# Component Allocation with Multiple Resource Constraints for Large Embedded Real-Time Software Design *

*Shige Wang, Jeffrey R. Merrick, and Kang G. Shin*
*Real-Time Computing Laboratory*
*Department of Electrical Engineering and Computer Science*
*The University of Michigan*
*Ann Arbor, MI 48109-2122*
*email:{wangsg,jmerrick,kgshin}@eecs.umich.edu*

## Abstract

*Allocating software components while meeting multiple platform resource constraints is crucial for model-based design of large embedded real-time software and automatic design model transformation. In this paper, we propose a new method for component allocation using an informed branch-and-bound and forward checking mechanism subject to a combination of resource constraints. We have implemented this method in the Automatic Integration of Reusable Embedded Software (AIRES) toolkit — which has been developed under the DARPA MoBIES Program — and applied it to an automotive electronic throttle control (ETC) system. Our evaluation based on randomly-generated design models has shown that the proposed method scales well for large and complex embedded real-time software.*

## 1. Introduction

Software in today's large embedded real-time systems, such as avionics mission computing and automotive vehicle controls, demands a significant amount of system resource to meet increasing functional and performance requirements. However, the availability of resources on a platform is usually limited by physical and economic constraints. As embedded software becomes larger and more complex, meeting multiple, sometimes conflicting, resource constraints becomes a high-level design decision problem rather than a low-level code optimization. Traditional approaches that resolve the constraints individually, one-by-one, by tuning the code-level implementation are inadequate. Recent model-based software development has been shown to be a promising approach to such high-level design decision problems. In this model-based approach, the embedded software is first modeled abstractly without considering its execution platform, and then transformed to a model on the target platform. The component-to-platform allocation, therefore, becomes a critical step in the transformation. Further, the component-allocation method must be scalable as the number of components in embedded software can be very large. After completing the component allocation, one can form OS processes, assign timing and scheduling attributes to components, analyze the system performance and perform the schedulability test.

In this paper, we present a scalable method to allocate the software components in a design model to a given platform that meets multiple platform resource constraints. The method uses an informed branch-and-bound algorithm to find an allocation for each component in the software model. At each step, the candidate allocations are evaluated using a *competence function* and are pruned using a *forward checking mechanism*. Components are allocated to a device in the platform in the order of their combined resource consumptions. As the method manipulates the information at the model level, it is applicable during an early design phase and supports design automation.

The rest of the paper is organized as follows. Section 2 defines the system models and the component-allocation problem. Section 3 details the techniques used in our allocation method. Section 4 demonstrates the method using an example of automotive electronic throttle control system. Section 5 evaluates the scalability and performance of our method using a set of randomly-generated software models. Section 6 states related work. The paper concludes with Section 7.

## 2. Software Model and Problem Statement

Meeting the resource constraints during component allocation requires models of the software architecture and the platform. In this paper, we model the software archi-

*The work reported in this paper was supported in part by DARPA under the US AFRL contracts F30602-01-02-0527 and F3615-00-1706, and by Ford Motor Company under a University Research Partnership grant.


1080-1812/04 $ 20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

tecture in a *structural model*, which contains a set of existing components with known resource consumptions. The target platform with resource constraints is modeled in a *platform model*. For simplicity, we consider only computation, communication and memory resources, although the method can be extended to other resource types.

**Definition 1** *A software component $M_c = (A, I, O, B)$ is a port-based object, where A is a set of computations, called actions, which implement a component's functionality; I is a set of input ports through which a component receives its inputs; O is a set of output ports with $I \cap O = \emptyset$, through which a component exports its computation results; and $B \subseteq E^+ \times I \rightarrow A^+ \times O$ specifies a predefined behavior of the component where E defines a set of events.*

**Definition 2** *A software structural model $M_s = (C, L, F)$ is a weighted directed component graph, where*

- *C is a set of nodes for software components in $M_c$;*
- *$L \subseteq \bigcup_{u \in C} O_u \times \bigcup_{v \in C} I_v, (u \neq v)$ is a set of directed links from the output ports of some component(s) to the input ports of some other component(s); and*
- *F defines a set of resource consumption functions as:*

  *Computation: $F_c : \bigcup_{u \in C} A_u^+ \rightarrow Q^+$ defines the computation resource consumption by component u. $Q^+$ is the set of non-negative rational numbers. $A_u^+$ implements a behavior of component u.*

  *Communication: $F_l : L \rightarrow Q^+$ defines the communication resource consumption.*

  *Memory $F_m : C \rightarrow Q^+$ defines the memory resource consumption.*

According to this definition, the computation and memory resource consumptions are modeled as the nodes' weights, while the communication resource consumptions are modeled as the links' weights. These resource consumptions must be specified in a platform-independent form when used for component allocations. The platform-independent form of resource consumptions can be obtained by measuring resource consumptions on a reference platform and converting them to platform-independent values using techniques such as scalar [1] or virtual resource service rate [10]. The conversion function can be included as part of component's resource consumption function $F$. The structural model definition also allows cycles in the model in order to describe some control functions like closed-loop feedback and multi-rate control. Such cycles can be eliminated using the techniques in [10] for analyses that require a directed acyclic graph. In this paper, we do not perform the cycle elimination before component allocation since the existence of cycles does not affect the correctness and applicability of the allocation method.

**Definition 3** *A platform model $M_p = (P, N, R)$ is a weighted undirected graph, where*

- *P is a set of devices providing computation and memory resources;*
- *N is a shared communication link for all devices;*
- *R is a set of "availability" functions of the modeled resources defined as:*

  *Computation: $R_c : P \rightarrow Q^+$ defines the computation resource capacity of a device. Different $R_c$ values reflect the heterogeneity of computation devices.*

  *communication: $R_l : N \rightarrow Q^+$ defines the communication resource capacity of a link.*

  *Memory: $R_m : P \rightarrow Q^+$ defines the memory resource capacity of a device. Similar to the computation resource, different $R_m$ values reflect devices' memory heterogeneity.*

Given a structural model of designed software and a target platform model, the task of allocating components can be viewed as a process of grouping components in the structural model such that (i) each group runs on one computation device in the platform model with sufficient resources, and (ii) the total amount of communication among the groups is within the available capacity of the link in the platform model. We call each component group a *partition*, and the resultant model a *partition graph*.

**Definition 4** *The partition graph of a structural model $M_{pn} = (PN, LN, H)$ is a weighted directed graph, where*

- *PN is a set of partitions, containing components in $M_s$.*
- *$LN \subset PN \times PN$ is a set of links between partitions. There exists a link $(PN_i, PN_j) \in LN$ if and only if there are (i) components $u \in PN_i$ and $v \in PN_j$ and (ii) a link $(u, v) \in L$ in $M_s$.*
- *H is a set of resource consumption functions for the partition graph. For a partition $PN_i$, its resource consumptions of computation ($H_c$), communication ($H_l$), and memory ($H_m$) can be derived as follows:*

$$\begin{array}{l} H_c(PN_i) = \sum_{u \in PN_i} F_c(u), \\ H_l(LN) = \sum_{y \in LN} F_l(y), \\ H_m(PN_i) = \sum_{u \in PN_i} F_m(u) \end{array} \quad (1)$$

Our component-allocation problem can then be formally stated as a model transformation problem as follows: *given a structural model in $M_s = (C, L, F)$ and a platform model $M_p = (P, N, R)$, the component-allocation problem is to find a partition graph $M_{pn}$ with*


1080-1812/04 $ 20.00 © 2004 IEEE
IEEE COMPUTER SOCIETY

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

*Property 2: there exists a one-to-one mapping $\mathbf{g} : PN_i \rightarrow P_i$ such that (i) for any resource $r$ of device $P_i$, $H_r(PN_i) \leq R_r(P_i)$, and (ii) $H_l(LN) \leq R_l$.*

Property 1 ensures every component in the structural model to be allocated to one and only one partition. Property 2 constrains every partition to be allocated to a distinct device with sufficient computation and memory resources, and the total communication resource consumption among partitions to be no greater than the link's capacity. The one-to-one mapping implies that the total number of partitions should be no greater than the number of computation devices in the platform, i.e., $|PN| \leq |P|$.

In this paper, we ignore the execution dependencies of components when the structural model is partitioned. This makes the thus-generated results pessimistic. The real-time constraints such as end-to-end deadlines are addressed indirectly in this method by reducing the system resource consumptions of the resource that may introduce longer delays.

## 3. The Allocation Algorithm

Our component-allocation algorithm is shown in Algorithm 1, which subsequently invokes a recursive function in Algorithm 2. The algorithm is based on *informed branch-and-bound*. At each step, an unallocated component is assigned to a partition which runs on a distinct computation device (branch step). For every unallocated component $u$, the algorithm maintains a list of candidate partitions, called the *partition domain* $DM(u)$, containing all partitions that $u$ can be allocated to, without violating any constraint. All partitions $PN_i$ in $DM(u)$ are ranked according to a *competence function* $CF(u, PN_i)$. At each branch step, the algorithm selects the best partition in $DM(u)$ for allocation according to the competence function values (informed). After each assignment, the algorithm adjusts the partition domains of every unallocated component and eliminates those partitions that result in constraint violation (bound step). This is achieved by applying a *forward checking* [2] mechanism after a *minimum constraint violation depth* (MCVD) is reached. 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.

A key component in this algorithm is the *competence function*. It is designed and used to estimate the combined effect of an assignment and to speed up the algorithm. For a component $u$ and a partition $PN_i$ in $DM(u)$, $CF(u, PN_i)$ can be computed as follows:

**Algorithm 1** Allocation subject to multiple constraints.

**input:** structural model $M_s = (C, L, F)$;
  platform model $M_p = (P, N, R)$;
**output:** partition $PN$ containing components.
**BEGIN**
  $MCVD$ is initialized to be $\infty$;
  create partition $PN_i = \emptyset$ for $P_i \in P$;
  order $u \in C$ in descending order of the combinational resource consumption $w(u)$;
  **foreach** component $u \in C$ **do**
    assign $DM(u)$ to be $P$;
  **allocate**($C$);
  **if** ($C$ is empty) **then**
    **return** ($succeed, PN$);
  **else**
    **return** $fail$;
**END.**

$$CF(u, PN_i) = c_1 * C_r(PN_i) + c_2 * L_r(PN_i) + c_3 * M_r(PN_i), \quad (2)$$

where $C_r$, $L_r$ and $M_r$ are the computation, communication, and memory resource consumptions, respectively, after allocating $u$ to $PN_i$. Since the resource consumptions in $C_r(PN_i)$, $L_r(PN_i)$, and $M_r(PN_i)$ are of different types and usually are not comparable to each other, we normalize them as follows:

$$C_r(PN_i) = \frac{H_c(PN_i)}{C_{ideal}(PN_i)} - 1$$

$$L_r(PN_i) = 1 - \frac{H_l(LN)}{\sum_{y \in L} F_l(y)}$$

$$M_r(PN_i) = \frac{H_m(PN_i)}{M_{ideal}(PN_i)} - 1$$

$C_{ideal}(PN_i)$ is the ideal computation resource consumption of $PN_i$, 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 $u$ to a partition $PN_i$ with the minimum $C_r(PN_i)$ complies with the pre-defined strategy. Similarly, we can determine the ideal memory resource consumption $M_{ideal}(PN_i)$ of $PN_i$. The communication resource consumption of $PN_i$ depends on the total communication in and out of $PN_i$. A smaller value of $L_r(PN_i)$ indicates less communication resource consumed if allocating component $u$ to $PN_i$. Constants $c_1$, $c_2$ and $c_3$ 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 $c_1, c_2$, and $c_3$. For example, choosing $c_1 >> max(c2, c3)$ with $C_{ideal}(PN_i)$ derived from even distribution of the total


IEEE COMPUTER SOCIETY

**Algorithm 2** Allocate function

```
allocate(C)
BEGIN
  if (C is empty) then return (success, PN);
  retrieve first u from C for allocation;
  if (DM(u) is ∅) then return fail;
  foreach (PN_i ∈ DM(u)) do
    compute CF(u, PN_i);
  sort DM(u) in ascending order of CF(u, PN_i);
  while (u is unallocated) do
    if (all PN_i ∈ DM(u) have been tried) then
      put u back to C;
      return fail;
    end-if
    allocate u to PN_i with min{CF(u, PN_i)} that has not be tried;
    if (no constraint is violated by this allocation) then
      if (total allocated components = MCVD) then
        forward_checking(C);
      if (total allocated components > MCVD) then
        minimized_forward_checking(C);
      if (any component x ∈ C with DM(x) is empty ) then
        return fail;
      allocate(C);
      if (function return with success) then
        return (success, PN);
    else if (allocated components less than MCVD) then
        reset MCVD to the number of allocated components;
    end-if-else
  end-while
END
```

workload implements the load-balancing allocation strategy. Similarly, choosing $c_2 >> max(c1, c3)$ minimizes the communication resource consumption.

The algorithm uses the competence function values to assist selection of the possibly best 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.

Another key component in our algorithm is the *forward checking*, which is introduced to remove 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. The effect of removing partitions from a component's domain is local, meaning that the partition domain of a component $u$ will be restored to its original one if the allocation of $u$ or any component before $u$ is backtracked. This ensures the completeness of the algorithm, which guarantees a solution to be found if one exists.

The forward checking incurs additional computation overhead as it must check and manipulate the partition domains of all unallocated components after every allocation. 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 the forward checking can benefit only after allocating a certain number of components. We define such a number as a *minimum constraint violation depth* (MCVD). The value of MCVD is determined by the minimum number of allocated components at which a resource constraint violation occurs. 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. We implemented a minimized forward-checking function as in Algorithm 3.

**Algorithm 3** Minimized forward checking

```
input: unallocated components C with:
    partition domains PN for each u ∈ C;
    the most recently-assigned component v in PN_v;
    constraints on PN and LN.
output: reduced partition domains PN for each u ∈ C.
BEGIN
  foreach u ∈ C do
    if PN_v ∈ DM(u) and (F_c(u) + H_c(PN_v) > R_c(PN_v) or
        F_m(u) + H_m(PN_v) > R_m(PN_v)) then
        remove PN_v from DM(u);
    if Σ_{(y=(v,u)∨(u,v))∈LN} F_l(y) + H_l(LN) > R_l then
        remove all partitions but PN_v from DM(u);
  end-foreach
END.
```

It is shown in [4] 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 for initial allocation. The combined resource consumption by component $u$, $w(u)$, is computed as follows:

$$w(u) = a_1 * \frac{F_c(u)}{\sum R_c(P_i)} + a_2 * \frac{\sum_{y(u)} F_l(y)}{R_l(L)} + a_3 * \frac{F_m(u)}{\sum R_m(P_i)} \quad (3)$$

where $a_1, a_2$ and $a_3$ 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; $y(u)$ represents an incoming or outgoing communication link induced by component $u$.


1080-1812/04 $ 20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

In our algorithm, the components are allocated in descending order of their combinational resource consumptions, implying that a component requiring more resources be allocated first.

## 4. An Example: ETC Servo Control

To demonstrate its usefulness and effectiveness, the proposed algorithm is applied to an automotive electronic throttle control (ETC) system. The details of the ETC software model and its execution platform can be found in [5]. Due to space limitation, we present only a small part of one subsystem, servo control, with artificially-scaled resource consumptions. The structural model of the subsystem consists of components for *fault detection* (FD), *mode switch* (MS), *human control* (HC), *cruise control* (CC), *traction control* (TC), *limp home* (LH), *shutdown* (SHDN), and *merge* (MG), as shown in Figure 1 with labeled resource consumptions (Kilobytes for memory and Bytes for links). The platform consists of 2 processors, $P_1$ and $P_2$, connected via a shared link $L$. The available resources are 15 and 20 for computation, 20 KB for the memory on each processor, 10 KB/s for the communication link.[1]

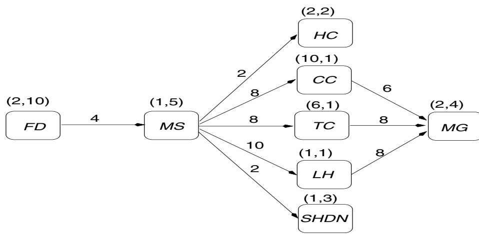


**Figure 1. The structural model of the servo controller.**

The components are first ordered according to their combined resource consumption $w(u)$ using Eq. (3) with all resources weighted equally $a_1 = a_2 = a_3 = 1$. The order of allocation is then $MS(w = 0.49)$, $CC(w = 0.45)$, $MG(w = 0.38)$, $TC(w = 0.36)$, $FD(w = 0.35)$, $LH(w = 0.23)$, $HC(w = 0.13)$, $SHDN(w = 0.12)$. Further, we would like to allocate the total workload proportionally to the processors' available computation resource, but equally on the memory resource, with $c_1 = c_2 = 10$ and $c_3 = 1$. Under this allocation strategy, we have $C_{ideal}(PN_1) = 10.71$, $M_{ideal}(PN_1) = 13.5$ and $C_{ideal}(PN_2) = 14.29$, $M_{ideal}(PN_2) = 13.5$, for partition $PN_1$ and $PN_2$, respectively. Figure 2 shows the allocation steps, the $CF$ values, and the partitions with their resource consumptions after each allocation.

Components are allocated individually, one-by-one, in descending order of their $w$ values. Initially, the partition domains of all components are assigned to be $(PN_1, PN_2)$ since every component consumes less resources than those available on each processor. As can be seen in Figure 2, we choose the partition with minimum $CF$ in an examined component's partition domain for allocation. At step 6, allocating $LH$ to $PN_2$ violates the memory constraint. So, $MCVD$ is reset to 6, and forward checking is performed. Since $PN_2$ has reached its maximum available memory, no other component can be allocated to it. The forward checking therefore removes $PN_2$ from the domains of all unallocated components — $LH$, $HC$, and $SHDN$. This makes $PN_1$ the only partition available for the unallocated components. Hence, $LH$, $HC$, and $SHDN$ are assigned to $PN_1$ immediately. The resultant partitions of the servo control subsystem are: the components $CC$, $LH$, $HC$ and $SHDN$ run on $P_1$; and $FD$, $MS$, $TC$ and $MG$ run on $P_2$. All resource constraints are met with this allocation.

Besides the algorithm process, the example also shows the effects of the constant selection in $w$ and $CF$. Ordering components for allocation with $a_1 = a_2 = a_3 = 1$ makes the $MS$ with the most communications allocated first instead of $CC$ with the most computation resource consumptions. Failure of the first attempt in allocating $LH$ at step 6 is caused by the small value of $c_3$ for memory in $CF$. Similarly, assigning an equal weight on computation and communication in $CF$ makes the total computation resource consumptions of $P_1$ and $P_2$ less proportional to their availability.

## 5. Evaluation

Our evaluation focuses on the scalability of the proposed component-allocation method. The metrics used in the evaluation include the number of nodes visited in the design space and the failure ratio. The number of nodes visited is obtained by counting all allocations explored. This implies the speed at which the algorithm finds a solution. The failure ratio is computed as the percentage of the number of experiments that the algorithm fails to find a solution over the total number of experiments conducted. To ensure all experiments complete within a reasonable time, we limit the algorithm's exploration to be 1 million nodes.

We experimented with a set of randomly-generated models. In each generated component graph, the number of components was selected from $100 \sim 1000$ in increments of 100. For each component, its link degree was randomly generated between 1 and 5, with computation and memory consumptions ranging from 0.005 to 0.05. The communication for each link was assigned randomly between 10 and 100. Similarly, the platform models were also generated randomly with the number of computation devices ranging from 5 to 50 in increments of 5. Each device has $0.6 \sim 1$ computation resource, and $0.3 \sim 1$ memory resource. The communication network has the capacity of

[1] In fact, the processors used for experiments have more resources than specified here. We limited them to show the effectiveness of our algorithm.


1080-1812/04 $ 20.00 © 2004 IEEE
IEEE COMPUTER SOCIETY

| steps | components | $CF:(PN_1,PN_2)$ | allocation | $PN_1/H_r$ | $PN_2/H_r$ | $H_l(LN)$ |
|---|---|---|---|---|---|---|
| 1 | assign $MS$ | $0.3, 0.07$ | $PN_2$ | $\emptyset/(0,0)$ | $\{MS\}/(1,5)$ | 0 |
| 2 | assign $CC$ | $8.83, 16.77$ | $PN_1$ | $\{CC\}/(10,1)$ | $\{MS\}/(1,5)$ | 8 |
| 3 | assign $MG$ | $10.41, -0.06$ | $PN_2$ | $\{CC\}/(10,1)$ | $\{MS,MG\}/(3,9)$ | 14 |
| 4 | assign $TC$ | $10.43, 4.06$ | $PN_2$ | $\{CC\}/(10,1)$ | $\{MS,TC,MG\}/(9,10)$ | 14 |
| 5 | assign $FD$ | $8.18, 4.71$ | $PN_2$ | $\{CC\}/(10,1)$ | $\{FD,MS,TC,MG\}/(11,20)$ | 14 |
| 6 | assign $LH$ | $5.41, 5.34$ | $PN_2$ | | memory constraint violation | |
| | | | $PN_1$ | $\{CC,LH\}/(11,2)$ | $\{FD,MS,TC,MG\}/(11,20)$ | 32 |
| 7 | assign $HC$ | | $PN_1$ | $\{CC,LH,HC\}/(13,4)$ | $\{FD,MS,TC,MG\}/(11,20)$ | 32 |
| 8 | assign $SHDN$ | | $PN_1$ | $\{CC,LH,HC,SHDN\}/(14,7)$ | $\{FD,MS,TC,MG\}/(11,20)$ | 36 |

**Figure 2. Steps for allocating the ETC servo control subsystem with multiple platform resource constraints.**

$100 \sim 1000$. For the purpose of comparison, we chose the standard branch-and-bound (BB) algorithm as a baseline. 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). In each case, we generated 3 structural+platform models, applied all selected algorithms for each model, and took the average as the results. We chose the load-balancing strategy for $C_{ideal}(PN_i)$ and $M_{ideal}(PN_i)$.

### 5.1. Performance while varying the model size

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 5. 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 5-processor platform. The generation was so tuned that not every allocation meets all resource constraints.

Figure 3 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. This implies that IBB+O+FC chose a proper allocation at its first trial, and hence, did not need to backtrack. Since IBB+O and IBB+O+FC both performed significantly better than BB but make 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. as the platform contains a small number of processors. 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, resulting in the increasing number of solutions and more likely to find a solution in fewer steps.

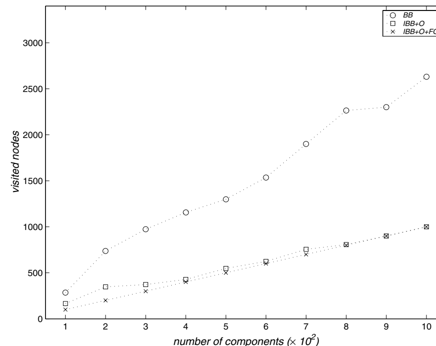


**Figure 3. The number of nodes visited for different model sizes.**

Figure 4 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. We observed that BB fails in more cases than IBB+O. 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 the specified number of steps. We also observed that the difference between failure ratios of different algorithms increases as the number of components increases. This may result from exploring only a fixed number of nodes in an expanding design space. The swings in the failure ratios for different graph sizes may also result from the randomness of the resource consumptions generated for each experiment.


IEEE COMPUTER SOCIETY

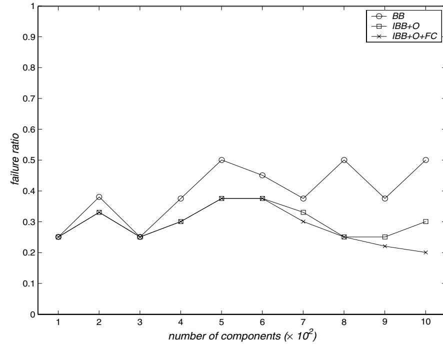


**Figure 4. The failure ratios of different algorithms.**

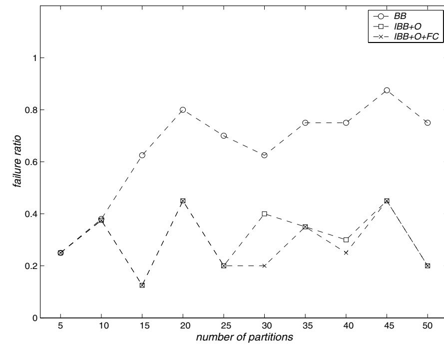


**Figure 6. The failure ratios of different algorithms.**

### 5.2. Performance while varying the number of devices in platform

The scalability of our algorithm is also affected by the number of devices in the platform. 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 find a solution within the exploration limit for the latter.

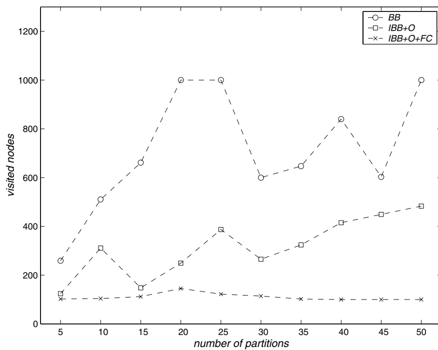


**Figure 5. The number of nodes visited with different numbers of devices.**

The results, shown in Figures 5 and 6, are similar to those for different model sizes. IBB+O+FC 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. This indicates that the forward checking improved the solution search by removing more partitions from a component's domain as the number of partitions increases. Similarly, BB experienced the largest failure ratio, and IBB+O+FC experienced the least. 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. Similarly, the swings of the failure ratios may also result from the randomness of the models in each experiment.

## 6. Related Work

Many approaches have been proposed to address multiple resource constraints during embedded software development [3, 7, 9]. Most of them rely on the assumption of independent resource constraints that can be handled individually, one at a time. Since meeting a constraint may potentially cause violation of other constraints, we need a solution that considers all constraints together. Q-RAM [8] is a framework supporting allocation multiple applications to multiple resources using a utility function of multiple QoS requirements. Although this approach is effective and scalable, the allocation in Q-RAM was concerned with how to provide sufficient resource of each type at the application level instead of considering the resource availability in individual devices and allocating them at the software component level. Neema *et al.* [6] developed and implemented a different approach based on OBDD in their tool called DESERT. However, the OBDD is unsuitable for modeling resource constraints. On the other hand, techniques for constrained search in the field of artificial intelligence, such as forward checking [2] and arc consistency check [4], can be used to improve the average-case performance of a guided exhaustive search.

Our work here differs from all of the above in that multiple resource constraints are addressed by allocating components to the platform using informed branch-and-bound, which is both scalable and complete. Our allocation is static and focuses on design-time decisions instead of runtime adaptiveness.


IEEE COMPUTER SOCIETY

## 7. Conclusions and Future Work

Allocating software components to a platform with multiple resource constraints is a crucial and challenging step in automatic model transformation for fast and low-cost development of embedded real-time software. We solved this problem by proposing a new method based on an informed branch-and-bound. Our algorithm uses a competence function to assist component allocation at each step, and uses a forward checking mechanism to reduce the partition domains of unallocated components. The overhead introduced by the forward checking is minimized by turning it on only after enough components have been allocated, and by examining only the partitions affected by the current allocation. Our evaluation results have shown that the algorithm is more scalable than others known to date. The method has been integrated into an embedded software design and analysis toolkit, called AIRES tool, which has been developed as part of the DARPA MoBIES Program, for automating the design of performance-constrained large embedded real-time software.

We have learned several lessons in this work. First, multiple resource constraints, sometimes inter-dependent and conflicting, must be considered as a whole, resulting in exponentially-increasing complexity. A simple technique that deals with one resource at a time, is therefore inadequate. Second, the accuracy of resource consumption estimations on a candidate target, particularly their relative values, has great impact on the accuracy of the final results since the estimations determine the effects of the weights in the competence function and combinational resource consumption. Finally, it is highly desirable and beneficial to build multiple allocation strategies in the design tool for a designer to choose the most suitable one, or to test "what-if" scenarios. Instead of implementing every strategy as a stand-alone function, we use a function to make the algorithm using different strategies by assigning different weights in this function. Such an approach not only simplifies the implementation, but also allows a combination of strategies.

Our future work includes improving the algorithm to deal with additional constraints and optimize the allocation results. Although the algorithm can be used to meet various types of resource constraints such as power, size, and cost, in addition to those treated in this paper, it would be challenging and interesting to determine the competence functions with such heterogeneous resource types. Particularly, there may exist resources whose consumptions cannot be modeled as a simple function. Additional criteria and mechanisms may be required to handle other types of resource constraints. We would also like to investigate and improve the optimality of the generated results. A key issue is to define a (possibly utility-based) function that can be used to evaluate the optimality involving multiple resource types.

## References


[1] 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.

[2] 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.

[3] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-chip. In *Proceedings of the 22nd Real-Time Systems Symposium (RTSS 2001)*, pages 73–83, December 2001.

[4] V. Kumar. Algorithms for constraint satisfaction problems: a survey. *A.I. Magazine*, 13(1):32–44, 1992.

[5] MoBIES Auotmotive OEP. Platform mathematical models and controller code. http://vehicle.me.berkeley.edu/mobies/, 2002.

[6] S. Neema, J. Sztipanovits, G. Karsai, and K. Butts. Constraint-based design-space exploration and model synthesis. In *Proceedings of the 3rd International Conference on Embedded Software (EMSOFT'03)*, volume 2855, pages 290–305, Philadelphia, PA, Octorber 2003.

[7] B. C. Neuman and S. Rao. Resource management for distributed parallel systems. In *Proceedings of the 2nd International Symposium on High Performance Distributed Computing*, pages 316–323, July 1993.

[8] R. Rajkummar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek. Practical solutions for QoS-based resource allocation problems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS'98)*, pages 296–306, Madrid, Spain, December 1998.

[9] R. Ramamoorthi, A. Rifkin, B. Dimitrov, and K. M. C. Butts. A general resource reservation framework for scientific computing. In *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE) Conference*, volume 1343, pages 283–290, December 1997.

[10] S. Wang and K. G. Shin. Early-stage performance modeling and its application for integrated embedded constrol software design. In *accepted by 2004 ACM Workshop on Software Performance (WOSP)*, Redwood, CA., January 2004.




1080-1812/04 $ 20.00 © 2004 IEEE



IEEE COMPUTER SOCIETY