Task Allocation Optimization for Multicore Embedded Systems

Juraj Feljan, Jan Carlson
Mälardalen Real-Time Research Centre
Mälardalen University
Västerås, Sweden
Email: juraj.feljan@mdh.se, jan.carlson@mdh.se

***Abstract*—In many domains of embedded systems, the increasing performance demands are tackled by increasing performance capacity through the use of multicore technology. However, adding more processing units also introduces the issue of task allocation — decisions have to be made which software task to run on which core in order to best utilize the hardware platform. In this paper, we present an optimization mechanism for allocating tasks to cores of a soft real-time embedded system, that aims to minimize end-to-end response times of task chains, while keeping the number of deadline misses below the desired limit. The optimization relies on a novel heuristic that proposes new allocation candidates based on information how tasks delay each other. The heuristic was evaluated in a series of experiments, which showed that it both finds better allocations, and does it in fewer iterations than two heuristics that we used for comparison.**

***Keywords*—architecture optimization; heuristic; task allocation; embedded systems; multicore; soft real-time; model-based development;**

I. Introduction

Most computer systems in use today are embedded systems. Their presence is ubiquitous, as they are used in industry, entertainment, transport, medicine, communication, commerce, etc. An aspect that they share with general-purpose computer systems is a constantly increasing performance intensity. They include more complex functionality than ever, while having to be reliable, flexible, maintainable and robust. At the same time, functionality that had traditionally been realized in hardware is instead being implemented in software (e.g., software defined radio [1]). There is a trend to cope with the increasing performance demands by increasing the number of processing units, for instance by using multicore technology. A multicore processor is a single chip with two or more processing units called cores, that are coupled tightly together in order to keep power consumption reasonable. While enabling a larger performance capacity, increasing the number of processing units also opens up an issue of how to best allocate software modules (in the embedded system domain typically referred to as tasks) to the available cores, in order to best utilize the hardware platform. Depending on the performance aspect of interest, the allocation can have a substantial impact on the performance. An intuitive example is allocating too many tasks to a core — the core will be overloaded and the tasks will miss their deadlines.

A possible approach for finding out whether a particular allocation yields satisfactory performance could be to implement, deploy and run the system in order to collect performance measurements. However, in order to avoid redeployment, which can be time consuming and costly, a preferred approach would be to predict the performance early in the development process, in line with what software performance engineering advocates [2]. The idea is to use models of the system under development to obtain performance predictions with sufficient accuracy, already prior to the implementation, and thus get an indication whether a particular allocation is good or bad in terms of performance. Also by using models, we can test the performance of a large number of candidate allocations in much shorter time than we could by performing measurements on a running system.

In our previous work [3], we presented a prototype model-based framework for allocating tasks to a soft real-time multicore embedded system. In this paper, we describe an enhanced version of the framework that now provides support for automatically optimizing task allocation with respect to end-to-end response times and deadline misses. At the center of the optimization process is a novel heuristic that guides the iterative search for a good allocation. The heuristic uses a so called delay matrix, which contains information how tasks delay each other, to propose a new candidate allocation for assessment in the next iteration of the optimization. We illustrate the efficiency of the heuristic by running a series of experiments in which we compare it to two reference heuristics.

The paper is organized as follows. Section II sets the context of the work by giving a short overview of the domain and the aforementioned framework. Sections III and IV are the core of the paper, they describe the optimization mechanism and the new heuristic, respectively. Section V presents the experiments conducted in order to evaluate the heuristic. Section VI presents related work, before Section VII concludes the paper and discusses future work.

II. Background and context

In this section we specify the domain of the work and briefly outline our model-based framework for allocating software tasks in a multicore embedded system. The framework was presented in [3], and it is the context within which we perform task allocation optimization. In other words, response time optimization and the novel heuristic we present in this paper constitute a part of the framework.

As described in the previous section, it is desirable to

be able to predict the performance of a system already at design-time. That way design faults which lead to performance issues can be caught early, prior to the implementation, when it is cheaper and simpler to correct them [4]. In general, performance is an umbrella term capturing many aspects that determine whether an allocation of tasks to cores is considered good or bad, including for instance throughput, the number of deadline misses, memory consumption, energy consumption, reliability, maintainability, security, etc. This work primarily targets soft real-time systems, i.e., systems where timing is crucial for the correct execution, but where occasional deadline misses are tolerated (as opposed to hard real-time systems where an absence of deadline violations must be guaranteed for the worst-case scenario). The performance metrics we focus on are end-to-end response times, the number of deadline misses and core load. Since these metrics depend heavily on the dynamic interplay between tasks, and since we focus on average performance (rather than a worst-case scenario), they cannot be derived analytically from task parameters. Instead, we obtain the metrics by simulating a model of the system.

Figure 1 illustrates our task allocation framework. As part of the complex manual functional design activity, the system designer defines software and hardware models of the system under development. The software of the system is specified as a collection of tasks and the connections between them. Tasks can be periodic or event-driven, and each task has a number of parameters: priority, best- and worst-case execution time and period (for periodic tasks). The hardware specification describes the execution platform, including the number of available cores, the type of scheduler each core runs, and the delays of accessing local and global memory, respectively. In addition to the software and hardware models, the system designer can (but does not necessarily need to, therefore the dashed line in Figure 1) specify a number of initial affinity specifications to be tested by the framework. An affinity specification (or allocation) is a mapping between the software and hardware models, defining to which core each task is allocated.

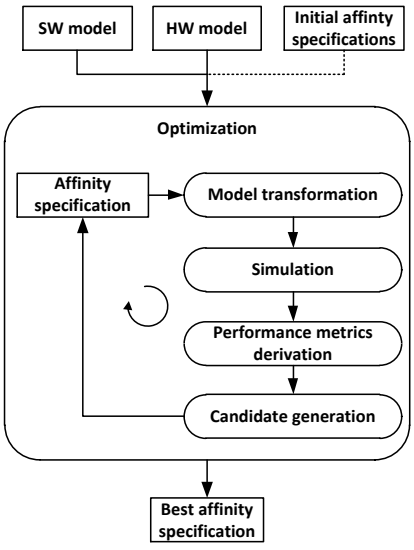


Fig. 1: Task allocation framework

The software model, hardware model and possibly initial affinity specifications are input into an optimization cycle, where each iteration generates a new allocation candidate, transforms the input models into a simulation model, executes it, derives relevant performance metrics, and determines whether the new allocation was an improvement of the best allocation found thus far. When the cycle stops, the framework outputs the best allocation it was able to find, which is then used as a specification for subsequent implementation activities.

III. End-to-end response time optimization

In this section we describe how this general task allocation framework is instantiated for optimizing end-to-end response times of task chains.

Allocating tasks to cores is a bin packing like problem. Bin packing is an NP-hard problem [5], i.e., no algorithm is known that can find the optimal solution in polynomial time. Furthermore, an inherent property of design-time model-based analysis — when detailed property values valid for the running system are typically unknown — is that the analysis methods use estimates and approximations. Having this in mind, rather than finding the optimal solution, our goal for optimizing task allocation to multicore embedded systems is to find a good allocation quickly (in as few iterations as possible).

As mentioned in the previous section, in the current work we focus on the following three performance metrics: end-to-end response times, the number of deadline misses and core load. The goal of the optimization is to reduce the average end-to-end response times of particular task chains, while keeping the total number of deadline misses in the system below a certain limit. A task chain represents a chain of execution flow, and is defined by a periodic task and the event-driven tasks it triggers in sequence. Information about which chains to optimize and what is the allowed limit of deadline misses is specified by the system designer as part of the software model.

The pseudocode of the optimization algorithm is shown in Listing 1. The algorithm starts by generating a simulation model, by means of model transformation, from a software model, a hardware model and an affinity specification. A simulation model is an executable model that captures the dynamic interaction between the tasks on the same core and across cores, respectively, as defined by the affinity specification. In each step of the simulation, on every core the corresponding scheduler checks which of the tasks allocated to the core are ready for execution, and based on their priorities grants execution to one of them. This pattern is then repeated for a number of simulation steps. The simulation duration can be defined by the system designer or computed automatically by the framework. By executing the simulation model, we obtain simulation data. As we envision the framework to support multiple analyses, the raw simulation results can hold data necessary for deriving various metrics. More details about the model transformation and simulation can be found in [3].

In the next step of the optimization algorithm, from the raw simulation data, we parse the metrics relevant for the

Listing 1: The optimization algorithm

```
optimizeAllocation(sw_model, hw_model,
     initial_affinity_specs) {
  for (i = 1 to NUMBER_OF_RESTARTS) {
    affinity_spec = initial_affinty_specs[i];
    best_allocation = null;
    for (j = 1 to NUMBER_OF_OPTIMIZATION_STEPS) {
      simulation_model = transform(sw_model, hw_model,
           affinity_spec);
      raw_simulation_results = simulate(simulation_model,
           NUMBER_OF_SIM_STEPS);
      allocation = derivePerformanceMetrics(
           raw_simulation_results, affinity_spec);
      if (betterOrEqual(allocation, best_allocation)) {
        best_allocation = allocation;
      }
      affinity_spec = generateNewAffinitySpecification(
           best_allocation, sw_model);
    }
    if (betterOrEqual(best_allocation,
         overall_best_allocation)) {
      overall_best_allocation = best_allocation;
    }
  }
  return getAffinitySpecification(overall_best_allocation);
}
```

Listing 2: The delay matrix heuristic

```
generateNewAffintySpecification(allocation, sw_model) {
  if (isFeasible(allocation)) {
    chain_of_interest = selectChainToOptimize(sw_model);
  } else {
    chain_of_interest = selectChainBasedOnDeadlineMisses(
         allocation);
  }
  delay_matrix = getDelayMatrix(allocation);
  task_to_move = selectProblematicTaskForChain(delay_matrix,
       chain_of_interest);
  old_affinity = getAffinity(task_to_move, allocation);
  new_affinty = selectCoreWithLowLoad(allocation,
       old_affinity);
  affinity_spec = getAffintySpecification(allocation);
  affinity_spec = moveTask(affinity_spec, task_to_move,
       new_affinty);
  if (random() <= TASK_SWITCH_PROBABILITY) {
    task_to_switch = selectRandomTaskFromCore(newAffinty);
    affinity_spec = moveTask(affinity_spec, task_to_switch,
         old_affinity);
  }
  return affinity_spec;
}
```

optimization: end-to-end response times and the number of deadline misses for task chains, and information about task delays. Core load is a metric not obtained by simulation, but rather calculated statically for each core, based on the tasks allocated to the core, their rate of triggering and their average execution times. The metrics and their corresponding affinity specification are contained in the `allocation` object shown in Listing 1. Having obtained the metrics, we are now able to compare the current allocation to the best one found so far. If the current allocation is at least as good as the best one found so far, it becomes the new best allocation. Allocations are compared in the following way. A feasible allocation (i.e., an allocation where the total number of deadline misses is below the given limit) is better than an infeasible one. Of two infeasible allocations, the one with less deadline misses is better. Of two feasible allocations, the better allocation is the one with the lower end-to-end response times of the chains being optimized.

In the following step, based on the best allocation so far and guided by our heuristic, we generate a new affinity specification, that is used as input in the next iteration. This forms the inner loop of the algorithm, which repeats model transformation, simulation, metrics derivation and generation of a new candidate. Since a new affinity specification is obtained by making a small modification to the best one found thus far, the inner loop performs local search around one starting allocation. In order to avoid getting stuck in a local optimum, this optimization cycle is repeated for several starting allocations (the outer loop in Listing 1). A set of starting allocations can be provided by the system designer and expanded by a number of random starting allocations generated by the framework. When the outer loops terminates, it outputs the best affinity specification it was able to find.

IV. The delay matrix heuristic

Next we focus on the heuristic used to generate a new affinity specification (Listing 2). It is based on the following principle: the most problematic task on a core is: (i) the task that considerably delays the tasks on the chain we are optimizing, and/or (ii) a task from the chain we are optimizing that is considerably delayed by the other tasks. When proposing a new affinity specification, the heuristic takes the best affinity specification found thus far, and moves such a problematic task to a core that has low load.

In each step of the optimization, the heuristic starts by identifying a *chain of interest* for that particular step. If the best allocation found so far is feasible, the chain of interest is selected among the ones the system designer chose to be optimized. On the other hand, if the best allocation found so far is infeasible, the heuristic will prioritize minimizing the number of deadline misses over minimizing the response time. Therefore, the chain of interest for an infeasible allocation is randomly selected among the ones that have deadline misses, with the probability of choosing a particular chain being proportional to the number of deadlines missed by the chain.

Having identified the chain of interest, the heuristic chooses a task to be relocated to a different core. This is done based on the *delay matrix*, a structure that holds information how the tasks delayed each other during the simulation. We define task delaying in the following way: in each step of the simulation, the task that gets executed delays all the other tasks that were ready for execution on the same core. The more a task delayed the tasks in the chain of interest, and the more a task in the chain of interest was delayed, the higher the probability it will get picked for relocation to a different core.

After identifying the task to be relocated, the heuristic decides where to place the task. This is done using core load — the less a core is loaded, the bigger the chance the task will be moved there.

In addition to relocating a task from one core to another, the heuristic occasionally performs a task switch. This addresses particular situations where all cores are close to fully loaded. In such cases, simply moving a task would result with an infeasible allocation. The probability of task switching is defined by the system designer as a parameter of the optimization. Having

moved a problematic task to a different core as described above, a random task is picked from the new core and moved to the original core of the problematic task.

Next we look into more detail how the delay matrix is used to choose a task to be relocated. The algorithm is given in Listing 3 and will be explained using a simple example system with four tasks: tasks T_1, T_2 and T_4 are periodic, while task T_3 is triggered by task T_2. The T_2–T_3 chain is the current chain of interest. Let us assume that the simulation and analysis resulted with the delay matrix shown in Table Ia. The matrix is read in the following way: task T_1 delays T_2 for a total of 50 units, T_1 delays T_3 for 20 units and so on. One unit of delay means a task delayed another task during one step of the simulation. The chain of interest is marked with gray in the table.

According to the algorithm shown in Listing 3, for each task a delay parameter is calculated — it corresponds to how much the task delays all the tasks in the chain of interest. Additionally, if the task itself belongs to the chain of interest, its delay parameter is increased by the amount it is delayed by all the other tasks. For task T_1 the delay parameter is 70 — 50 for delaying task T_2 plus 20 for delaying task T_3. For task T_2 the delay equals to 50 — it does not delay the other task in the chain (T_3), but it is delayed by task T_1 by 50 units. Similarly, the delay for task T_3 is 20 and for task T_4 it is 0. The delay parameters for all the tasks are normalized, i.e., divided by the sum of all delays (in this case 140). The normalized delays represent the probabilities of selecting a particular task as the problematic one — the more it cumulatively delays and is delayed, the bigger the chance it will be picked for relocation. The resulting probabilities in this particular example are shown in Table Ib.

Listing 3: Identifying a problematic task using the delay matrix

```
selectProblematicTaskForChain(delay_matrix,
     chain_of_interest) {
  delay_sum = 0;
  foreach task in tasks {
    delay = 0;
    foreach task2 in chain_of_interest {
      delay += delay_matrix[task][task2];
    }
    if (belongsTo(task, chain_of_interest)) {
      foreach task2 in tasks {
        delay += delay_matrix[task2][task];
      }
    }
    delay_info = addPair(delay_info, task, delay);
    delay_sum += delay;
  }
  foreach task in tasks {
    probability = getDelay(delay_info, task) / delay_sum;
    prob_info = addPair(prob_info, task, probability);
  }
  selected_task = selectRandomWithProbabilities(prob_info);
  return selected_task;
}
```

TABLE I

(a) Delay matrix

	T_1	T_2	T_3	T_4
T_1	—	50	20	40
T_2	0	—	0	25
T_3	0	0	—	0
T_4	0	0	0	—

(b) Resulting probabilities

	Delay	Prob.
T_1	70	50%
T_2	50	35.71%
T_3	20	14.29%
T_4	0	0%

V. Experiment

A series of experiments were conducted to evaluate the efficiency of the delay matrix heuristic. Here we describe the experiment setup and experiment results, in their respective subsections. We used two reference heuristics to compare the delay matrix heuristic against: random and load. The former proposes a new allocation candidate by taking a random task from a random core of the best allocation found so far, and moving it to a random other core. The latter heuristic tries to balance the load evenly among the cores. In each step of the optimization, it moves a random task from a random core with high load to a random core with low load. The more a core is loaded, the higher the chance that a task belonging to it will be moved, and equivalently, the less a core is loaded, the higher the chance that the task chosen for relocation will be moved there.

A. Experiment setup

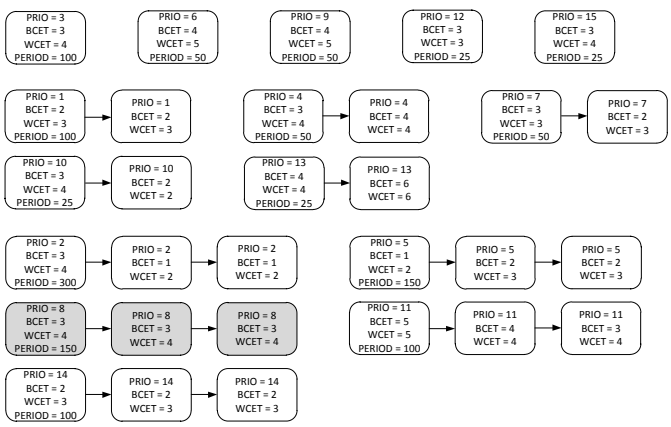
We ran allocation optimization using the delay matrix, random and load heuristics, respectively, on four representative systems covering scenarios of both low and high load in the system, and scenarios of both short and long task chains. We used two software architectures (shown in Figure 2), one with short chains and one with long chains, and varied the load in both cases, giving in total 4 tested systems (see Table II). The load was changed by scaling the best-case and worst-case execution times of the tasks by a factor of 2.

Each system consists of 30 tasks and 4 cores, which makes a total of 4^{30} possible allocation candidates. All the cores use a preemptive priority scheduler. Since we identified in previous work that there is no significant difference in communication duration when communicating tasks run on the same core versus when they run on separate cores [6], the simulations used the same cost for accessing global and local memory.

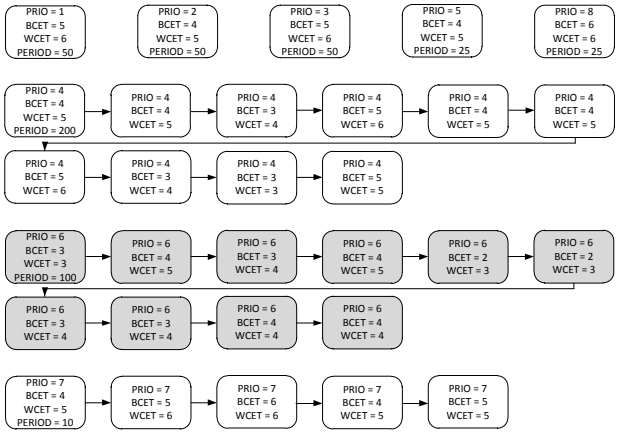
In the system with short chains, the tasks are organized into 5 chains consisting of 3 tasks, 5 chains of 2 tasks and 5 chains with a single task (Figure 2a). The system with long chains has 2 chains of 10 tasks, 1 chain of 5 tasks and 5 single-task chains (Figure 2b). All tasks within one chain have the same priority. One chain in the middle of the priority span was chosen for optimization in each system (marked with grey in Figure 2). The feasibility limit was set to 0, meaning that an allocation was considered feasible only in the case when no chain deadlines were missed.

TABLE II: Experiment systems and their load

	Best-case load	Worst-case load
Low load, short chains	0.40	0.47
Low load, long chains	0.36	0.42
High load, short chains	0.80	0.94
High load, long chains	0.72	0.84



(a) Short chains



(b) Long chains

Fig. 2: Experiment systems

All the optimization runs started from the same starting allocation, where the tasks were distributed evenly among the four available cores. 4 systems and 3 different heuristics gave 12 optimization runs. Each optimization run executed for 150 steps. In other words, each optimization run tested 150 allocation candidates beginning with the same starting allocation. The 12 optimization runs were repeated 100 times, in order to be able to draw conclusions about the general performance of the heuristics. The probability of task switching was set to 30%, for all three heuristics. Both the probability of task switching and the number of steps for each optimization run were chosen arbitrarily. Additional experiments would be necessary to be able to reason about possible rules of thumb for these parameters.

B. Experiment results

The results of the experiments are summarized in Tables III and IV and in Figures 3, 4, 5 and 6.

The tables show the number of feasible final allocations, the average feasibility point, and the average final response time for the optimized chain, for each of the 4 systems and 3 heuristics. The number of feasible final allocations tells how many of the 100 repetitions of the optimization runs ended with a feasible allocation as the best one. A feasibility point is the step in an optimization run when the first feasible allocation was found. The table shows the average feasibility point for the 100 optimization runs for each system and heuristic. Since each optimization run had 150 steps, the average feasibility point is in the interval between 0 and 150. The average final response time is based only on the optimization runs that ended with a feasible allocation.

If we focus only on the systems with low load, we see that all three heuristics have an average feasibility point of 0 (Table III). This is because of the fact that already the starting allocation for these runs happened to be feasible, due to the low load in the system and the equal distribution of the tasks to all cores. Starting an optimization run from a feasible allocation means that the run will always identify a feasible allocation as the best one. Therefore, all three heuristics have the maximum possible number of feasible final allocations. All three heuristics have managed to minimize the response time of the chosen chain to a similar average value, with the values found by the delay matrix heuristic being slightly lower.

Figures 3 and 4 present in more detail the impact of the heuristics on the systems with low load. Each point in the diagrams shows the average value of the 100 optimization runs

TABLE III: Experiment results, low load

(a) Short chains

	Random	Load	Delay matrix
Number of feasible final allocations	100	100	100
Average feasibility point	0	0	0
Average final RT	10.44	10.52	10.24

(b) Long chains

	Random	Load	Delay matrix
Number of feasible final allocations	100	100	100
Average feasibility point	0	0	0
Average final RT	35.41	35.62	35.18

TABLE IV: Experiment results, high load

(a) Short chains

	Random	Load	Delay matrix
Number of feasible final allocations	25	25	54
Average feasibility point	109.72	101.76	84.69
Average final RT	75.00	75.24	69.07

(b) Long chains

	Random	Load	Delay matrix
Number of feasible final allocations	38	44	38
Average feasibility point	101.11	101.14	85.42
Average final RT	88.81	89.55	87.79

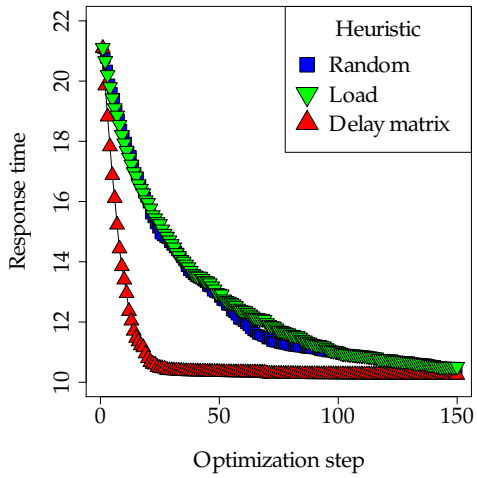


Fig. 3: Low load, short chains

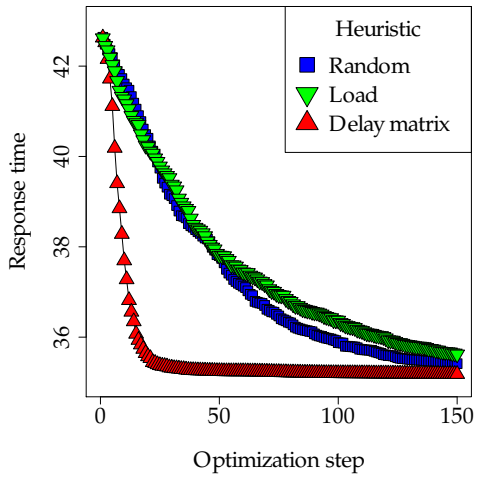


Fig. 4: Low load, long chains

at the corresponding optimization step. Even though all three heuristics end up with a roughly similar response time value after 150 optimization steps, it is clear that the delay matrix heuristic converges much faster than the other two, and gets close to the final value already after roughly 30 optimization steps.

Shifting focus to the systems with high load, from Table IV we can see that the delay matrix heuristic both finds a slightly lower average final response time, and that it finds a feasible allocation faster than the two other heuristics. Also, in the case of the system with short chains, it identified more feasible allocations as the best ones.

The optimization runs for the systems with high load are illustrated with two separate diagrams in Figures 5 and 6 — one diagram up to the feasibility point and one from the feasibility point onwards. This was necessary due to the fact that up to the feasibility point, the optimization process tries to minimize the number of chain deadline misses, while only from the feasibility point onwards does it try to minimize the chain response times, as explained in Section IV. The values shown in the diagram before the feasibility point give the number of chain deadline misses, while the values in the diagram after the feasibility point give the response time of the chosen chain. As feasibility points are different for different optimization runs, rather than showing the absolute optimization steps, the x-axes of the diagrams after the feasibility point show the optimization steps relative to the feasibility point denoted by N. Also, since not all optimization runs ended up with a feasible allocation, the diagram values after the feasibility point represent an average of less than 100 values (while the diagram values before the feasibility point represent an average of exactly 100 values).

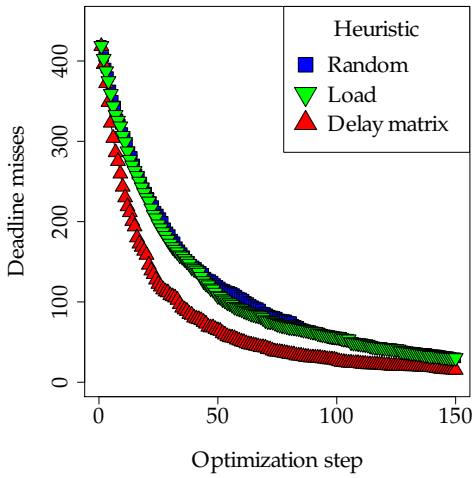
Looking at Figures 5a and 6a, the trend is again clear, and confirms what the average feasibility points show — that the delay matrix heuristic minimizes deadline misses faster than the two other heuristics. Similarly, from Figures 5b and 6b, we can see that our heuristic minimizes response times faster, and ends up with an overall lower response time.

VI. Related work

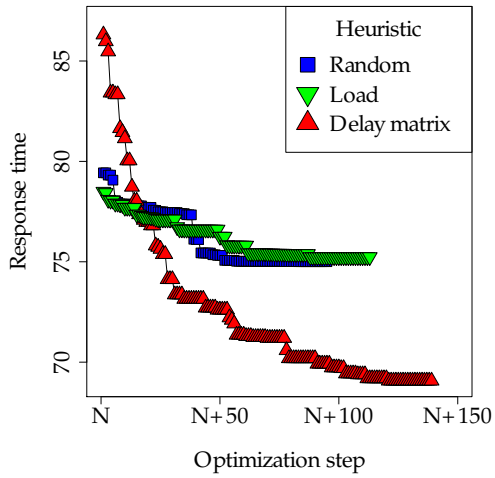
As we tackle both architecture optimization and task allocation, we present related work from these two perspectives.

Architecture optimization in general is a very broad area of research, covering different domains (e.g., embedded systems, enterprise systems), different phases of system development (design-time versus run-time), different quality attributes (e.g., cost, energy, safety, reliability, timing), using different architecture representations (e.g., standard modeling languages, custom modeling languages, graphs, matrices) and allowing different levels of freedom (e.g., allocation, software component selection, hardware component selection, scheduling). Architecture optimization of embedded systems done at early phases of development typically aims to find near-optimal architectures, since finding the optimal ones is usually not feasible within reasonable time, due to the large search space. Design-time architecture optimization is typically supported by model-based analysis, which is the source of the metrics relevant for optimization, and paired with a search technique, which can either be general-purpose (e.g., genetic algorithm) or problem-specific (as is our delay matrix heuristic). The latter try to leverage domain specific knowledge about the optimization problem to guide the search process to a better solution and/or in less iterations than the former, but at the expense of being applicable to a smaller set of problems. The literature survey by Aleti et al. [7] provides an in depth overview of the field or architecture optimization (not limited to embedded systems or design-time). However, none of the approaches are specifically tailored for multicore soft real-time systems.

ArcheOpterix [8] is a framework for architecture optimization of embedded systems modeled in AADL (Architecture Analysis and Description Language) [9]. The quality attributes it supports are reliability, performance and energy. Through its extension called Robust ArcheOpterix [10], it can account for uncertainty of design-time parameter estimates, and propose architectures that reduce the impact of the uncertainties. It supports several general-purpose search techniques including genetic algorithms, Bayesian learning and hill climbing.

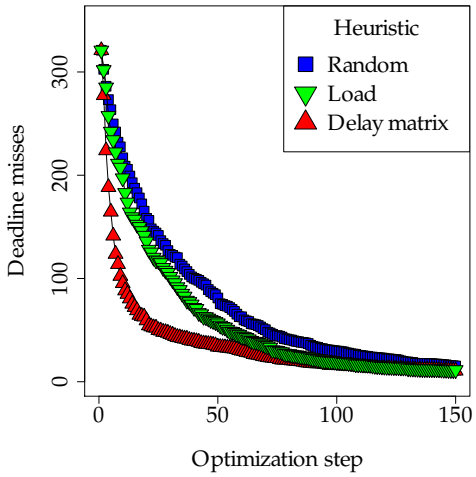


(a) Before the feasibility point

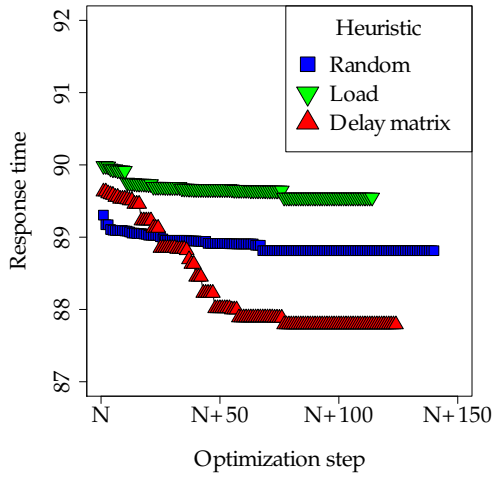


(b) After the feasibility point

Fig. 5: High load, short chains



(a) Before the feasibility point



(b) After the feasibility point

Fig. 6: High load, long chains

Task allocation is typically addressed as a sub-problem of scheduling real-time multicore systems. The approaches can be grouped into partitioning (tasks are statically allocated to cores and each core has its own scheduler), global scheduling (tasks can move between cores, under the control of a global scheduler) and hybrid scheduling (a combination of the two). Our approach belongs to the first category. Being an NP-hard problem (as mentioned in Section III), task allocation is typically aided by search techniques. These are in turn paired with schedulability analysis.

Early approaches for allocating tasks to a multiprocessor were defined by Dhall and Liu [11] — rate monotonic next fit scheduling and rate monotonic first fit scheduling. These try to allocate tasks to cores using the next fit and first fit heuristics, respectively, while keeping each core schedulable according to rate monotonic scheduling. In other approaches, additional bin-packing like heuristics (such as best fit, best fit decreasing, first fit decreasing) have been combined with different scheduling algorithms — these can be found in the survey by Davis and Burns [12]. Also, problem-specific heuristics have been developed for the purpose of task allocation. For example, Nemati et al. [13] define a custom heuristic for allocating tasks to a multicore platform in such a way that the total amount of blocking time is reduced. Unlike our approach, which focuses on soft real-time systems and combines a heuristic with model-based simulation and analysis, these approaches focus on hard real-time systems and combine heuristics with schedulability analysis.

VII. Conclusion and future work

We have presented our method for automatic optimization of task allocation to soft real-time multicore embedded systems. In an iterative search process, by simulating different allocation candidates, the method tries to find an allocation where end-to-end response times of selected task chains are minimized, while the number of deadline misses in the system is kept below the desired limit. The search process relies on a novel heuristic for proposing new allocation candidates. The heuristic makes decisions on which task to relocate to a different core based on a delay matrix — a structure which holds information on how tasks delayed each other during simulation.

In a preliminary experiment study, we have shown that the heuristic exhibits promising results, and it fulfills the goal of quickly finding a good allocation. The conducted experiments demonstrated that the heuristic converges towards a good allocation faster than two reference heuristics we used for comparison. Also, our heuristic identified final allocations which were on average slightly better than the final allocations found by the two reference heuristics.

The planned future work has two separate tracks. One is related to the optimization framework, and aims at extending it with support for optimization based on additional types of performance metrics. This involves changes to the simulation model and defining new heuristics. The second track of future work relates to the delay matrix heuristic and applying it to additional types of systems, outside of our optimization framework. We consider the heuristic also suitable for hard real-time multicore embedded systems, and distributed embedded systems. In the former case, instead of obtaining performance metrics by simulation, it would be possible to obtain them by analytically solving the system models. In the latter case, task communication over the network is more expensive than communication within the same node, therefore increasing response times of chains that have tasks allocated to several nodes, and consequently increasing delays between the tasks. As our heuristic identifies delaying tasks as problematic, it should group tasks that communicate a lot to the same node, and thus reduce chain response times. For both types of systems, we would first need to to find a suitable method for obtaining performance metrics that the delay matrix heuristic requires.

Acknowledgment

This work was supported by the Swedish Foundation for Strategic Research via the Ralf 3 project.

References

[1] T. Ulversoy, "Software defined radio: Challenges and opportunities," *Communications Surveys Tutorials, IEEE*, vol. 12, no. 4, pp. 531–550, 2010.

[2] M. Woodside, G. Franks, and D. C. Petriu, "The Future of Software Performance Engineering," in *Workshop on the Future of Software Engineering*, 2007, pp. 171–187.

[3] J. Feljan, J. Carlson, and T. Seceleanu, "Towards a model-based approach for allocating tasks to multicore processors," in *38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2012, pp. 117–124.

[4] J. Bézivin, "On the unification power of models," *Software & Systems Modeling*, vol. 4, no. 2, pp. 171–188, 2005.

[5] D. S. Johnson, "Near-optimal bin packing algorithms," Ph.D. dissertation, Massachusetts Institute of Technology, Dept. of Mathematics, 1973.

[6] J. Feljan and J. Carlson, "The Impact of Intra-core and Inter-core Task Communication on Architectural Analysis of Multicore Embedded Systems," in *The Eighth International Conference on Software Engineering Advances (ICSEA)*, 2013, pp. 402–407.

[7] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software architecture optimization methods: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 658–683, 2013.

[8] A. Aleti, S. Björnander, L. Grunske, and I. Meedeniya, "ArcheOpterix: An extendable tool for architecture optimization of AADL models," in *ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, 2009, pp. 61–71.

[9] SAE standard, no. AS5506, "Architecture Analysis & Design Language (AADL)," 2012.

[10] I. Meedeniya, A. Aleti, I. Avazpour, and A. Amin, "Robust ArcheOpterix: Architecture optimization of embedded systems under uncertainty," in *2012 2nd International Workshop on Software Engineering for Embedded Systems (SEES)*, 2012.

[11] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

[12] R. I. Davis and A. Burns, "A Survey of Hard Real-time Scheduling for Multiprocessor Systems," *ACM Comput. Surv.*, vol. 43, no. 4, pp. 1–44, 2011.

[13] F. Nemati, T. Nolte, and M. Behnam, "Partitioning Real-Time Systems on Multiprocessors with Shared Resources," in *Proceedings of 14th International Conference On Principles Of Distributed Systems*, 2010.