Periodic Real-Time Scheduling for FPGA Computers

Klaus Danne[1] and Marco Platzner[2]

[1]Design of Parallel Systems Group,
Heinz Nixdorf Institute,
University of Paderborn, Germnay
danne@upb.de

[2]Computer Engineering Group,
Department of Computer Science,
University of Paderborn, Germany
platzner@upb.de

Abstract — *Todays reconfigurable hardware devices, such as FPGAs, have high densities and allow for the execution of several hardware tasks in parallel. This paper deals with scheduling periodic real-time tasks to such an architecture, a problem which has not been considered before. We formalize the real-time scheduling problem and propose two preemptive scheduling algorithms. The first is an adaption of the well-known* Earliest Deadline First (EDF) *technique to the FPGA execution model. The algorithm reveals good scheduling performance; task sets with system utilizations of up to 85% can be feasibly scheduled. However, the EDF approach is practical only for a small number of tasks, since there is no efficient schedulability test. The second algorithm uses the concept of servers that reserve area and execution time for other tasks. Tasks are successively merged into servers, which are then scheduled sequentially. While this method can only feasibly schedule task sets with a system utilization of up to some 50%, it is applicable to large tasks sets as the schedulability test runs in polynomial time. Equally important, the method requires only a small number of FPGA configurations which directly translates into reduced memory requirements.*

1 Introduction

Reconfigurable hardware devices, the most prominent one being the field-programmable gate array (FPGA), are general-purpose devices that can be programmed after fabrication. SRAM-based FPGA variants can be re-programmed arbitrarily often, opening up the way to FPGA-based computing. For a number of applications, FPGAs have been shown to outperform general-purpose processors, and even specialized processors, in performance and

[1]This work was partly developed in the course of the Graduate College 776 -Automatic Configuration in Open Systems- and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

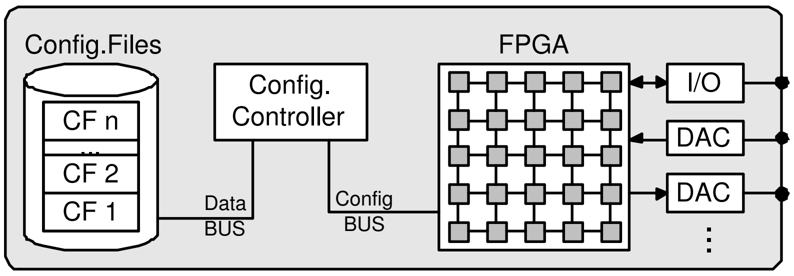

Figure 1: Target architecture of an embedded FPGA computer.

energy efficiency [1] [2]. Compared to application-specific integrated circuits (ASICs) FPGAs pay an area and performance penalty, but they offer flexibility. Todays largest FPGAs have densities beyond 10M gates and are fabricated in cutting-edge silicon technologies, which often nullifies the ASICs' speed and area advantages over FPGAs. A recent trend in FPGAs is to integrate them with processor cores and memories on a single chip. Such configurable systems on chip are promising targets for the implementation of demanding future classes of embedded systems, including ambient intelligent systems [3] and wearable computers [4].

Several recent research efforts focused on the mapping of hardware circuits (tasks) onto FPGAs. As FPGAs are reconfigurable, their silicon area can be reused for different tasks over time. As modern FPGAs have high densities, several tasks can be mapped to the device at the same time, enabling true parallel execution. Placement and scheduling of aperiodic tasks to FPGAs has already been discussed [5] [6], including realtime tasks [7].

In contrast to previous work, we deal with scheduling *periodic* realtime tasks to FPGAs, a problem that has not been addressed yet. The typical embedded reconfigurable target architecture is shown in Figure 1, and comprises an FPGA, a controller, memory, and various I/O devices. Besides the embedded software and data sections, the memory stores the configurations for the logic resource. For such an architecture, we are interested in devising scheduling algorithms for periodic real-time tasks respecting following objectives:

– *high scheduling performance:* We want to be able to generate feasible schedules for a wide range of task sets.

– *efficient schedulability test:* We want to quickly decide whether all tasks will meet their deadlines in a given schedule.

– *small number of required FPGA configurations:* The number of configurations determines the overall time spent for reconfiguration as well as the required amount of embedded memory.

The contribution of this paper lies in the formal modeling of the scheduling problem and in the presentation of two scheduling algorithms: EDF-NF and MSDL. EDF-NF is a straight-forward adaption of the EDF algorithm to our specific system model. While revealing remarkable scheduling performance, EDF-NF lacks an efficient schedulability test and requires an unbearable number of FPGA configurations. MSDL comes with a test of acceptable efficiency and keeps the number of required configurations small, at the price of a decreased scheduling performance.

2 Related Work

Hardware multitasking on FPGAs and other reconfigurable hardware devices has been studied in, e.g., [6, 5, 8, 9, 10]. Most authors assume a 2-dimensional area model that assumes partial reconfigurability and treats tasks as relocatable rectangles which can be placed anywhere on the FPGA device. The authors focus on placement and scheduling strategies in off-line and on-line application scenarios, mostly optimizing cost functions such as the total make span or the average response time. To the best of our knowledge, [7] is the only related work considering FPGA real-time scheduling. The practical realization of such systems on current technology rises several issues: First, partial reconfiguration is often limited in practice by device architectures and insufficient tool support. Some FPGA families are not partially reconfigurable at all, others, e.g., the Xilinx Virtex families, are partially reconfigurable only in full columns. Second, the issue of communication between tasks is rarely considered in the models used. Finally, related projects require tasks to be relocatable, which might be difficult to achieve for modern FPGA architectures that are not fully homogeneous.

Our work differs in that we use full FPGA reconfiguration and focus on preemptive periodic real-time scheduling. The full reconfiguration model can be used on *all* SRAM-based FPGAs and can be realized using standard design implementation tools. Task preemption requires a runtime system to be able to save the state of a task and, later on, resume it. Concepts and implementations of preemptive execution environments on FPGAs can be found in [11, 12, 10].

For periodic real-time scheduling on single processor systems, the *earliest deadline first algorithm (EDF)* has been proven optimal. EDF can schedule task sets with processor utilizations of up to 100%. Multiprocessors can execute several tasks in parallel, similar to our FPGA model. Real-time scheduling problems on multiprocessors are in general NP-hard. There exist two basic approaches, namely the *partitioning* and the *non-partitioning* method [13]. In the partitioning method, all instances of a task are executed on the same processor. Therefore, the overall task set is partitioned into m (number of processors) subsets and each subset is scheduled on another processor using single processor algorithms such as EDF. In the non-partitioning method, a task is allowed to execute on a different processor after preemption. This can be done by a multiprocessor derivate of EDF, which is suboptimal. A schedulability test for this case is given in [14].

Our scheduling problem differs from the single processor scheduling problems in that we have several hardware tasks executing in parallel. The difference to multiprocessor scheduling lies in the fact that hardware tasks share the single area resource provided by the FPGA. There is no natural way to partition the task set onto subareas of the FPGA. Rather, we present a non-partitioning EDF variant in Section 4.

3 Problem Modeling and Metrics

In this section, we introduce the task and resource model used in our work, define the scheduling problem, and discuss utilization metrics.

T_i	P_i	C_i	A_i	U_i^T	U_i^S
T_1	4	2	1/2	1/2	1/4
T_2	6	5	1/4	5/6	5/24
T_3	12	3	3/4	1/4	3/16
				1.58	0.65

Table 1: Example task set Γ^*

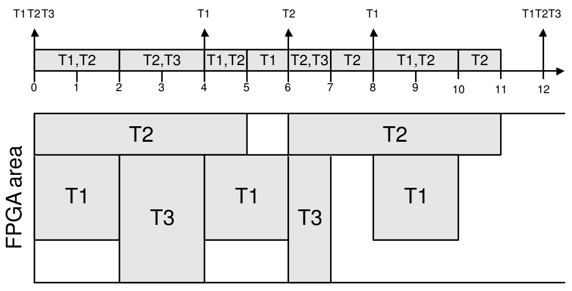

Figure 2: Preemptive schedule of three periodic FPGA tasks.

3.1 Task and Resource Models

We consider a set of periodic tasks Γ. Each task $T_i \in \Gamma$ refers to some computation which has to be performed periodically. The instances $T_{i,j}$ of task T_i are released with period P_i. That is, the release time of instance $T_{i,j+1}$ is given by $r_{i,j+1} = r_{i,j} + P_i$, where $r_{i,j}$ is the release time of instance $T_{i,j}$. C_i denotes the worst case computation time of task T_i, which is the same for all of its instances. The finishing time of task instance $T_{i,j}$ is denoted by $f_{i,j}$. In our model, we assume real-time tasks with deadlines equal to periods. Hence, the deadline of a task instance $T_{i,j}$ is given by the release time of the next instance, $r_{i,j+1}$. Finally, the amount of reconfigurable logic resources a task requires is given by A_i. We normalize all resource requirements to the available resource offered by the FPGA. Assuming that no single task requires more resources than available, we get $A_i \in [0 \ldots 1]$ (We assume that a computational task whose area exceeds the FPGA is modelled as several tasks with area less than one).

The reconfigurable hardware device offers a certain amount of computational resources, e.g., the configurable logic blocks of an FPGA, which is also referred to as the *area* of the device. We normalize this area to 1. The device can execute any set $R \subseteq \Gamma$ of tasks simultaneously, as long as the amount of resources required by the task set does not exceed the available area, i.e., $\sum_{T_i \in R} A_i \leq 1$.

A running instance of a task T_i can be preempted by another task T_j before its completion and, later on, be resumed. More general, any set of running tasks R can be preempted to execute a new set of tasks $\tilde{R}$. Technically, the runtime system has to interrupt the execution of R and to save the contexts of all tasks $T_i \in R$. Then, the FPGA is fully reconfigured with a new configuration including all tasks $T_j \in \tilde{R}$. When R is scheduled for execution again, the previously saved contexts of $T_i \in R$ are restored and R is restarted.

The time for the preemption and restore processes is neglected in our scheduling analysis. For current FPGA devices, these times are in the range of a few to a few tens of milliseconds. It must be mentioned that many research efforts address the reduction of reconfiguration times. The most promising approaches include multi-context devices and coarse-grained architectures.

As an example, Figure 2 displays a possible schedule for the task set shown in Table 1. The upper part of Figure 2 indicates the release times and deadlines for the tasks, as well as the running tasks. The lower part of Figure 2 illustrates the tasks' areas and the sharing of the FPGA area over time. Overall, four different FPGA configurations are needed for this schedule. The schedule shown can easily be proven feasible, because every task instance meets its deadline for the entire *hyper-period* of the task set (which amounts to 12 time units). The hyper-period is the least common multiplier of all task periods in the task set. A feasible schedule defined over the hyper-period can be repeated an infinite number of times without any missed deadline.

Formally, a schedule for the task set Γ assigns a set of running tasks $R_k \subseteq \Gamma$ to every point in time k, such that $\sum_{T_i \in R_k} A_i \leq 1$. No instance of a task must start execution before its release time. We call the schedule *feasible*, if each task instance finishes its execution before its deadline, i.e., $\forall i, j : f_{i,j} \leq r_{i,j+1}$.

3.2 Utilization Metrics

We define two utilization metrics to measure the computational load generated by a task set Γ. These metrics are central to the scheduling algorithm proposed in Section 5. Similar to the processor utilization factor defined in single processor real-time scheduling, we define the *time-utilization factor* of a task set Γ to be

$$U^T(\Gamma) = \sum_{T_i \in \Gamma} \frac{C_i}{P_i}. \tag{1}$$

For the special case that all tasks are executed sequentially, U^T is the fraction of time the FPGA spends executing tasks whereas $1 - U^T$ is the idle time. While such a schedule can mean an enormous waste of resources, it has two advantages. First, it allows to rely on efficient schedulability tests known from single processor scheduling. Second, the number of required FPGA configurations is bound by the number of tasks.

Improved scheduling techniques will try to better utilize the FPGA resources and execute several tasks in parallel. To describe the computational load for such a situation, we define as a more expressive metric the *system-utilization factor* of a task set Γ as

$$U^S(\Gamma) = \sum_{T_i \in \Gamma} \frac{C_i}{P_i} A_i. \tag{2}$$

U^S presents the fraction of the area-time product occupied by a task set. Visually, U^S corresponds to the gray areas in the schedule of Figure 2. The white areas in the schedule of Figure 2 correspond to the unused computational resource.

Obviously, we cannot find a feasible schedule for a task set with $U^S > 1$. Whether a feasible schedule exists for a task set with $U^S \leq 1$ depends on the specific relations among the task properties, in particular the area requirements A_i. $U^T(\Gamma)$ and $U^S(\Gamma)$ are also defined for single tasks, as they are (minimal) instances of task sets. Table 1 shows the time and system utilization factors for the example tasks as well as for the complete task set.

As we cannot expect to fully utilize the FPGA area, the resulting system utilization will generally stay below 1. In this paper, we use U^S to experimentally rate the quality of a scheduling algorithm. We do not derive bounds for U^S that can be used to decide schedulability for a given algorithm.

Alg. 1 Earliest Deadline First - Next Fit

```
Require: list $Q$ of ready tasks, sorted by increasing absolute deadlines
 1: procedure EDF-NF($Q$)
 2:     $R \leftarrow \emptyset$
 3:     $A^{running} = 0$
 4:     for $i \leftarrow 1, |Q|$ do
 5:         if $A^{running} + A_i \leq 1$ then
 6:             $R \leftarrow R \cup T_i$
 7:             $A^{running} = A^{running} + A_i$
 8:         end if
 9:     end for
10:     return $R$
11: end procedure
```

4 EDF-NF Scheduling

For single processor real-time systems, the *Earliest Deadline First (EDF)* algorithm which executes all ready tasks in the order of their absolute deadlines, has been proven optimal. EDF can schedule any task set which utilizes the processor less or equal to 100%. For an m-processor machine, the EDF strategy is defined to execute always the m tasks with the m smallest absolute deadlines among all ready tasks. Multiprocessor EDF is not optimal, since for some task sets a feasible schedule exists which is not found by EDF.

We can apply the EDF schedulability test to a schedule that executes all tasks sequentially on the FPGA. Whenever $U^T \leq 1$, the task set is guaranteed to be feasibly scheduled by EDF without any need to group several tasks into one FPGA configuration. By executing tasks in parallel we increase the range of task sets for which feasible schedules can be found. Grouping tasks together can further be beneficial to reduce the number of configurations and, in turn, reconfiguration time and memory requirements.

To this end, we adopt the simple EDF strategy for our execution model and present the scheduling algorithm *EDF - Next Fit (EDF-NF)*, shown in Alg. 1. Similar to EDF for single (and multiprocessor) systems, EDF-NF keeps a list of all tasks which have been released and have not yet finished in a ready queue Q. The ready queue is sorted by increasing absolute task deadlines. To determine the set R of running tasks, EDF-NF scans through the ready list. A task T_i is added to the set of running tasks R, as long as the sum of the area of all running tasks $A^{running}$ remains less or equal to one. Whenever the next task cannot be added, EDF-NF proceeds in the ready queue and tries to add tasks with longer absolute deadlines. At this point, EDF-NF diverges from the pure EDF rule. The motivation for adding tasks in next-fit manner is to improve the device utilization.

The procedure EDF-NF is run whenever a new instance of a periodic task is released (added to the ready list) or running instances of tasks terminate. The EDF-NF algorithm shown in Alg. 1 is performed in $O(n)$ time, where n is the number of tasks. Unfortunately, there is no efficient schedulability test. To prove schedulability, we have to simulate task executions and terminations for the complete hyperperiod. Further, the number of configurations can grow fairly large which is a major disadvantage of this algorithm.

5 Server-based Scheduling

In this section, we present a scheduling technique called *Merge Server Distribute Load (MSDL)*. To construct a schedule MSDL uses the concept of *server tasks*, or briefly servers. A server is a periodic task that reserves execution time and FPGA area for other tasks. We define a server as $S_i = (R_i, P_i, C_i, A_i)$, where $R_i = \{T_a, T_b, \dots\} \subseteq \Gamma$ is a set of tasks for which execution time and area is reserved. P_i, C_i, A_i denote the period, the computation time and the area, respectively. The area for a server is set to be the sum of all tasks represented by the server, $A_i = \sum_{T_k \in R_i} A_k$. Consequently, whenever the server S_i is running, all tasks it represents are running.

The rationale of the MSDL algorithm is to construct a set of servers Ω from the original task set Γ, such that any feasible schedule for Ω implies a feasible schedule for Γ. More specifically, MSDL constructs a set of servers Ω by properly *merging* tasks together for parallel execution. The resulting servers are then scheduled for *sequential* execution on the FPGA with single processor EDF. Feasibility for the resulting set of servers is thus efficiently checked by the utilization test $U^T(\Omega) \leq 1$.

5.1 The Merge-server Distribute Load (MSDL) Algorithm

Algorithm 2 shows the pseudo code for the MSDL technique. First, each of the initial tasks is turned into a server (line 3). Then the main loop is entered in which, iteratively, a server pair is identified and merged if possible. The selection of the two servers S_x and S_y that should be merged is done by the function *selectValidPair()* (line 8). For the

Alg. 2 Merge Server - Distribute Load

```
 1: procedure MSDL($\Gamma$)
 2:     $\Omega \leftarrow \emptyset$
 3:     for all $T_i \in \Gamma$ do                                   ▷ init
 4:         $S_i \leftarrow (\{T_i\}, P_i, C_i, A_i)$
 5:         $\Omega \leftarrow \Omega \cup S_i$
 6:     end for
 7:     loop
 8:         $S_x, S_y \leftarrow selectValidPairToMerge(\Omega)$
 9:         if no pair found then
10:             return $\Omega$                                       ▷ exit
11:         end if
12:         $S_z \leftarrow (R_x \cup R_y, P_x, C_x, A_x + A_y)$          ▷ $P_y \leq P_x$
13:         $C_x \leftarrow C_x - takeOverTime(S_x, S_z)$
14:         $\Omega \leftarrow \Omega \cup S_z$                           ▷ add server
15:         $\Omega \leftarrow \Omega \setminus S_y$
16:         if $C_x \leq 0$ then
17:             $\Omega \leftarrow \Omega \setminus S_x$
18:         end if
19:     end loop
20: end procedure
```

implementation of this function, several heuristics are conceivable. In our current version we employ a greedy strategy that select the pair of servers giving the greatest reduction in time utilization $U^T(\Omega_{old}) - U^T(\Omega_{new})$ per modest increase of system utilization $U^S(\Omega_{new}) - U^S(\Omega_{old})$.[1] Any valid pair of servers S_x and S_y must have a disjunct set of represented tasks ($R_x \cap R_y = \emptyset$) and must jointly fit onto the FPGA ($A_x + A_y \leq 1$).

If no valid server pair could be found, the algorithm exits and returns Ω as the final set of servers (line 10). Otherwise, the servers S_x and S_y are merged. Without loss of generality, we can assume that S_y is the server with the shorter period. Then, a new server S_z is created representing all tasks of the two original servers (line 12). The period and the computation time for S_z is set equal to those of S_y. Therefore, S_z is a full replacement of S_y, and S_y can be removed from Ω. The computation time of S_x is reduced, since the new server S_z reserves area and computation time for the tasks of S_x as well. The actual computation time reduction depends on how often the new server S_z executes within the period of S_x. A pessimistic approximation for the reduction is given by

$$C_x \leftarrow C_x - C_z(\lfloor P_x/P_z \rfloor - 1). \tag{3}$$

By a more involved analysis it can be shown that the reduction time of server S_x reduces to $C_x \leftarrow C_x - f(S_x, S_z)$, with

$$f(S_x, S_z) = \min\Big\{ \begin{array}{ll} C_z(\lfloor P_x/P_z \rfloor - 1) & + \max\big\{2C_z((\lfloor P_x/P_z \rfloor + 1)P_z - P_x), 0\big\}, \\ C_z \lfloor P_x/P_z \rfloor & + \max\big\{2C_z((\lfloor P_x/P_z \rfloor + 2)P_z - P_x), 0\big\} \end{array} \Big\} \tag{4}$$

Table 2 continues the example from Section 3. The table shows the set of servers Ω_k^* generated in each iteration k. Initially, the servers $\Omega_0^* = \{S_1, S_2, S_3\}$ are created. In the first iteration, S_1 and S_2 are selected and merged into S_4. S_2 receives the new computation time $C_2 \leftarrow C_2 - 2 = 3$. The server with the shorter period, S_1, is removed. In the second iteration, the residual S_2 and S_3 are merged into S_5. Not only the server with the shorter period is removed, but also S_3 since its computation time is reduced to zero. Ω_2^* is the final server set, since neither R_3, R_4 are disjunct nor $A_3 + A_4 \leq 1$. As shown in Table 2, the time utilization factor $U^T(\Omega_2^*) = 1$. Consequently, Ω_2^* can be feasibly scheduled by EDF. The resulting schedule is shown in Figure 3. The figure also indicates the original tasks of Γ^* executed *inside* the servers. Compared to the schedule given in Figure 2, MSDL requires only two FPGA configurations.

Table 2 also lists the system utilization factor U_i^S which increases over the iterations, since larger severs will reveal more idle areas and times inside their reservations. In essence, MSDL trades system utilization for time utilization to allow for an efficient schedulability test and to reduce the number of FPGA configurations.

[1] Ω_{old} denotes the task set before, whereas Ω_{old} denotes the task set after merging the selected pair of servers.

S_i	R_i	P_i	C_i	A_i	U_i^T	U_i^S
S_1	T_1	4	2	1/2	1/2	1/4
S_2	T_2	6	5	1/4	5/6	5/24
S_3	T_3	12	3	3/4	1/4	3/16
					1.58	0.65
$\cancel{S_1}$	T_1	4	0	1/2	1/2	1/4
S_2	T_2	6	3	1/4	1/2	1/8
S_3	T_3	12	3	3/4	1/4	3/16
S_4	T_1, T_2	4	2	3/4	1/2	3/8
					1.25	0.69
$\cancel{S_2}$	T_2	6	0	1/4	1/2	1/8
$\cancel{S_3}$	T_3	12	0	3/4	1/4	3/16
S_4	T_1, T_2	4	2	3/4	1/2	3/8
S_5	T_2, T_3	6	3	1	1/2	1/2
					1	0.88

Table 2: Servers generated for the example task set Γ^* by the *MSDL (Merge Server Distribute Load)* algorithm

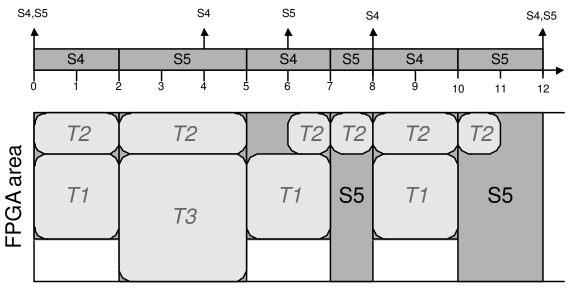

Figure 3: Schedule of server task set generated by MSDL

6 Simulation Results

We have conducted simulation experiments with synthetic workloads to evaluate the scheduling performance of the EDF-NF and MSDL algorithms. We have generated random task sets with varying values for the system utilization factor $U^S(\Gamma)$. To this end, we have proceeded as follows: We have chosen an interval $[a, b] \subset N$ for the tasks' computation time, an interval $[a, b] \subseteq [0, 1]$ for the tasks' area, and an interval $[a, b] \subseteq [0, 1]$ for the tasks' system utilization factor. Finally, we have set a bound $\hat{U}^S \leq 1$ for the system utilization of the entire task set. Then, new tasks T have been created one by one with computation times, areas, and periods equally distributed in the intervals given above. Tasks have been added to the task set, as long as $U^S \leq \hat{U}^S$.

The randomly generated task sets have been scheduled with the EDF-NF and MSDL algorithms. Figure 4 shows the percentage of feasible scheduled task sets for both techniques over the task set's system utilization factor. For the curves denoted by *n=small*, task areas A_i were equally distributed in $[0.2, 0.4]$, and computation times and periods were chosen such that the time utilization factor U_i^T is equally distributed in $[0.2, 0.4]$. These settings result in task sets of approximately 10 tasks on average. As expected, Figure 4 shows clearly the superiority of EDF-NF over MSDL in scheduling performance. EDF-NF is able to schedule about 50% of the task sets with a system utilization factor around 85% and accepts almost all task sets with U^S less than 75%. In contrast to that, MSDL is able to schedule only few task sets with a U^S exceeding 70%, and achieves an acceptance rate of 50% for task sets with a U^S around 55%.

For the curve denoted by *n=medium*, task areas A_i were equally distributed in $[0.1, 0.2]$, and computation times and periods were chosen such that the time utilization factor U_i^T is equally distributed in $[0.1, 0.2]$. These settings result in task sets of approximately 40 tasks on average. MSDL performs slightly worse than on smaller task sets. For EDF-NF,

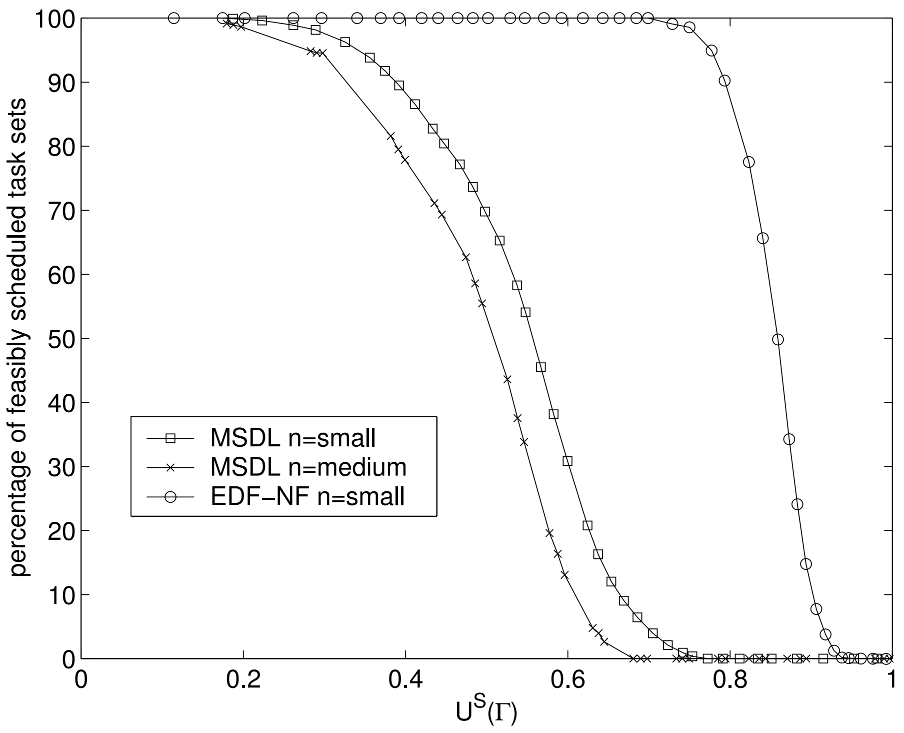

Figure 4: Percentage of feasibly scheduled task sets over task set's system utilization factor by MSDL and EDF-NF

however, we could not gain results as the EDF-NF schedulability test did not terminate in reasonable time.

7 Conclusion

We have discussed the problem of real-time scheduling onto FPGA computers and have presented two scheduling algorithms, EDF-NF and MSDL. EDF-NF is much better than MSDL in the sense that it can generate feasible schedules for task sets with higher system utilization. The experiments, however, emphasized one benefit of MSDL: an efficient schedulability test. For larger real-time task sets that need a schedulability guarantee, EDF-NF is not an option.[2] The second benefit of MSDL lies in the reduced number of FPGA programming files, as demonstrated in Section 5.

In future work we will emphasize this advantage by a quantitative evaluation of the number of required programming files by EDF-NF compared to MSDL based on further simulations. Moreover, we plan to develop and evaluate different heuristics for selecting the servers to be merged. One goal is the further reduction of the number of programming files. In addition we will incorporate the modelling of the time overhead resulting from FPGA reconfiguration and state saving during task preemption in order to increase the accuracy of our results.

[2]The hyperperiod of a task set grows extremely fast. For example, for task sets with periods bounded by 100, the worst case hyperperiod exceeds 4×10^9 for 5 tasks, and 3×10^{18} for 10 tasks.

References

[1] Oskar Mencer, Martin Morf, and M. J. Flynn. Hardware software tri-design of encryption for mobile communication units. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 5, pages 3045–3048, 1998.

[2] A. Abnous, K. Seno, Y. Ichikawa, M. Wan, and J. Rabaey. Evaluation of a low-power reconfigurable DSP architecture. In *Proceedings of the 5th Reconfigurable Architectures Workshop (RAW)*, volume 1388, pages 55–60. Springer, 1998.

[3] Rudy Lauwereins. Creating a World of Smart Re-configurable Devices. In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 790–794, 2002.

[4] Christian Plessl, Rolf Enzler, Herbert Walder, Jan Beutel, Marco Platzner, Lothar Thiele, and Gerhard Tröster. The Case for Reconfigurable Hardware in Wearable Computing. *Personal and Ubiquitous Computing*, pages 299–308, October 2003. Springer-Verlag.

[5] J. Teich, S. Fekete, and J. Schepers. Optimization of dynamic hardware reconfigurations. *The J. of Supercomputing*, 19(1):57–75, May 2000.

[6] K. Bazargan, R. Kastner, and M. Sarrafzadeh. Fast template placement for reconfigurable computing systems. *IEEE Design and Test of Computers*, pages 68–83, March 2000.

[7] Christoph Steiger, Herbert Walder, and Marco Platzner. Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-time Tasks. *IEEE Transactions on Computers*, 53(11):1392–1407, November 2004.

[8] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt. Dynamic scheduling of tasks on partially reconfigurable FPGAs. *IEE Proceedings – Computers and Digital Techniques*, 147(3):181–188, May 2000.

[9] Herbert Walder and Marco Platzner. Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations. In *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, pages 284–287. CSREA Press, June 2003.

[10] H. Simmler, L. Levinson, and Reinhard Manner. Multitasking on FPGA coprocessors. In *FPL*, pages 121–130, 2000.

[11] Klaus Danne. Memory management to support multitasking on fpga based systems. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig04) ISBN 970-692-169-9*. Mexican Society of Computer Science, SMCC, 20 - 21 September 2004.

[12] Klaus Danne. Operating systems for fpga based computers and their memory management. In *ARCS 2004 Organic and Pervasive Computing, Workshop Proceedings*, volume P-41 of *GI-Edition Lecture Notes in Informatics (LNI)*, Bonn, 26 March 2004. Köllen Verlag.

[13] Björn Andersson and Jan Jonsson. Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. In *RTCSA*, pages 337–346, 2000.

[14] Joel Goossens, Sanjoy Baruah, and Shelby Funk. Real-time scheduling on multiprocessor. In *Proceedings of the 10th International Conference on Real-Time System*, 2002.