggers the execution of actions associated to port $y$. Similarly, a connection $< x,y >$ is said to be *asynchronous* if the invocations of the action associated with port $y$ do not depend on the output from port $x$. Given the definitions of components and connections, a structural model is formally defined as a set of directed graphs $M_s = < C_s, L_s >$, where $C_s$ is a set software components, and $L_s$ a set of connections.

The system behaviors are modeled as *transactions* in the structural model, each of which is a directed graph for a parallel control process. Components in a transaction are classified into *input components*, *output components*, and *processing components*. An *input component* models a start point of a transaction, while an *output component* models an end point of a transaction.

Multiple connections are allowed to come from/to a component using the same or different ports. These connections are assumed to be AND if they run at the same rate. If the rates are different, the component runs at either one of the rates or the combination of some input rates according to the component's functional role. A component running at one of its input rates is called a synchronization component. A component running at the combination rate of its inputs is called a shared component. In our structural model, a copy of the shared component is created in every transaction using it. These transactions can then be treated as independent graphs. Data dependency cycles among components in a transaction are also allowed to model multi-rate systems.

The activations of transactions depend on the system modes. Some transactions may be active in multiple modes, while others are active in only one mode. Automatic identification of concurrent transactions in each mode is beyond the scope of this paper, and is discussed somewhere else [4].

We also assume that high-level performance constraints are given for each transaction in the structural model. In this paper, we focus only on timing performance, and assume that end-to-end deadlines, invocation rates, and bounds of response and output jitter are given for all transactions.

## 3. PERFORMANCE MODELING AND ANALYSIS METHOD

In this section, we discuss the performance of modeling elements, and the method for deriving the system performance.

### 3.1 Component performance modeling

The performance of a modeling component may contain both performance requirements such as activity deadlines and invocation frequencies, and the performance characteristics such as execution times and message delays. We therefore classify the performance parameters into two disjoint categories: *characteristics* and *constraints*. To associate the performance parameters with the components, we choose the annotation approach. Figure 1 shows the component structure with annotated performance parameters. Similarly, the communication-related performance parameters and values can be annotated to connections.

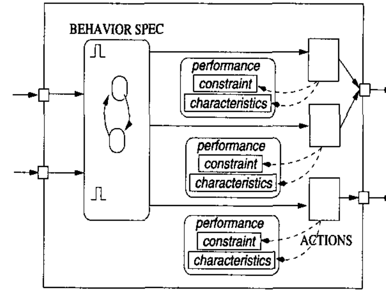


**Figure 1: Component structure with performance parameters.**

In order to evaluate the performance quantitatively, the values of the performance parameters for all constituent components must be determined. Deriving values of performance constraints of a component is fairly easy and can be done using techniques like rate propagation and deadline distribution. The values of performance characteristics, on the other hand, vary with the platform and cannot be determined at an early design stage before knowing the platform configuration. Since we are interested in modeling the performance values uniformly so that they are additive and comparable, we introduce the virtual resource demands for the values of performance characteristics. To obtain the virtual resource demands of a component, we assume that a dedicated resource exist for the execution of the component. We call such a dedicated resource a *virtual resource*. The virtual resources serves the components at a fixed rate, called *virtual service rate*. It is used as a scalar to uniform the software resource demands. The service demands of a component is then defined as the virtual time required to complete the component's workload on the dedicated virtual resource.

To generate the virtual resource demands reflecting the component's workload on a real platform, we first need to measure the component's workload on a reference target. Given the measured execution time $e_i^p$ of a component on a reference target $p$, the platform service rate of the target $r_p$, and the virtual service rate of the resource $r_R$, the virtual resource demand $e_i$ of the component can then be computed as $e_i = \frac{e_i^p \cdot r_p}{r_R}$.

Further, since the communication and computation resources are different possibly with different virtual service rates, we define a *computation-communication service ratio* $\delta$ as $\delta = \frac{r_v^{comp}}{r_v^{comm}}$. in order to treat the virtual resource demands of different resources uniformly during modeling and analysis.

### 3.2 Performance computation

Our performance computation is based on the graph traversal algorithm. Before using the graph traversal algorithm, we need to transform the structural model to a directed acyclic graph with single input and single output. A cycle in a model represents an inner control loop inside a transaction. Its rate must be an integral multiple of the rate of the transaction. To remove the cycle, we can separate the original transaction into two: one represents the cycle with the feedback connection eliminated, and the other represents the original transaction with the cycle replaced by an aggregate node. The rate of the transaction of the cycle is assigned to be the rate of the inner control loop. The resource demands of the aggregate

[1] Although there can be a sequence of actions performed inside a component, these actions can be treated as a combinational one since the sequence appears indivisible to the outside world.

node is assigned to be the resource demand bounds of the cycle. To eliminate multiple inputs and outputs of a transaction, we introduce a pseudo start and end node with a connection between the start node to every input node and a connection between every output node to the end node. The resource demands of start and end nodes are both assigned to be zero. The connection delays between start-input and output-end can be assigned as the input and output jitter bounds.

After the model is transformed to an acyclic, single-input single-output directed graph, the performance computation can be done based on the weighted graph search. In this paper, the end-to-end processing delay (response time) of each transaction and total system resource demands of the system are used as performance metrics.

### *3.2.1 End-to-end delay*

The end-to-end delay of a transaction is the time taken to complete all its components' executions. The end-to-end delay is evaluated based on *bound estimations* [5] of transactions in our analysis. The bound estimations are computed using best-case and worst-case scenarios under the assumption of sufficient resources without contention among concurrent transactions.

In the bound estimation, the best-case bound is of the most interest as it is useful for checking the satisfiability of the constraints and sufficiency of platform resources. A best-case scenario is the scenario with the minimum weight of the longest path in the transaction graph among all possible allocations of its parallel paths, where the weight of each component/connection is assigned as its virtual resource demand. This transforms the problem into a component allocation problem, i.e., finding an optimal allocation of the transaction's components that maximizes the benefit of parallel execution. To find the optimal allocation, we first need to differentiate the resource demand of a connections as local and remote, denoted by $e_l(local)$ and $e_l(remote)$, respectively. Then for a given transaction with multiple concurrent paths, we examine the weight of the longest path under different allocations of concurrent paths using local or remote communications. An allocation that yields the minimum weight of the longest path is the optimal one. The best-case end-to-end delay bound can then be computed as the weight of the longest path in the optimal allocation.

Finding the optimal allocation leading to minimum end-to-end delay is an NP-hard problem. In this paper, we use a simple heuristic algorithm to find a best-case scenario: if a parallel remote path with the maximum weight (fully parallel) has greater weight than a path of merging the remote paths with a local one to execute on the same virtual resource (sequentially), we allocate the components of both paths to the same virtual resource. Otherwise, we keep them on different resources for parallel execution. Figure 2 shows the algorithm to compute the best end-to-end delay of a given transaction.

### *3.2.2 System resource demands*

The system resource demands include the resource demands of all concurrently active transactions and all their instances. Since the connections and computations consume different types of resources, their demands are computed separately.

At runtime, the system resource demands vary because the number of active transactions and the number of their instances change dynamically. For a transaction $T$ with the arrival rate $r_T$ and the resident time $t_T$, it can be shown that the number of concurrently active instances of the transaction $T$ is $\lfloor t_T \cdot r_T \rfloor \le N_I(T) \le \lfloor t_T \cdot r_T \rfloor + 1$ For an instance of transaction $T$, given each component $i$ and connection $l$ in $T$ with resource demand $e_i$ and $e_l$, respectively, the

```
Algorithm 1.Compute-Delay
Input: transaction as T =< N, L >, s: the start node, e: the end node.
Output: E2E delay D between (s,e), path P =< N_P, L_P > yields D
Begin
       /* initialize node and link weight as resource demands */
01     foreach n ∈ N, l ∈ L do
02         w(n) = e_n;     w(l) = e_l(local);
03     end-for
04     P = find_longest_path(s, e, w);
05     D = Σ_{n∈N_P} w(n) + Σ_{l∈L_P} w(l);
06     for a sub-path P(i,j) ∈ P(s,e) that has m alternative paths P^1(i,j)...P^m(i,j) between i and j do
07         foreach P_k(i,j), (1 ≤ k ≤ m) do
08             w(l_ki) = e_{l_ki}(remote); /* l_ki: connection between i and the first node on P^k(i,j) */
09             w(l_kj) = e_{l_kj}(remote); /* lpkj: connection between the last node on P^k(i,j) and j */
10         end-for
11
12         repeat
13             P^x(i,j) = max_{1≤k≤m}(P^k(i,j) with maximum D);
14             D^x(i,j) = Σ_{n∈N_{P^x(i,j)}} w(n) + Σ_{l∈L_{P^x(i,j)}} w(l);
15             D(i,j) = Σ_{n∈N_{P(i,j)}} w(n) + Σ_{l∈L_{P(i,j)}} w(l);
16
17             if (D(i,j) + e_{l_xi}(local) + e_{l_xj}(local)) ≤ (w(l_xi) + w(l_xj)) then
18                 P = P+P^x(i,j)-(i,j);
19                 D = D + D^x(i,j) - w(i) - w(j) - w(l_xi) - w(l_xj) + e_{l_xi}(local) + e_{l_xj}(local);
20             else
21                 P = P-P(i,j)+P^x(i,j);
22                 D = D - D(i,j) + D^x(i,j);
23             end-if-else;
24         until all P_k(i,j) are checked or D^x(i,j) ≤ D(i,j);
25     end-for
26     return P, D;
End.
```

**Figure 2: Algorithm of finding the best end-to-end delay.**

resource demands to complete this instance of $T$ can be computed as

$$w_T(comp) = \Sigma_{i \in T} e_i, \quad w_T(comm) = \Sigma_{l \in T} e_l$$

In the system resource demands computation, the worst-case workload of the system is of the most interest. Such workload is usually used for platform evaluation and capacity planning as meeting the timing constraints under the worst-case workload guarantees meeting the same timing constraints for all scenarios. The worst-case workload happens when a transaction $T$ generates the maximum number of its active concurrent instances that fully utilize its virtual resource. Given the deadline of transaction $T$ is $D_T$, the virtual resource of $T$ is fully utilized when its resident time is its deadline $D_T$. Therefore, the maximum number of instance of transaction $T$ is $N_I(T) = 1 + \lfloor r_T \cdot D_T \rfloor$. So, for a given transaction $T$ with invocation rate $r_T$, resource demands in each instance $w_T$, the deadline $D_T$, and the computation-communication ratio $\delta$, the maximum resource demands in the duration of $D_T$ is

$$w_T^{max} = (w_T(comp) + \delta w_T(comm)) \cdot (1 + 2 \sum_{i=1}^{\lfloor r_T \cdot D_T \rfloor} (1 - \frac{i}{r_T \cdot D_T}))$$

The resource demand of ECSW can then be computed after the worst-case resource demand of every transaction. is obtained. For each system mode $m$, the system resource demand $w_{sys}^m$ is the sum of the resource demands of all active transactions under $m$. We use the maximum resource demands among all modes as the system resource demands. Thus, the system resource demand is computed as

$$w_{sys} = \max_{all m} \{ \sum_{all T \in \{T^m\}} (r_T \cdot w_T^{max}) \}$$

# 4. APPLICATION OF MODELING RESULTS

The obtained performance results can be used to evaluate the performance specifications and assist design. In this paper, we focus on finding the platform configuration that provides sufficient resource to satisfy the performance constraints.

In order to provide sufficient resource for the ECSW execution, the platform should have a minimum service rate that guarantees all transactions to meet their deadlines. We call such a service rate *minimum desired service rate*, and compute it as $r^T = w_T^{max}/D_T$. The minimum desired service rate for a system mode $m$ can be computed as $r^m = \sum_{\forall T \in \{T^m\}} r^T$, and the desired service rate for the system is $r^{sys} = \max_{\forall m}(r^m)$. Given a platform consists of $k$ processing units $P_1, \ldots, P_k$, and $n$ connection links $L_1, \ldots, L_n$, the following equations must be held for the platform to provide sufficient resource to guarantee the performance constraints to be met:

$$\Sigma_{i=1}^{m} r_p(i) \geq r^{sys}(comp), \quad \Sigma_{i=1}^{n} r_l(i) \geq r^{sys}(comm)$$

After the platform configuration is determined, a runtime model must be constructed with more design details for implementation. A critical step in the runtime model generation is allocating software components and connections to physical components in the platform. Here we use the service rate to make the allocation decision. For a component $c$ with $e_c$ under $r_v$, if it belongs to transaction $T$ with $D_T$, $c$ requires the desired service rate $r_c = e_c \cdot r_v / D_T$ to allow $T$ meeting its deadline. We then modify the traditional utilization-based allocation algorithm by (i) substituting the utilization bounds of a device with the service rate of the device, and (ii) substituting the utilization introduced by a component with its minimum desired service rate. The new algorithm can then allocate the components in the abstract structural model without knowing real execution time. After the components in ECSW are allocated, task formation and timing assignment can be done as in [4]. A follow-up traditional timing and schedulability analysis should be performed to verify the satisfaction of timing constraints after the runtime model is generated completely.

# 5. EXAMPLE: DESIGN AND ANALYSIS OF VEHICLE CONTROL SOFTWARE

In this section, we present an example of vehicle control software (VCS) to illustrate the use of our modeling method. The main functionality of the vehicle control software is to perform Cooperative Adaptive Cruise Control (CACC) with Cooperative Forward Collision Warning (CFCW). The responsibility of ECSW is to monitor the distance between a moving vehicle and those around it, predict if collision will occur, and adjust the speed of the vehicle by either increasing engine speed or applying brake. To simplify the discussion, we use only two transactions that are representative and sufficient for the discussion. The models of these transactions with the performance parameters are shown in in Figure 3. The values shown in the model have been artificially translated from the original ones with the computation-communication service rate ratio $\delta = 1$. All rates in the model are in Hz, while the resource demands of components and connections are based on virtual resource service rate $r_v = 1$. Also for simplicity, we assume the deadlines of the transactions equal to their periods.

In the VCS ECSW structural model, transaction $T_1$ runs in all system modes, while $T_2$ and $T_3$ only run in *braking* mode and *accelerating* mode, respectively.

To compute the end-to-end delay of each transaction, we first eliminate the cycles in $T_1$ by separating *lat_sense* $\rightarrow$ *MBI* as another transaction $T_1'$ with rate $r = 500$, and represent $T_1'$ as a single node in $T_1$. For $T_2$ and $T_3$, we need to add a *start* node since they both have multiple input components. We show $T_1$ and $T_2$ models after transformation in Figure 4. $T_3$ appears the same as $T_2$ after transformation.

We now apply Algorithms 1 and 2 to each transaction to find the end-to-end delays for best- and worst-case scenarios. Taking $T_2$ as an example. For the best-case, the longest path found in the first round is $P(s, brake) = \{s, long_sense, long_cntrl, speed_cntrl, brake_cmd\}$ with the delay $D = 179$. There are 2 alternative paths between ($s$, *hi_cntrl*), 3 between ($s$, *long_cntrl*), and 5 between ($s$, *speed_cntrl*). To evaluate the alternative paths, the weights of connections *gps* $\rightarrow$ *hi_cntrl*, *radar_read* $\rightarrow$ *hi_cntrl*, and *hi_cntrl* $\rightarrow$ *speed_cntrl* are replaced by the values for the remote communications, namely 60, 180, and 144, respectively. The longest path between $(s, speed)$ after such a change becomes $P^x(s, speed) = \{s, radar_read, hi_cntrl, speed_cntrl\}$ with the $D^x(s, speed) = 381 \geq 170 = D(s, speed)$. At this point, we need to evaluate the scenario where all components on $P^x(s, speed)$ running on the same resource with $P(s, speed)$. The result shows that when running all components on both paths on the same local resource, $D(s, speed) = 203 \leq D^x(s, speed)$. Therefore, the new path will include components $P(s, brake) = \{s, long_sense, radar_read, hi_cntrl, long_cntrl, speed_cntrl, brake_cmd\}$. Repeating the process results in the *GPS* path being allocated to the same resource. The best-case end-to-end delay for $T_2$ is then $D = 219$. For the worst-case, *long_sense*, *GPS_read*, and *radar_read* are merged into one node *LGR* with resource demands 56 for the new nodes. Links *LGR* $\rightarrow$ *long_cntrl* and *LGR* $\rightarrow$ *hi_cntrl* are with resource demands 118 and 240. Although there are alternative paths between *hi_cntrl* and *speed_cntrl*, *long_cntrl* cannot be merged. Therefore, the longest path for the worst-case is $P(s, brake) = \{s, LGR, hi_cntrl, long_cntrl, speed_cntrl, brake_cmd\}$ with the delay $D = 935$. Table 1 presents the best-case and worst-case end-to-end delays for each transaction.

| transaction | best-case (comp+conn) | worst-case (comp+conn) |
|---|---|---|
| $T_1'$ | 69 (69+0) | 131 (69+62) |
| $T_1$ | 124 (100+24) | 295 (100+195) |
| $T_2$ | 219 (157+62) | 935 (157+778) |
| $T_3$ | 224 (165+62) | 943 (155+778) |

**Table 1: End-to-end delays of individual transactions.**

Similarly, we can compute the resource demands for transactions as well as for the overall system. The resource demands of the given VCS example are listed in Table 2.

| transaction | computation | local comm. | remote comm. |
|---|---|---|---|
| $T_1$ | 36050 | 1200 | 37450 |
| $T_2$ | 7850 | 5250 | 52000 |
| $T_3$ | 8250 | 5250 | 52000 |
| system | 44300 | 6450 | 89450 |

**Table 2: Resource demands of the system.**

These performance estimates can be used for design assistance and evaluation. For example, let's assume that the end-to-end deadline for each transaction equals its period, the deadlines are $2ms$ for $T_1'$, and $20ms$ for $T_1$, $T_2$, and $T_3$. Then, the selected platform should have at least service rate $r_p$ for computation and $r_c$ for communication such that

$$\begin{cases} 69/r_p \leq 2ms \\ 100/r_p + 24/r_c \leq 20ms \\ 157/r_p + 62/r_c \leq 20ms \\ 165/r_p + 62/r_c \leq 20ms. \end{cases}$$

According to the above equation, the required service rates for computation and communication are $r_p \geq 38500$ and $r_c \geq 8946$,

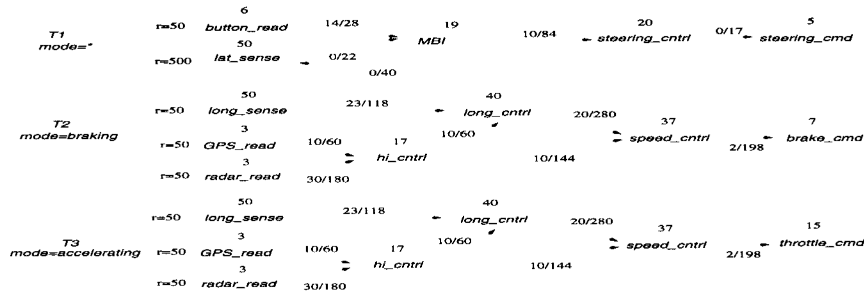


**Figure 3: Software structural model with 3 transactions in the VCS system.**

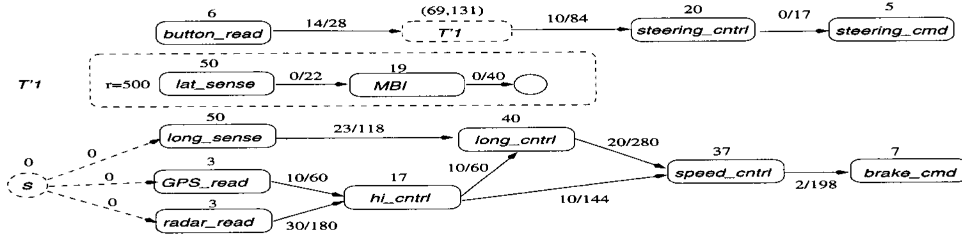


**Figure 4: Transaction graph after transformation.**

respectively, to meet the end-to-end deadlines for all transactions in the best-case. Similarly, the worst-case service rates for computation and communication can be derived. Then, the selected platform should provide the maximum service rates for both resources. Suppose the selected resource has the minimum service rates (38500, 8946) for both computation and communication, then at least 2 computation resources ($\lceil 44300/38500 \rceil$) and 10 connection resource ($\lceil 89450/8946 \rceil$) are required to provide sufficient resource for the ECSW to execute. We can also see in this example that the remote communications require more resource and introduce longer delays in the worst-case scenarios. Therefore, reducing the remote communication (by either changing the connection in the structural model or providing a faster communication mechanism) will more likely improve the overall system performance. It can also be seen in Table 2 that $T_1'$ consumes over 80% of the total computation resource. Therefore, improving the performance of $T_1'$ can potentially improve the performance of the system. It will yield better performance to allocate $T_1'$ on a computation resource with a faster service rate if the computation resources in the platform are asymmetric.

# 6. CONCLUSIONS

In this paper, we presented a method to model and analyze the ECSW performance at an early design stage with incomplete information. The performance modeling is based on a software structural model, which is constructed by integrating reusable software components. The structural model contains a set of graphs of transactions. Each transaction implements some system behavior(s). The performance model is constructed based on the structural model and performance of individual components, which are classified into parameters for constraints and for characteristics. We used virtual resource demands to model component performance characteristics, and made their values platform-independent. The performance and resource demands can then be computed using the virtual resource demands of the constitute components and connections in the model without requiring the platform and deployment information. Specifically, we computed the end-to-end delay that achievable in the best-case, and system resource demands. The results can be applied to guide platform design and components' allocation for runtime model generation. We demonstrated how the our proposed method is used in a VCS system design. All of these demonstrated the power and utility of our method in bringing the performance analysis to a higher-level abstraction.

# 7. REFERENCES


[1] H. Gomaa and D. A. Menasce. Design and performance modeling of component interaction patterns for distributed software architecture. In *Proceedings of the 2nd International Workshop on Software and Performance*, pages 117–126, Ottawa, Ont, Canada., 2000.

[2] Z. Gu, S. Kodase, S. Wang, and K. G. Shin. A model-based approach to system-level dependency and real-time analysis of embedded software. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'2003)*, pages 78–87, Washington D.C., May 2003.

[3] 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.

[4] S. Kodase, S. Wang, and K. G. Shin. Transforming structural model to runtime model of embedded software with real-time constraints. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, pages 170–175, Munich, Germany, March 2003.

[5] C. U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley Publishing Company, 1990.

[6] 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.

[7] 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.