Optimizing Resource Usage in Component-Based Real-Time Systems

Johan Fredriksson, Kristian Sandström, and Mikael Åkerholm

Mälardalen Real-Time Research Centre,
Department of Computer Science and Engineering,
Mälardalen University, Box 883, Västerås, Sweden,
http://www.mrtc.mdh.se
johan.fredriksson@mdh.se

Abstract. The embedded systems domain represents a class of systems that have high requirements on cost efficiency as well as run-time properties such as timeliness and dependability. The research on component-based systems has produced component technologies for guaranteeing real-time properties. However, the issue of saving resources by allocating several components to real-time tasks has gained little focus. Trade-offs when allocating components to tasks are, e.g., CPU-overhead, footprint and integrity. In this paper we present a general approach for allocating components to real-time tasks, while utilizing existing real-time analysis to ensure a feasible allocation. We demonstrate that CPU-overhead and memory consumption can be reduced by as much as 48% and 32% respectively for industrially representative systems.

1 Introduction

Many real-time systems (RTS) have high requirements on safety, reliability and availability. Furthermore the development of embedded systems is often sensitive to system resource usage in terms of, e.g., memory consumption and processing power. Historically, to guarantee full control over the system behaviour, the development of embedded systems has been done using only low level programming. However, as the complexity and the amount of functionality implemented by software increase, so does the cost for software development. Also, since product lines are common within the domain, issues of commonality and reuse are central for reducing cost. Component-Based Development (CBD) has shown to be an efficient and promising approach for software development, enabling well defined software architectures as well as reuse. Hence, CBD can be used to achieve goals such as cost reduction, and quality and reliability improvements.

In embedded RTS timing is important, and scheduling is used to create predictable timing. Furthermore, these systems are often resource constrained; consequently memory consumption and CPU load are desired to be low. A problem in current component-based embedded software development practices is the allocation of components to run-time tasks [1]. Because of the real-time requirements on most embedded systems, it is vital that the allocation considers

temporal attributes, such as worst case execution time (WCET), deadline (D) and period time (T). Hence, to facilitate scheduling, components are often allocated to tasks in a one-to-one fashion. However, for many embedded systems it is desired to optimize for memory and speed [2], thus the one-to-one allocation is unnecessarily memory and CPU consuming.

Embedded RTS consist of periodic and sporadic events that usually have end-to-end timing requirements. Components triggered by the same periodic event can often be coordinated and executed by the same task, while still preserving temporal constraints. Thus, it is easy to understand that there can be profits from allocating several components into one task. Some of the benefits are less memory consumption in terms of stacks and task control blocks or lower CPU utilization due to less overhead for context switches. Different properties can be accentuated depending on how components are allocated to tasks, e.g., memory usage and performance; Hence, there are many trade-offs to be made when allocating components to tasks.

Allocating components to tasks, and scheduling tasks are both complex problems and different approaches are used. Simulated annealing and genetic algorithms are examples of algorithms that are frequently used for optimization problems. However, to be able to use such algorithms, a framework to calculate properties, such as memory consumption and CPU-overhead, is needed. The work presented in this paper describes a general framework for reasoning about trade-offs concerning allocating components to tasks, while preserving extra-functional requirements. Temporal constraints are verified and the allocations are optimized for low memory consumption and CPU-overhead. The framework is evaluated using industrially relevant component assemblies, and the results show that CPU-overhead and memory consumption can be reduced by as much as 48% and 32% respectively.

The idea of assigning components to tasks for embedded systems while considering extra-functional properties and resource utilization is a relatively uncovered area. In [3, 4] Bondarev et. al. are looking at predicting and simulating real-time properties on component assemblies. However, there is no focus on increasing resource utilization through component to task allocation. The problem of allocating tasks to different nodes is a problem that has been studied by researchers using different methods [5, 6]. There are also methods proposed for transforming structural models to run-time models [7, 8, 1], but extra-functional properties are usually ignored or considered as non-critical [9]. In [10], an architecture for embedded systems is proposed, and it is identified that components has to be allocated to tasks, however there is no focus on the allocation of components to tasks. In [9] the authors propose a model transformation where all components with the same priority are allocated to the same task; however no consideration is taken to lower resource usage. In [11], the authors discuss how to minimize memory consumption in real-time task sets, though it is not in the context of allocating components to tasks. Shin et. al [12] are discussing the code size, and how it can be minimized, but does not regard scheduling and resource constraints.

The outline for the rest of the paper is as follows; section 2 gives an overview of the component to task allocations, and describes the structure of the components and tasks. Section 3 describes a framework for calculating the properties of components allocated to tasks. Section 4 discusses allocation and scheduling approaches, while evaluations and simulations are presented in section 5. Finally in section 6, future work is discussed and the paper is concluded. Detailed data regarding the simulations can be found in [13].

2 Allocating components to real-time tasks

In RTS temporal constraints are of great importance and tasks control the execution of software. Hence, components need to be allocated to tasks in such a way that temporal requirements are met, and resource usage is minimized. Given an allocation we determine if it is feasible and calculate the memory consumption and task switch overhead. To impose timing constraints, we define end-to-end timing requirements and denote them transactions. Transactions are defined by a sequence of components and a deadline. Thus, the work in this paper has three main concerns:

1. Verification of allocations from components to tasks.
2. Calculating system properties for an allocation
3. Minimizing resource utilization

CBSE is generally not used when developing embedded RTS, mostly due to the lack of efficient mappings to run-time systems and real-time properties. One approach that allows an efficient mapping from components to a RTS is the Autocomp technology [14]. An overview of the Autocomp technology can be seen in Fig 1. The different steps in the figure are divided into design-time, compile-time, and run-time to display at which point in time during development they are addressed or used. The compile-time steps, illustrated in Fig 1, incorporate an allocation from the component-based design, to a real-time model and mapping to a real-time operating system (RTOS). During this step the components are allocated to real-time tasks and the component requirements are mapped to task-level attributes.

By combining the notion of transactions and the pipe-and-filter interaction model we get a general component model that is easy to implement for a large set of component technologies for embedded systems such as Autocomp [14], SaveCCM [15], Rubus [16], Koala [17], Port-based objects [18], IEC61131[19] and Simulink[20]. The component model characteristics are described in the section 2.1 and the task model characteristics are described in section 2.2.

2.1 Component model characteristics

In this section we describe characteristics for a general component model that is applicable to a large set of embedded component models. Both component and task models described are meta-models for modelling the most important

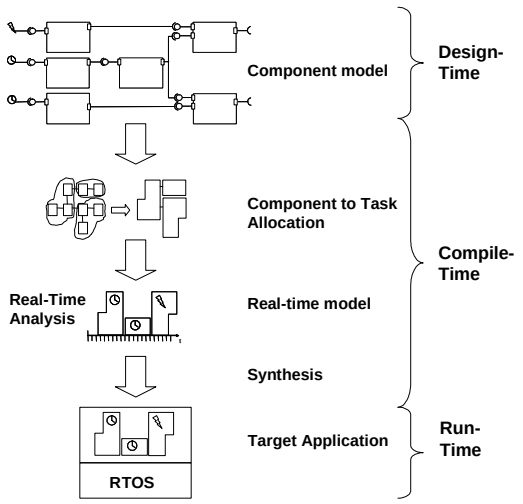

Fig. 1. Autocomp system description

attributes of an allocation between components and tasks. The component interaction model used throughout this paper is a pipe-and-filter model with transactions. Each component has a trigger; a time trigger or an event trigger or a trigger from a preceding component. A component transaction describes an order of components and defines an end-to-end timing requirement. In Fig 2, the notation of a component assembly with six components and four transactions is described. The graphical notation is similar to the one used in UML.

The component model chosen is relatively straight forward to analyse and verify. The pipe-and-filter interaction model is commonly used within the embedded systems domain. Many component models for embedded systems have the notion of transactions built in; however, if a component model lacks the notion of transactions, there are often possibilities to model end-to-end timing requirements and execution order at a higher abstraction level. In general a system is described with components, component relations, and transactions (flow) between components. The component model is described with:

Component c_i is described with the tuple $< S_i, Q_i, X_i, M_i >$, where S_i is a signal from another component, an external event or a timed event. Q_i represents the *minimum inter arrival time* (MINT) in the case of an external event. It represents the period in the case of a timed trigger and it is unused if the signal is from another component. The parameter X_i is the WCET for the component, and M_i is the amount of stack required by the component.

Isolation set I defines a relation between components that should not be allocated. It is described with a set of component pairs $I =< (c_1, c_2), (c_3, c_4) >$ that define what components may not be allocated to the same task. There may be memory protection requirements or other legitimate engineering reasons to avoid allocating certain combinations of components; for example,

if a component has a highly uncertain WCET. The isolation set is indexed with subscripts denoting next inner element, i.e., $I_1 = (c_1, c_2)$ and $I_{12} = c_2$.

Component Transaction ctr_i is an ordered relation between components $N_i = c_1, c_2, ..., cn$, and an end-to-end deadline dc_i. The deadline is relative to the event that triggered the component transaction, and the first component within a transaction defines the transaction trigger. A component transaction can stretch over one or several components, and a component can participate in several component transactions. The component c_a should execute before the component c_b, and the component c_b should execute before c_c to produce the expected results etc. The correct execution behaviour for the set $N = c_1, c_2, ..., c_n$ can be formalized with the regular expression denoted in 1.

$$c_1 \Sigma^* c_2 \Sigma^* ... c_n \tag{1}$$

Where Σ^* denotes all allowed elements defined by N.

In a component assembly, event triggers are treated different from the periodic triggers as the former is not strictly periodic. There is only a lower boundary restricting how often it can occur, but there is no upper bound restricting how much time may elapse between two invocations. Thus, if an event trigger could exist inside or last in a transaction, it would be impossible to calculate the response time for the transaction, and hence a deadline could never be guaranteed.

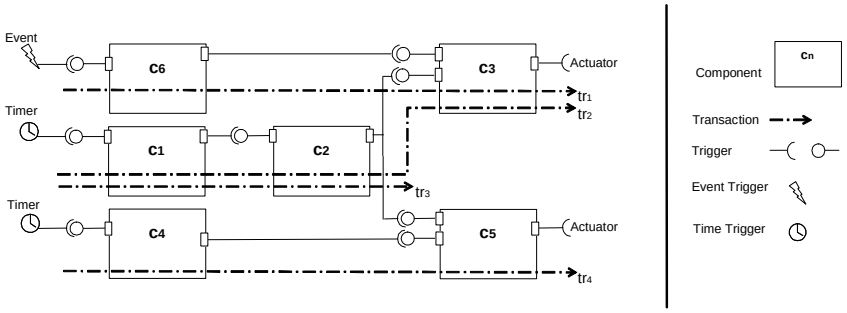

Fig. 2. Graphical notation of the component model.

2.2 Task characteristics

The task model specifies the organization of entities in the component model into tasks and transactions over tasks. During the transformation from component model to run-time model, extra-functional properties like schedulability and response-time constraints must be considered in order to ensure the correctness of the final system. Components only interact through explicit interfaces; hence tasks do not synchronize outside the component model. The task model is for evaluating schedulability and other properties of a system, and is similar to standard task graphs as used in scheduling theory, augmented with exclusion constraints (isolation). The task model is described with:

System K is described with the tuple $< A, \tau, \rho >$ where A is a task set scheduled by the system. The constant τ is the size of each task control block, and can be considered constant and the same for all tasks. The constant ρ is the time associated with a task switch. The system kernel is the only explicitly shared resource between tasks; hence we do not consider blocking. Also blocking is not the focus of this paper.

Task t_i is described with the tuple $< C_i, T_i, wcet_i, stack_i >$ where C_i is an ordered set of components. Components within a task are executed in sequence. Components within a task are executed at the same priority as the task, and a high priority task pre-empts a low priority task. T_i is the period or minimum inter arrival time of the task. The parameters $wcet_i$ and $stack_i$ are worst case execution time and stack size respectively. The $wcet_i$, $stack_i$ and period (T_i) are deduced from the components in C_i. The $wcet_i$ is the sum of all the WCETs for all components allocated to the task. Hence, for a task t_i, the parameters $wcet_i$ and $stack_i$ are calculated with (2) and (3) .

$$wcet_n = \sum_{\forall_i (c_i \in C_n)} (X_i) \tag{2}$$

$$stack_n = \forall_i (c_i \in C_n) max(M_i) \tag{3}$$

Task transaction ttr_i is a sequence of tasks $O_i = t_1, t_2, ..., t_k$ and a relative deadline dt_i. O_i defines an ordered relation between the tasks, where in the case of $O = t_1, t_2$; t_1 is predecessor to t_2. The timing and execution order requirements of a task transaction ttr_i are deduced from the requirements of the component transactions ctr_i. The task transaction ttr_i has the same parameter as the component transactions ctr_i but t_1, t_2,..., t_k are the tasks that map the component $c_a, c_b, ..., c_n$, as denoted in Fig 4. If several task transactions ttr_i span over the exact same tasks, the transactions are merged and assigned the shortest deadline. An event-triggered task may only appear first in a transaction. Two tasks can execute in an order not defined by the transactions. This depends on that the tasks have different period times, and thereby suffer from period phasing; hence transactions can not define a strict precedence relation between two tasks. Fig 3 is an execution trace that shows the relation between tasks and transactions. The tasks and transactions are the same as in Fig 4, left part.

3 Allocation framework

The allocation framework is a set of models for calculating properties of allocations of components to tasks. The properties calculated with the framework are used for optimization algorithms to find feasible allocations that fulfil given requirements on memory consumption and CPU-overhead.

For a task set A that has been mapped from components in a one-to-one fashion, it is trivial to calculate the system memory consumption and CPU-overhead since each task has the same properties as the basic component. When several

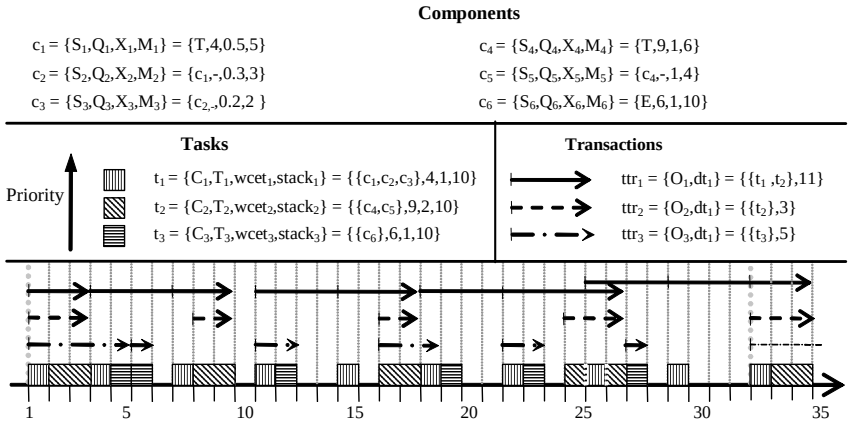

Fig. 3. Task execution order and task transactions.

components are allocated to one task we need to calculate the appropriateness of the allocation and the tasks properties. For a set of components, c_1,...,c_n, allocated to a set of tasks A, the following properties are considered.

- CPU-overhead p_A
- Memory consumption m_A

Each component c_i has a memory consumption stack. The stack of the task is the maximum size of all components stacks allocated to the task since all components will use the same stack. The CPU overhead p, the memory consumption m for a task set A in a system K are formalized in equations 4 and 5:

$$p_A = \sum_{\forall_i (t_i \in A)} \frac{\rho}{T_i} \tag{4}$$

$$m_A = \sum_{\forall_i (t_i \in A)} (stack_i + \tau) \tag{5}$$

Where p_A represents the sum of the task switch overhead divided by the period for all tasks is the system, and m_A represents the total amount of memory used for stacks and task control blocks for all tasks in the system

3.1 Constraints on allocations

There is a set of constraints that must be considered when allocating components. These are:

- Component isolation
- Intersecting transactions
- Trigger types and period times
- Schedulability

Each constraint is further discussed below:

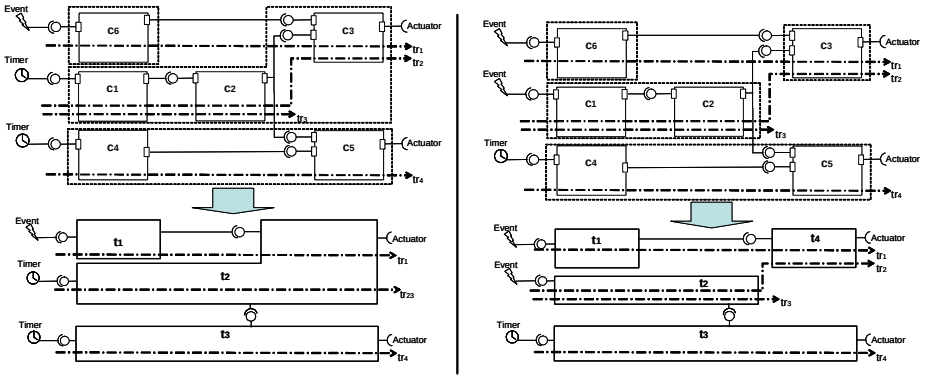

Fig. 4. Two allocations from components to tasks dependent on intersecting transactions.

Isolation It is not realistic to expect that components can be allocated in an arbitrary way. There may be explicit dependencies that prohibits that certain components are allocated together, therefore the isolation set I defines which components may not be allocated together. There may be specific engineering reasons to why some components should be separated. For instance, it may be desired to minimize the jitter for some tasks, thus components with highly uncertain WCET should be isolated. There may also be integrity reasons to separate certain combinations of components. Hence it must be assured that two components that are defined to be isolated do not reside in the same task. This can be validated with equation 6:

$$Iso(a,b) : c_a \text{ has an isolation requirement to } c_b$$

$$\neg\exists_i(\forall_j\forall_k(c_j \in C_i \wedge c_k \in C_i \wedge Iso(j,k))) \tag{6}$$

Where there must not exist any task t_i that has two components c_j and c_k, if these components have an isolation requirement.

Intersecting transactions If component transactions intersect, there are different strategies for how to allocate the component where the transactions intersect. The feasibility is described in equations 7 and 8. A component in the intersection should not be allocated with any preceding component if both transactions are event triggered; the task should be triggered by both transactions to avoid pessimistic scheduling. A component in the intersection of one time-triggered transaction and one event-triggered transaction can be allocated to a separate task, or with a preceding task in the time-triggered transaction. A component in the intersection of two time-triggered transactions can be allocated arbitrarily. In Fig 4, two different allocations are imposed due to intersecting event-triggered transactions. In the left part of Fig 4 there is an intersection between a time triggered and an event triggered transaction. Then the intersecting

component c_3 is allocated to the task triggered by the time triggered transaction. In the right part of the figure, where two event triggered transactions intersect, the component c_3 is allocated to a separate task, triggered by both transactions.

$$T_E(tr) : \text{transaction is event triggered}$$
$$T_T(tr) : \text{transaction is time triggered}$$
$$P(a,b,d) : c_a \text{ is predecessor to } c_b \text{ in the set } N_d$$
$$X_a^{bc} = c_a \in N_b \wedge c_a \in N_c$$
$$Y_{ab}^{c} = c_a \in C_c \wedge c_b \in C_c$$

$$\neg\exists_i(\forall_j\forall_k\forall_l\forall_m(X_l^{jk} \wedge Y_{lm}^{i} \wedge T_E(ctr_j) \wedge T_E(ctr_k) \wedge (P(m,l,k) \vee P(m,l,j)))) \quad (7)$$
$$\neg\exists_i(\forall_j\forall_k\forall_l\forall_m(X_l^{jk} \wedge Y_{lm}^{i} \wedge c_m \in N_k \wedge T_T(ctr_j) \wedge T_E(ctr_k) \wedge P(c_m, c_l, N_k))) \quad (8)$$

Where there must not exist any task t_i that has two components c_l and c_m in a way that two component transactions ctr_j and ctr_k intersect in c_l, and c_m precedes c_l in the transactions ctr_j or ctr_k, if ctr_j or ctr_k are event-triggered.

Triggers Some allocations from components to tasks can be performed without impacting the schedulability negatively. A component that triggers a subsequent component can be allocated into a task if it has no other explicit dependencies, see (1) in Fig 5. Components with the same period time can be allocated together if they do not have any other explicit dependencies, see (2) in Fig 5. To facilitate analysis, a task may only have one trigger, so time triggered components with the same period can be triggered by the same trigger and thus allocated to the same task. However, event triggered components may only be allocated to the same task if they in fact trigger on the same event, and have the same minimum inter arrival time, see (3) in Fig 5. Components with harmonic periods could also be allocated to the same task. However, harmonic periods create jitter. Consider two components with the harmonic periods five and ten that are allocated to one task. The component with the period five will run every invocation, while the other component will run every second invocation, which creates a jitter; therefore we have chosen not to pursue this specific issue.

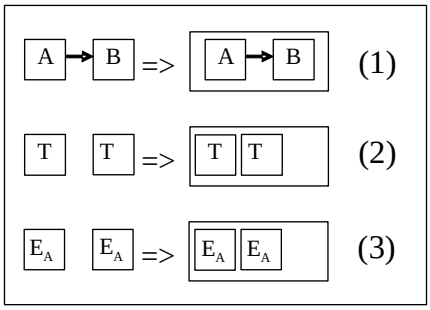

Fig. 5. Component to task allocation considering triggers.

Schedulability Schedulability analysis is highly dependent on the scheduling policy chosen. Depending on the system design, different analyses approaches have to be considered. The task and task transaction meta-models are constructed to fit different scheduling analyses. In this work we have used fixed priority exact analysis. However, the model can easily be extended with jitter and blocking for real-time analysis models that use those properties. The framework assigns each task a unique priority pre run-time, and it uses exact analysis for schedulability analysis, together with the Bate and Burns [21] approach for verifying that the transaction deadlines are met.

4 Using the framework

An allocation can be performed in several different ways. In a small system all possible allocations can be evaluated and the best chosen. For a larger system, however, this is not possible due to the combinatorial explosion. Different algorithms can be used to find a feasible allocation and scheduling of tasks. For any algorithm to work there must be some way to evaluate an allocation. The proposed allocation framework can be used to calculate schedulability, CPU-overhead and total memory load. The worst-case allocation is a one-to-one allocation where every component is allocated to one task. The best-case allocation on the other hand, is where all components are allocated to one single task. To allocate all components to one task is very seldom feasible. Also, excessive allocation of components may negatively affect scheduling, because the granularity is coarsened and thereby the flexibility for the scheduler is reduced.

Simulated annealing, genetic algorithms and bin packing are well known algorithms often used for optimization problems. These algorithms have been used for problems similar to those described in this paper; bin packing, e.g., has been proposed in [22] for real-time scheduling. Here we briefly discuss how theses algorithms can be used with the described framework, to perform component to task allocations.

Bin Packing is a method well suited for our framework. In [23] a bin packing model that handles arbitrary conflicts (BPAC) is presented. The BPAC model constrains certain elements from being packed into the same bin, which directly can be used in our model as the isolation set I, and the bin-packing feasibility function is the schedulability.

Genetic algorithms can solve, roughly, any problem as long as there is some way of comparing two solutions. The framework proposed in this paper give the possibility to use the properties memory consumption, CPU-overhead and schedulability as grades for an allocation. In, e.g., [24] and [25], genetic algorithms are used for scheduling complex task sets and scheduling task sets in distributed systems.

Simulated annealing (SA) is a global optimization technique that is regularly used for solving NP-Hard problems. The energy function consists of a schedulability test, the memory consumption and CPU-overhead. In [6][26] simulated annealing is used to place tasks on nodes in distributed systems.

5 Evaluation

In order to evaluate the performance of the allocation approach the framework has been implemented. We have chosen to perform a set of allocations and compare the results to a corresponding one-to-one allocation where each component is allocated to a task. We compare the allocations with respect to if the allocation is feasible (real-time analysis), memory consumption and CPU overhead.

The implementation is based on genetic algorithms (GA) [27], and as Fig 6 shows, each gene represents a component and contains a reference to the task it is assigned. Each chromosome represents the entire system with all components assigned to tasks. Each allocation produced by the GA is evaluated by the framework, and is given a fitness value dependent on the validity of the allocation, the memory consumption and the CPU overhead.

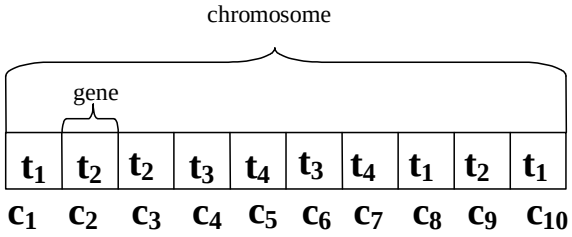

Fig. 6. The genetic algorithm view of the component to task allocation; a system with ten components, allocated to four tasks.

5.1 Fitness function

The fitness function is based on the feasibility of the allocation together with the memory consumption and CPU overhead. The feasibility part of the fitness function is mandatory, i.e., the fitness value for a low memory and CPU overhead can never exceed the value for a feasible allocation. The feasibility function consists of: I which represents component isolation, IT representing intersecting transactions, Tr representing trigger types and period times, and finally Sc represent scheduling. Consider that each of these feasibility tests is assigned a value greater than 1 if they are true, and a value of 0 if they are false. The parameter n represents the total number of components. Then, the fitness function can be described as with equation 9.

$$Fitness = \Big((I + IT + Tr + Sc)F + \big(\frac{n}{m_A} + \sum_{\forall i(t_i \in A)} \frac{\rho \cdot n}{T_i}\big)O\Big) \cdot (I \cdot IT \cdot Tr \cdot Sc + 1) \quad (9)$$

Where the fitness is the sum of all feasibility values times a factor F, added with the inverted memory usage and performance overhead, times a factor O, and $F >> O$. The total fitness is multiplies with 1 if any feasibility test fail, and the products of all feasibility values plus 1 if all feasibility tests succeed.

5.2 Simulation set up

This section describes the simulation method and set up. For each simulation the genetic algorithm assigns components to tasks and evaluates the allocation, and incrementally finds new allocations.

The system data is produced by creating a random schedulable task set, on which all components are randomly allocated. The component properties are deduced from the task they are allocated. Transactions are deduced the same way from the task set. In this way it is always at least one solution for each system. However, it is not sure that all systems are solvable with a one-to-one allocation. The components and component transactions are used as input to the framework. Hereafter, systems that are referred to as generated systems are generated to form input to the framework. Systems that come out of the framework are referred to as allocated systems. The simulation parameters are set up as follows:

- The number of components of a system is randomly selected from a number of predefined sets. The numbers of components in the systems are ranging in twenty steps from 40 to 400, with a main point on 120 components.
- The period times for the components are randomly selected from a predefined set of different periods between 10 and 100 ms.
- The worst case execution time (WCET) is specified as a percentage of the period time and chosen from a predefined set. The WCETs together with the periods in the system constitutes the system load.
- The transaction size is the size of the generated transactions in percentage of the number of components in the system. The transaction size is randomly chosen from a predefined set. The longer the transactions, the more constraints, regarding schedulability, on how components may be allocated.
- The transaction deadline laxity is the percentage of the lowest possible transaction deadline for the generated system. The transaction deadline laxity is evenly distributed among all generated systems and is always greater or equal to one, to guarantee that the generated system is possible to map. The higher the laxity, the less constrained transaction deadlines.

One component can be involved in more than one transaction, resulting in more constraints in terms of timing. The probability that a component is participating in two transactions is set to 50% for all systems.

To get as realistic systems to simulate as possible, the values used to generate systems are gathered from some of our industrial partners. The industrial partners chosen are active within the vehicular embedded system segment. A complete table with all values and distributions, of the system generation values, can be found in [13]. The task switch time used for the system is 22 μs, and the tcb size is 300 bytes. The task switch time and tcb size are representative of commercial RTOS tcb sizes and context switch times for common CPUs.

The simulations are performed for four different utilization levels, 30%, 50%, 70% and 90%. For each level of utilization 1000 different systems are generated with the parameters presented above.

5.3 Results

A series of simulations have been carried out to evaluate the performance of the proposed framework. To evaluate the schedulability of the systems, FPS scheduling analysis is used. The priorities are randomly assigned by the genetic algorithm, and no two tasks have the same priority. We compare the allocation approach described in this paper to one-to-one allocations. Table 5.3 summarizes the results from the simulations. The columns entitled "stack" and "CPU" displays the average memory size (stack + tcb) and CPU overhead respectively, for all systems with a specific load and transaction deadline laxity. The column entitled "success" in the 1-1 allocation section displays the rate of systems that are solvable with the 1-1 allocation. The column entitled "success" in the GA allocation section displays the rate at which our framework finds allocations, since all systems has at least one solution. The stack and CPU values are only collected from systems where a solution was found.

Load	Laxity	1-1 allocation			GA allocation		
		Stack	CPU	success	stack	CPU	success
30%	*All*	*28882*	*4,1%*	*74%*	*17380*	*2,0%*	*87%*
	1.1	25949	3,5%	39%	14970	1,6%	58%
	1.3	33077	4,4%	78%	21005	2,2%	97%
	1.5	26755	4,1%	95%	15503	2,0%	99%
50%	*All*	*37277*	*4,8%*	*82%*	*24297*	*2,4%*	*90%*
	1.1	35391	4,3%	49%	23146	2,3%	64%
	1.3	38251	4,8%	88%	25350	2,5%	96%
	1.5	37043	4,9%	98%	23740	2,3%	100%
70%	*All*	*44455*	*5,1%*	*85%*	*30694*	*2,7%*	*91%*
	1.1	44226	5,0%	58%	31638	2,7%	73%
	1.3	44267	5,1%	94%	30686	2,7%	98%
	1.5	44619	5,2%	98%	30232	2,6%	100%
90%	*All*	*46943*	*5,6%*	*87%*	*37733*	*3,1%*	*93%*
	1.1	54858	5,7%	65%	41207	3,4%	80%
	1.3	49607	5,5%	92%	35470	3,0%	98%
	1.5	53535	5,7%	98%	38260	3,1%	99%

Table 1. Memory, CPU overhead and success ratio for 1-1 and GA allocations

The first graph for the simulations (Fig 7) shows the success ratio, i.e., the percentage of systems that were possible to map with the one-to-one allocation, and the GA allocation respectively. The success ratio is relative to the effort of the GA, and is expected to increase with a higher number of generations for each system. Something that might seem confusing is that the success ratio is lower for low utilization than for high utilizations, even though, intuitively, it should be the opposite. The explanation to this phenomenon is that the timing constraints become tighter as fewer tasks participate in each transaction (lower utilization often leads to fewer tasks). With fewer tasks the task phasing, due to different periods, will be lower, and the deadline can be set tighter.

The second graph (Fig 8) shows that the deadlines are relaxed with higher utilization, since the allocations with relaxed deadlines perform well, and the

systems with a more constrained deadline show a clear improvement with higher utilization.

The third graph (Fig 9) shows for both approaches the average stack size for the systems at different utilization. The comparison is only amongst allocations that are have been successfully mapped by both strategies. The memory size consists of the tcb and the stack size, and the tcb size is 300 bytes. As described earlier, each task allocates a stack that is equal to the size of the largest stack among its allocated components.

The fourth graph (Fig 10) shows the average task switch time in micro seconds for the entire system. The task switch overhead is only dependent on how many tasks there are in the system. The average improvement of GA allocation in comparison to the 1-1 allocation is, for the success ratio, 10%. The memory size is reduced by 32%, and the task switch overhead is reduced by 48%. Hence we can see a substantial improvement in using smart methods to map components to tasks. A better strategy for setting priorities would probably lead to an improvement in the success ratio. Further we observe that lower utilization admits larger improvements than higher laxity of the deadlines; and since lower utilization in the simulations often gives tighter deadlines, we can conclude that the allocation does not negatively impact schedulability. However, regarding the improvements, the more components the more constrains are put on each transaction, and thereby on the components, making it harder to perform *good* allocations.

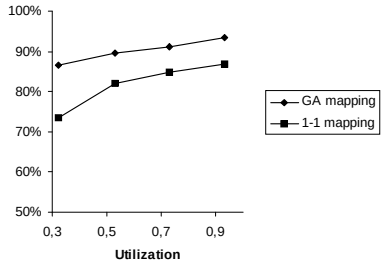

Fig. 7. Average success ratio

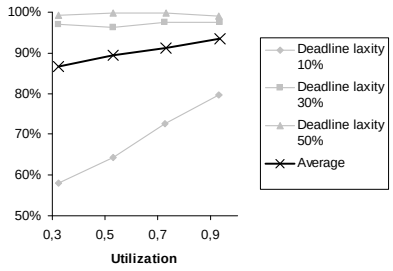

Fig. 8. Success rate for allocations

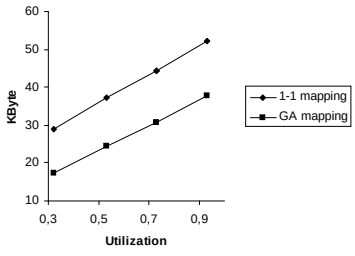

Fig. 9. Average memory size

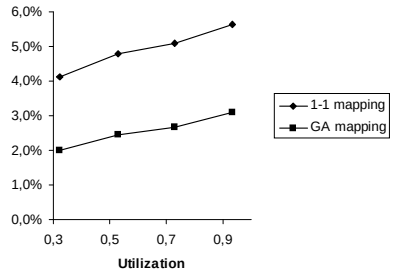

Fig. 10. Average task switch overhead

6 Conclusions and Future Work

Resource efficiency is important for RTS, both regarding performance and memory. Schedulability, considering resource efficiency, has gained much focus, however the allocation between components to tasks has gained very little focus. Hence, in this paper we have described an allocation framework for allocating components to tasks, to facilitate existing scheduling and optimization algorithms such as genetic algorithms, bin packing and simulated annealing. The framework is designed to be used during compile-time to minimize resource usage and maximize timeliness. It can also be used iteratively in case of design changes; however with some obvious drawbacks on the results. The framework can easily be extended to support other optimizations, besides task switch overhead and memory consumption. Results from simulations show that the framework gives substantial improvements both in terms of memory consumption and task switch overhead. The described framework also has a high ratio in finding feasible allocations. Moreover, in comparison to allocations performed with a one-to-one allocation our framework performs very well, with 32% reduced memory size and 48% reduced task switch overhead. The simulations show that the proposed framework performs allocations on systems of a size that covers many embedded systems, and in a reasonable time for an off-line tool. We have also shown how CPU load and deadline laxity affects the allocation. Future work includes adding other allocation criteria, e.g., by looking at jitter requirements, and blocking. By adding jitter constraints and blocking, trade-offs arise between switch overhead and memory size versus deviation from nominal start and end times and blocking times. Furthermore, a more efficient scheduling policy and priority assignment will be applied. Due to the nature of GA it is easy to add new optimizations as the ones suggested above.

References

1. Mills., K., Gomaa, H.: Knowledge-based automation of a design method for concurrent systems. IEEE Transactions on Software Engineering **28** (2002)
2. Crnkovic, I.: Component-based approach for embedded systems. In: Ninth International Workshop on Component-Oriented Programming, Oslo. (2004)
3. Bondarev, E., Muskens, J., de With, P., Chaudron, M., Lukkien, J.: Predicting real-time properties of component assemblies: a scenario-simulation approach. In: Proceedings of the 30th Euromicro conference, Rennes, France, IEEE (2004)
4. Bondarev, E., Muskens, J., de With, P., Chaudron, M.: Towards predicting real-time properties of a component assembly. In: Proceedings of the 30th Euromicro conference, Rennes, France, IEEE (2004)
5. Hou., C., Shin, K.G.: Allocation of periodic task modules with precedence and deadline constraints in distributed real-time system. IEEE Transactions on Computers **46** (1995)
6. Tindell, K., Burns, A., Wellings, A.: Allocating hard real-time tasks (an np-hard problem made easy). Real-Time Systems **4** (1992)
7. Douglas, B.P.: Doing Hard Time. 0201498375. Addison Wesely (1999)
8. Gomaa, H.: Designing Concurrent Distributed, and Real-Time Applications with UML. 0-201-65793-7. Addison Wesely (2000)

9. Kodase, S., Wang, S., Shin, K.G.: Transforming structural model to runtime model of embedded software with real-time constraints. In: In proceeding of Design, Automation and Test in Europe Conference and Exhibition, IEEE (1995) 170–175
10. Shin, K.G., Wang, S.: An architecture for embedded software integration using reusable components. In: proceeding of the international conference on Compilers, architectures, and synthesis for embedded systems, San Jose, California, United States, IEEE (2000) 110–118
11. Gai, P., Lippari, G., Natale, M.D.: Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In: Proceedings of the Real-Time Systems Symposium, London (UK) Dec, IEEE (2001)
12. Shin, K.G., Lee, I., Sang, M.: Embedded system design framework for minimizing code size and guaranteeing real-time requirements. In: Proceedings of the 23rd IEEE Real-Time Systems Symposium, RTSS 2002, Austin, TX, December 2-5, IEEE (2002)
13. Fredriksson, J., Sandström, K., Åkerholm, M.: Optimizing resource usage in component-based real-time systems - appendix. Technical report, Technical Report, Mälardalen Real-Time Research Centre, Västerås, Sweden (2005) http://www.mrtc.mdh.se/publications/0836.pdf.
14. Sandström, K., Fredriksson, J., Åkerholm, M.: Introducing a component technology for safety critical embedded real-time systems. In: Proceeding of CBSE7 International Symposium on Component-based Software Engi-neering, IEEE (2004)
15. Hansson, H., M.Åkerholm, Crnkovic, I., Törngren, M.: Saveccm - a component model for safety-critical real-time systems. In: Euromicro Conference, Special Session Component Models for Dependable Systems Rennes, France, EEE (2004)
16. Arcticus: (Arcticus homepage: http://www.arcticus.se)
17. van Ommering, R., van der Linden, F., Kramer, J.: The koala component model for consumer electronics software. In: IEEE Computer, IEEE (2000) 78–85
18. Stewart, D.B., Volpe, R.A., Khosla, P.K.: Design of dynamically reconfigurable real-time software using port-based objects. In: IEEE Transactions on Software Engineering, IEEE (1997) 759–776
19. IEC: International standard IEC 1131: Programmable controllers (1992)
20. Mathworks: (Mathworks homepage : http://www.mathworks.com)
21. Bate, A., Burns, I.: An approach to task attribute assignment for uniprocessor systems. In: Proceedings of the 11th Euromicro Workshop on Real Time Systems, York, England, IEEE (1999)
22. Oh, Y., Son, S.H.: On constrained bin-packing problem. Technical report, Technical Report, CS-95-14, Univeristy of Virginia (1995)
23. Jansen, K., R, O.S.: Approximation algorithms for time constrained scheduling. In: proceeding of Workshop on Parallel Algorithms and Irregularly Structured Problems, IEEE (1995) 143–157
24. Monnier, Y., Beauvis, J.P., Deplanche, J.M.: A genetic algorithm for scheduling tasks in a real-time distributed system. In: Proceeding of 24th Euromicro Conference, IEEE (1998) 708–714
25. Montana, D., Brinn, M., Moore, S., Bidwell, G.: Genetic algorithms for complex, real-time scheduling. In: Proceeding of IEEE International Conference on Systems, Man, and Cybernetics, IEEE (1998) 2213–2218
26. Cheng., S.T., K., A.A.: Allocation and scheduling of real-time periodic tasks with relative timing constraints. In: Second International Workshop on Real-Time Computing Systems and Applications (RTCSA), IEEE (1995)
27. Fonseca., C.M., Flemming, P.J.: An overview of evolutionary algorithms in multiobjective optimization. Evolutionary computation **3** (1995)