Abstract: A prototype system that executes a set of periodic real-time tasks utilising dynamic hardware reconfiguration is presented. The proposed scheduling technique, merge server distribute load (MSDL), is not only able to give an offline guarantee for the feasibility of the task set, but also minimises the number of device configurations. After describing this technique, the schedulability analysis is extended to cover different runtime system overheads, including the device reconfiguration time. Then, a lightweight runtime system that performs the online part of the MSDL scheduling technique is detailed. The runtime system is implemented entirely in hardware. Finally, the corresponding synthesis tool flow is outlined and the overhead posed by the runtime system is reported.

1 Introduction and related work

Reconfigurable hardware devices, such as field-programmable gate arrays (FPGA), are increasingly used in embedded systems that operate under real-time conditions. Today’s FPGAs can execute several hardware tasks in parallel because of the increasing logic capacity, as well as sequentially because of runtime reconfiguration capabilities. To use reconfigurable devices for the real-time workloads of embedded system applications, scheduling techniques and execution environments are required that create predictable task timings.

Most reconfigurable devices use an SRAM-based configuration technology and thus need re-programmed arbitrarily often, even at runtime [1]. Over the years, runtime reconfiguration capabilities have improved [2], opening up the way to multitasking on reconfigurable hardware. Multitasking requires a runtime system to manage the resources and the execution of hardware tasks. Several authors have investigated the related resource management problems, such as scheduling and placement. In [3], a resource manager schedules concurrent tasks onto a multi-FPGA platform. However, each task requires one or more FPGAs. In [4], a one-dimensional area model is used where multiple tasks run simultaneously on the same FPGA. Tasks span the entire device height but can have arbitrary widths. In [5–9], the authors assumed a two-dimensional area model that treats tasks as relocatable rectangles. Techniques for off-line placement and scheduling are described in [7]. The goal is to minimise the total execution time of a given task graph on an FPGA. The minimisation of the mean response time in online scenarios is considered in [6, 8]. Preemptive multitasking calls for a runtime system able to stop a task and save its context, and later on restore the context and resume the task. Concepts and implementations of preemptive runtime systems for FPGAs have been described in, for example, [10–13]. Some work has also addressed real-time scheduling on FPGAs. In [5], online arriving real-time tasks are considered. Scheduling periodic real-time tasks onto a CPU-FPGA system has been described by [14], and scheduling such tasks onto partially reconfigurable devices in [15]. In contrast to related work, we present a system executing periodic real-time tasks on fully reconfigurable devices. As we do not rely on partial reconfiguration, our technique is applicable to all SRAM-based devices. This paper extends our previous work [16] with a detailed discussion of the scheduling method and the prototype implementation, new simulation experiments, and implementation results. The task model is described in Section 2. In Section 3, we develop a scheduling algorithm that guarantees at design time that no task will miss its deadline at runtime. The performance of the scheduler is evaluated in Section 4. An all-hardware implementation of the runtime system is discussed in Section 5, followed by an outline of the synthesis tool flow in Section 6. Finally, in Section 7 we report on the overheads posed by the runtime system and discuss a proof-of-concept application.

2 Task and resource model

Periodic activities represent the major computational demand in many real-time systems [17]. Examples are the cyclic processing of sensory data, control loops and monitoring tasks. To investigate the mapping of such workloads to dynamically reconfigurable hardware, we adapt the widely accepted and well-studied periodic task model introduced by Liu and Layland [18].

We consider a set of periodic tasks \( \Gamma \). Each task \( T_i \in \Gamma \) refers to some computation which has to be performed periodically. The instances \( T_{ij} \) of task \( T_i \) are released with period \( P_i \), that is, the release time of instance \( T_{ij+1} \) is given by \( r_{ij+1} = r_{ij} + P_i \). \( C_i \) denotes the worst-case computation time of task \( T_i \), which is the same for all of its instances. In our model, we assume real-time tasks with deadlines equal to periods. Hence, the deadline of a task instance \( T_{ij} \) is given by the release time of the next instance, \( r_{ij+1} \). Finally, the amount of reconfigurable logic resources a task requires is given by the area parameter \( A_i \).
normalise all area requirements to the area offered by the FPGA. Assuming that no single task requires more resources than available, we get \( 0 < A_i \leq 1 \).

We define two utilisation metrics that measure the computational load generated by a task set. First, the time utilisation factor of a task set \( \Gamma \) is

\[
U^{(T)}(\Gamma) = \sum_{I_i \in \Gamma} \frac{C_i}{P_i}
\]

(1)

In a single processor system or when all tasks are executed sequentially on an FPGA, \( U^{(T)} \) is the fraction of time the device spends executing tasks whereas \( 1 - U^{(T)} \) is the idle time. A time utilisation factor exceeding one means that the task set is not schedulable. However, this metrics does not cover the fact that several tasks might execute simultaneously on an FPGA. Hence, as a more expressive metrics including the task areas we define the system utilisation factor of a task set \( \Gamma \) as

\[
U^{(S)}(\Gamma) = \sum_{I_i \in \Gamma} \frac{C_i A_i}{P_i}
\]

(2)

A system utilisation factor exceeding one means that the task set is not schedulable on the FPGA. The objectives for devising an appropriate scheduling technique are threefold: first, we have to generate feasible schedules, that is, all instances of all periodic tasks must meet their deadlines. Second, we are interested in an efficient schedulability test. Such a test allows us to quickly determine offline, whether we will meet all deadlines. Third, we seek to minimise the number of device configurations (bitstreams) in order to reduce the system’s memory requirements.

3 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. Servers are a well-known concept in real-time scheduling [17]. Servers are artificial periodic tasks offering computational load (MSDL). To construct a schedule for the task set is not schedulable. However, this metrics does concede, the goal must be to reduce the time utilisation of the resulting server set by increasing the number of parallel executing tasks.

We achieve this by a greedy strategy that selects among all server pairs the one that gives the greatest reduction in time utilisation \( U^{(T)}(\Omega_{old}) - U^{(T)}(\Omega_{new}) \) per increase of system utilisation \( U^{(S)}(\Omega_{new}) - U^{(S)}(\Omega_{old}) \) where \( \Omega_{old} \) denotes the set of servers before, whereas \( \Omega_{new} \) denotes the server set after merging the selected pair of servers.) Any valid pair of servers \( S_x \) and \( S_y \) must have a disjoint 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 can be found, the algorithm exits and returns \( \Omega \) as the final set of servers (line 9). Otherwise, the servers \( S_x \) and \( S_y \) are merged. Without loss of generality, we assume that \( S_x \) is the server with the shorter period. Then, a new server \( S_z \) is created representing all tasks of the two original servers (line 10). The period and the computation time for \( S_z \) are set to equal those of \( S_x \). Therefore \( S_z \) is a full replacement of \( S_x \), and \( S_y \) can be removed from \( \Omega \). Generally, the server \( S_z \) will not be fully replaced as neither its computation time nor its period must be shorter than that of server \( S_x \). The actual reduction of computation time depends on how often the new server \( S_z \) executes within the period of \( S_x \). Obviously, \( S_z \) is released at least \( \lceil P_x/P_z \rceil \) times between the release time and the deadline of the old server \( S_y \). However, only \( \lceil P_x/P_z \rceil - 1 \) instances of \( S_y \) might be completed in that period of time. The amount of computation time server \( S_z \) can take over from server \( S_x \) is given by

\[
\text{take Over Time}(S_x, S_z) = C_x \lfloor P_x/P_z \rfloor - 1
\]

(3)

Equation 3 is a pessimistic estimate. Using a more involved analysis, we have shown an improved estimate for the reduction of \( S_x \) in [19].

As an example, we apply the MSDL algorithm to the task set shown in Table 1. The table lists the set of servers \( \Omega_x \) generated during two iterations of the MSDL algorithm. Initially, the servers \( \Omega_x = \{ S_1, S_2, S_3 \} \) are created. In the first iteration, \( S_1 \) and \( S_2 \) are selected and merged into \( S_x \). \( S_3 \) receives the new computation time \( C_z = C_2 - C_z = 2 \). The server with the shorter period, \( S_1 \), is removed. In the second iteration, the residual \( S_3 \) and \( S_z \) are merged into \( S_z \). Not only the server with the shorter period is removed, but also \( S_3 \) since its computation time is reduced to zero. \( \Omega_z \) is the final server set, since neither \( R_x \) nor \( R_z \) are disjoint nor \( A_x + A_z \leq 1 \). As shown in Table 1, the time utilisation factor \( U^{T}(\Omega_z) = 1 \). Consequently, \( \Omega_z \)

Fig. 1 shows the pseudo-code for the MSDL technique. First, each of the initial tasks is turned into a server
can be feasibly scheduled by EDF. The resulting schedule, shown in Fig. 2, requires two servers which corresponds to two FPGA programming files.

The MSDL schedulability of a task set is decided in polynomial time by constructing the final server set and verifying the single processor EDF condition on this set. MSDL generates only feasible schedules and tries to minimise the single processor EDF condition on this set. MSDL schedules tasks of a task set by the single processor EDF. Hence, we can use a result from EDF theory, that bounds the number of times a server $S_i$ can get preempted to

$$N_i = \sum_{S \in \Omega} \left[ \frac{P_i}{T_i} \right] - 1 \quad (4)$$

Knowing the maximum number of preemptions a server can receive, we can easily calculate the maximum reconfiguration time overhead. Each server is configured once when it starts execution and for at most $N_i$ times when it resumes from preemption. We now add the maximum reconfiguration time to the server’s original computation time.

$$\tilde{C}_i = C_i + (1 + N_i) \cdot t_{rc} \quad (5)$$

Using the increased server computation times $\tilde{C}_i$, we recalculate the total time utilisation of $\Omega$.

$$\tilde{U}^T_{\Omega} = \sum_{S \in \Omega} \tilde{C}_i / T_i \quad (6)$$

The set of servers is feasibly scheduled by MSDL, if and only if $\tilde{U}^T(\Omega) \leq 1$.

Another type of overhead is added by the preemption mechanism, that is, the context save and store operations. Similar to [13], our implementation of the runtime system requires the tasks to include logic that stores/loads the context to/from an external memory (see Section 5). As these load and store functions are called by the runtime system, we can add the associated time overheads to the server execution times similar to (5).

The last type of overhead is caused by the runtime system which, in our implementation, is completely mapped to reconfigurable hardware. The resulting area overhead is accounted for by reducing the available total device area for the scheduling analysis. The time overhead, which is because of loading and storing the context of the runtime system, is again added to the server execution times.

### 4 Scheduling performance

We conducted a series of simulation experiments to evaluate the performance of the MSDL scheduling method for different benchmarks. The benchmarks are based on synthetically created tasks, as there exists no common benchmark for real-time tasks on reconfigurable systems. Using parameters of typical tasks is of limited use, as the schedulability depends also on the device capacity. For example, an eight-point 1D DCT core requires 982 VirtexII slices [20], which is less than 5% of a XC2V250 but more than 50% of a XC2V4000. Hence, we resorted to synthetic tasks but tried to cover a wide range of parameters. The base benchmark $BM_{std}$ includes tasks $T_i$ with areas $A_i$ and time utilisation $U^T(T_i)$ equally distributed in $[0.1, 0.5]$. The computation times were chosen from $[10, 20, 30, \ldots, 300]$, which also determines the periods. Overall, 5000 task sets with various $U^T(\Gamma)$ were created by randomly combining single tasks. Such a setup creates task sets between 1 and 13 tasks, each with a period between 20 and 3000 time units. We further created a benchmark $BM_{area, util}$ using smaller area values $A_i$ from the interval $[0.05, 0.25]$ but larger time utilisations $U^T(T_i)$ with values from $[0.2, 1]$. The benchmark $BM_{area, util}$ was created with interchanged intervals for $A_i$ and $U^T(T_i)$.
The left-hand side of Fig. 3 shows the performance of MSDL in terms of the success rate over the system utilisation $U$ of the workload. These simulations do not include preemption overheads. For the base benchmark, MSDL achieves an acceptance rate of 50% for task sets with $U$ around 0.6, but is able to schedule only few task sets with $U$ exceeding 0.7. The performance on the BM small-big benchmark drops dramatically compared to the base benchmark. The reason for this behaviour is that MSDL attempts to perform a rather high number of server merges, since the tasks have small area values but high time utilisations. Each merging step adds some overhead to the set of servers leading to a degraded schedulability. The opposite effect takes place on task sets with large area values but small time utilisations. Here, only few server merges are required and thus little overhead is introduced.

To study the impact of the size of a task set on the performance, we conducted another series of experiments where the number of tasks per task set was kept constant over different values for the system utilisation. For a given system utilisation, task sets with many tasks will comprise smaller and shorter tasks than task sets with fewer tasks. The right-hand side of Fig. 3 shows the resulting scheduling performance for task sets with $n = 10$, $n = 20$ and $n = 50$ tasks, respectively. Using this setup, the performance of MSDL was shown to be almost independent of the number of tasks $n$.

Finally, we evaluated the performance of MSDL concerning the reconfiguration overhead as described in Section 3.2. We applied MDSL to the base benchmark and varied the reconfiguration time $t_{rc}$ in $\{0, 0.1, 1, 10\}$, corresponding to zero, one-hundredth, one-tenth and one-time the computation time of the shortest task. Fig. 4 shows the results. A reconfiguration time of 0.1 has almost no effect on the schedulability. For a reconfiguration time of 1.0, the performance drop is noticeable but still small. Even for a reconfiguration time of 10, a considerable amount of task sets can be feasibly scheduled by MSDL.

5 System architecture

5.1 All-hardware runtime system

We implemented an all-hardware runtime system for the execution of MSDL schedules. The decision for implementing the complete runtime system in hardware was driven by the fact that a hardware scheduler can control the user tasks in a very efficient manner and, thus, minimise the runtime overhead and deliver maximum scheduling performance. In a processor-based real-time system, a periodic timer triggers a kernel function that maintains time in terms of system ticks. The smaller the timer period is, the finer the system’s time resolution is but the higher the runtime overhead is. In our all-hardware runtime system, the scheduler runs in parallel to the user tasks and, thus, no execution time is wasted for updating the internal time reference.

As the runtime system itself undergoes full device reconfiguration, we have to include it in each single configuration bitstream. On one hand, this opens up the opportunity for compile-time optimisation of parts of the runtime system. On the other hand, we also have to save and restore the context of the runtime system.

Our prototype is based on a CELLOXICA RC203 board. Fig. 5 shows the system architecture consisting of an FPGA, a complex programmable logic device (CPLD), a flash memory card and two banks of SRAM. The CPLD is connected to FPGA user I/Os, to the FPGA configuration port, as well as to the flash memory card. This way, the CPLD acts as a reconfiguration controller that reads configuration bitstreams from the flash memory card on request of the logic implemented in the FPGA.

The main runtime system components shown in Fig. 5 comprise the timer, the scheduler, the task management and an optional memory management unit (MMU).

5.1.1 Timer: The timer provides the time reference by scaling down the clock to a system tick signal with a user-defined period. All execution times and periods are expressed in terms of system ticks.
5.1.2 Scheduler: The scheduler is the central part of the runtime system as it controls which user tasks are executed at any given time. The scheduler further decides whether the FPGA has to be reconfigured and, if so, what the proper configuration is. To control the execution of servers and tasks, three states are introduced: READY, RUN and IDLE. A server (task) is in the RUN state, when it is currently being executed on the FPGA. A server (task) in the READY state has been released and has not yet terminated, but it is not currently being executed on the FPGA. This implies that another server is currently assigned to the FPGA. Finally, a server (task) enters the IDLE state, when its current instance has terminated. In this case, the server (task) has to wait for its next release.

5.1.3 Task management: The task manager controls the task operations and the context switches. Whenever a reconfiguration is required, the scheduler triggers the task manager to save the entire system context. Similarly, the task manager is called to restore the entire system context after the FPGA reconfiguration. The task manager is comprised a server-independent part and a server-dependent part. The server-independent part is identical in all configuration bitstreams and implements logic to save and restore the context of the runtime system. The server-dependent part implements a logic to start, stop and resume the user tasks and to save and restore their contexts. Since each server contains a different set of user tasks, and each user task has a different context size, we apply server-specific optimisations in the compilation process.

5.1.4 MMU: The memory management unit is an optional component. In a runtime system without MMU, the task manager connects directly to an SRAM bank. This SRAM is then exclusively used as intermediate context storage. The second SRAM bank is devoted to implement data storage for the user tasks. Using the MMU [21], the SRAM banks are divided into pages which can be shared among the runtime system and the user tasks. Sharing the same page among two or more tasks could also be used for intertask communication.

5.2 Scheduler data structures

Executing an MSDL schedule means applying the sequential EDF scheduling policy to a precomputed set of servers. To this end, the scheduler implements the data structures shown in Fig. 6.

The server control block array (SCBarray) holds for each server a control block (SCB) with the following constant entries: server ID, computation time, period, the address of the corresponding bitstream file in the flash memory and the set of tasks (stored as a bit-mask) which constitute the server. The variable part of the SCB stores the server’s current state, its absolute deadline, its remaining computation time and a pointer to another SCB.

In general, the task periods may differ from the server periods. Tasks can start and terminate at arbitrary times within their servers. Hence, also the task states can differ from those of the servers, requiring the scheduler to implement another data structure for tasks. The task control block array (TCBarray) holds for each task a control block (TCB) with the following constant entries: task ID, period and the size of the context. The variable fields of the TCB are the task’s current state and its absolute deadline.

The ready queue is used to store a list of servers in the ready state, sorted according to non-decreasing absolute deadlines. We do not implement the ready queue directly, but provide a register, readyQueueHead, that points to the entry in SCBarray holding the server with the currently earliest deadline. Additionally, the pointer in the variable part of SCBarray points to the next server in the ready queue. Finally, the register runningServer contains the ID of the server in execution.

5.3 Scheduler cycle

Time proceeds in system ticks. The period of the system tick corresponds to a user-defined number of clock cycles which is constant during the application’s runtime. A scheduling cycle denotes all the operations of the scheduler during a system tick period.

There are basically two different sequences of scheduler operations during a cycle, depending on whether a device

![Fig. 5 System architecture](image_url)

![Fig. 6 Scheduler data structures](image_url)

![Fig. 7 Scheduler phases during a system tick period](image_url)
reconfiguration is needed or not. These two cases are outlined in Fig. 7.

In any case, the scheduler cycle starts with the Run Tasks (RT) phase. Here, all tasks that are included in the current server and are in the READY state are set to the RUN state. Consequently, these tasks receive a signal to start or resume their execution. In the example of Fig. 7, S1 starts execution. Then, the scheduler idles until almost the end of the system tick period. During this time interval, the user tasks are executed and some may terminate.

The first scheduler phase after the idle time is called end cycle. Here, the scheduler decreases the remaining computation time of the currently running server. If the remaining computation time reaches zero, the server has finished its execution for its current period and its state is set to IDLE.

In the following wake up phase, the scheduler checks if any server or task has reached the end of its period. This is done by comparing the absolute deadlines to the system tick. If a server or task is at the end of its period, it is set to the READY state and receives a new deadline. For servers, the remaining computation time is set to the server’s initial computation time.

The dispatch server (DPS) phase increments the system tick and determines the server that should run in the next system tick period following the EDF rule. Since the queue of ready servers is sorted according to non-decreasing deadlines, DPS only compares the deadline of the running server to that of the first server in the ready queue. If the currently running server terminates or has to be preempted, the first server in the ready queue becomes the next running server. In this case, shown in Fig. 7a, the save system context (SSC) phase is entered. If the currently running server is allowed to continue execution, shown in Fig. 7b, an idle phase is entered.

If a device reconfiguration is intended, the SSC phase requests all currently running tasks to stop their execution. Then, the contexts of all user tasks are stored in the external SRAM, followed by the context of the runtime system. Upon completion of the context saves, the scheduler triggers the FPGA reconfiguration and the subsequent RC phase is entered. After reconfiguration, the scheduler executes the load system context (LSC) phase and restores first the context of the runtime system and, afterwards, the contexts of all tasks in the new server. The following RT phase starts the next scheduler cycle.

The total time required by the SSC, RC, LSC and RT phases is the runtime system overhead as it cannot be used for user task execution. This overhead actually combines overheads because of device reconfiguration (RC), task and runtime system context saves and restores (SSC, LSC) and runtime system management overhead (RT).

In case no device reconfiguration is intended, the scheduler enters another idle phase after the DPS phase. The duration of this delay matches exactly the sum of the SSC, RC and LSC phases. By this, we ensure a constant time interval for each system tick. As can be observed from Fig. 7, the runtime system overhead only occurs if the system switches from one configuration to another. If a server executes for several system ticks, which can be expected to be the regular case, there is no runtime overhead at all.

The presented scheduler cycle assumes that a system tick period is larger than the reconfiguration time. While this constraint on the system tick period simplifies time keeping and scheduling, it also limits the system’s time resolution. An alternative implementation could make use of an external, non-reconfigured real-time clock as a time reference and reduce the system tick period considerably.

5.4 Task preemption

Our execution model assumes preemptive multitasking. The runtime system must be able to interrupt a running server with its running tasks and to save their state. Later on, when a preempted server is scheduled again for execution the previously saved state has to be restored and the tasks must be resumed. Technically, such preemption mechanisms have been demonstrated using two different approaches:

The first approach realises task preemption by stopping the clock and reading back the FPGA configuration. The FPGA configuration includes the state information of all storage elements. This state information is then used to initialise the storage elements before the task is resumed [13]. The benefit is that user tasks can stay unmodified and no extra logic is required to implement preemption.

The second approach adds to each task extra logic and an interface by which the task can be stopped, started and its internal state registers can be read and written. Although the extra logic has to be manually inserted into each task, we follow this approach because it gives us several advantages: first, we do not require the readback capability and are independent of the FPGA bitstream format. Second, the approach is fast since only the actual task state information is written and read, compared to the much larger amount of state data in the entire FPGA bitstream. Finally, the technique is independent of the user task’s position on the FPGA. The MSDL technique may schedule a user task into different servers. Implementing task preemption by readback would require us to keep the positions of the state elements not only for each task, but also for each pair of server and task.

We measured the logic overhead needed to implement the context save and restore mechanism. We organised the context registers in the form of a shift register chain and counted the number of additional LUTs required to implement the save and restore functions, depending on the register size. The results reveal an overhead of two LUTs per bit for a context register of 32 bit, decreasing to 1.2 LUTs per bit for context register sizes of 256 and more.

6 Synthesis tool flow

Fig. 8 shows the basic steps of the synthesis tool flow that generates the configuration bitstreams for a particular application. We use CELOXICA’s parallel programming language Handel-C to specify both the runtime system and the user tasks.

The application, that is, the task set Γ, is specified in textual form (appl.txt) and contains the period, worst-case computation time and area requirement for each user task. The first step of the tool flow calls the MSDL algorithm to compute a feasible set of servers Ω (cmp. Section 3). If such a feasible set of servers is found, the relevant task and server parameters are written to the Handel-C header file task_serv_data.hch. This file contains mainly preprocessing macros, defining several properties of the servers and tasks such as the IDs and periods. Furthermore, for each server the computation time, the set of included tasks and the address of the configuration bitstream in the flash memory card is defined.

The second step of the tool flow takes this header file together with the Handel-C source files for the user tasks and the runtime system and creates the individual
configuration bitstreams for the server tasks. We use the Handel-C compiler (CELOXICA DK4) and standard FPGA vendor tools (XILINX ISE 7.2) for this step.

Besides the \( m \) servers generated by MSDL, we create another bitstream for an idle server (server_0). The idle server contains only the runtime system but no user tasks. It is run as initial configuration after system startup and when all other servers are idle. When the synthesis tool flow is straight-forward.

The scheduler takes over control and decides which server to reconfigure next according to the MSDL schedule. The scheduler allows the runtime system to start and stop a task and to initiate context save and restore operations. Although our current prototype expects user tasks specified in Handel-C, integrating other HDL components into the synthesis flow is straightforward.

7 Results

We implemented the entire system on CELOXICA’s RC203 board, which hosts a XILINX Virtex II 3000 FPGA, two 2 MB SRAM banks and a 64 MB Smart-Media flash memory card. An FPGA configuration bitstream has a size of approximately 1.25 MB, which allows us to store up to 50 server bitstreams.

To determine the area requirement for the runtime system, we conducted synthesis experiments with different numbers of tasks and servers. These experiments use a runtime system without MMU and minimal user task implementations in order to reduce the effect of the user task logic on our measurements.

The results are presented in Table 2. The number of required LUTs grows with the number of servers and ranges from 16% to 45% of the overall LUT resources. The required number of flip flops (FFs) ranges from 5% to 14% of the overall device FFs. Assigning almost 50% of the device resources to the runtime system might look unrealistic at the first glance. However, this number corresponds to the rather high number of 15 servers and 15 tasks, respectively. Then, these figures were derived from a first, unoptimised version of the runtime system. For example, all scheduler data structures are currently implemented as registers. Mapping these structures to block-RAM will reduce the required resources substantially. Finally, in the future we expect that much larger devices will become available, reducing the runtime system overhead for such server sets.

The time overheads for the runtime system are fairly small. With \( s \) as the number of servers and \( t \) as the number of overall tasks we measured a runtime for the SSC phase of \( 3s + 7t \) clock cycles and for the LSC phase of \( 3s + 8t \) cycles, not including context read and write times. The wake up phase takes \( 3s + t \) cycles and, finally, the end cycle and DPS phases take just one clock cycle each. For a runtime system controlling three servers and nine tasks we achieved a maximal clock frequency of about 50 MHz. Although the Virtex II 3000 can be reconfigured within some 20 ms via the SelectMap interface, the speed of the SmartMedia flash memory card limits the reconfiguration time on the RC203 board to 180 ms.

In order to test our synthesis tool flow, we created a proof-of-concept implementation consisting of three user tasks that process and display frames of video data. We extended the runtime system by a video module that grabs video frames of size \( 214 \times 248 \) pixel with a colour depth of 24 bit. The frames are then copied into local buffers provided by the user tasks. The user tasks process these data and store the results again in local buffers. The video module reads all frames from the tasks’ local buffers and displays them on a VGA device. Task T1 implements a colour region detector that scans a video frame for regions of a certain size and colour value interval. Task T2 implements edge detection by a Sobel filter, and task T3 implements an animation effect.

Table 3 shows the LUT, FF and buffer requirements for the runtime system including the video module and for the three user tasks. These data include the context read and write functions. Obviously, all tasks and the video module would fit together on the device - to implement this functionality neither a scheduler nor a runtime system is needed. However, to demonstrate our synthesis tool flow we set the periods, computation times and areas for

<table>
<thead>
<tr>
<th>#tasks</th>
<th>#servers</th>
<th>LUTs (% of device)</th>
<th>FFs (% of device)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>3</td>
<td>7786 (27%)</td>
<td>2703 (9%)</td>
</tr>
<tr>
<td>15</td>
<td>6</td>
<td>9802 (33%)</td>
<td>3028 (10%)</td>
</tr>
<tr>
<td>15</td>
<td>9</td>
<td>10297 (35%)</td>
<td>3390 (11%)</td>
</tr>
<tr>
<td>15</td>
<td>12</td>
<td>11655 (40%)</td>
<td>3725 (12%)</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>13109 (45%)</td>
<td>4030 (14%)</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4470 (16%)</td>
<td>1336 (5%)</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>5297 (18%)</td>
<td>1677 (6%)</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>6123 (21%)</td>
<td>2019 (7%)</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>6948 (24%)</td>
<td>2361 (8%)</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>7787 (27%)</td>
<td>2703 (9%)</td>
</tr>
</tbody>
</table>
Table 3: Synthesis results for the video tasks

<table>
<thead>
<tr>
<th>design</th>
<th>LUTs (% of device)</th>
<th>FFs (% of device)</th>
<th>SRAM (kb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>runtime, video</td>
<td>4053 (14.1%)</td>
<td>1911 (6.7%)</td>
<td>1 × 64 kb</td>
</tr>
<tr>
<td>T1 (color detection)</td>
<td>730 (2.5%)</td>
<td>304 (1.1%)</td>
<td>1 × 64 kb</td>
</tr>
<tr>
<td>T2 (edge detection)</td>
<td>1696 (6.9%)</td>
<td>849 (3.0%)</td>
<td>2 × 64 kb</td>
</tr>
<tr>
<td>T3 (animation)</td>
<td>929 (3.2%)</td>
<td>295 (1.0%)</td>
<td>2 × 64 kb</td>
</tr>
<tr>
<td>server 1 (T1,T3)</td>
<td>5636 (19.7%)</td>
<td>2525 (8.8%)</td>
<td></td>
</tr>
<tr>
<td>server 2 (T2,T3)</td>
<td>6651 (22.9%)</td>
<td>3062 (10.7%)</td>
<td></td>
</tr>
</tbody>
</table>

the user tasks such that MSDL created two servers. Server 1 includes tasks T1 and T3, and server 2 includes tasks T2 and T3. The resource requirements for the servers are also shown in Table 3. The task periods, which determine the video refresh rates, we also set high enough to tolerate the long reconfiguration times of the CELOCIXA’s RC203 board.

8 Conclusion

We presented a system that executes periodic real-time tasks on dynamically reconfigurable hardware. We used the MSDL scheduling technique which offers an offline schedulability test. MSDL decides in polynomial time whether a task set can be feasibly scheduled and creates a schedule with a minimal number of device configurations. We showed an implementation of this technique on an FPGA board, including an all-hardware runtime system and the corresponding synthesis tool flow.

Although we could demonstrate the feasibility of server-based scheduling, our prototype implementation suffers from the long reconfiguration time of the CELOCIXA board. Hence, in the future we plan to port the system to an FPGA platform with shorter reconfiguration time. Reconfiguration times in the order of milliseconds will make our approach amenable to a larger number of applications. Further, we intend to investigate the use of partial reconfiguration. In fact, we consider it a benefit of our approach that it can work with full device reconfiguration, and thus with all SRAM-based devices on the market. However, partial reconfiguration could be used to keep the runtime system permanently in the device and only reconfigure between different servers. This would not only help to reduce the reconfiguration time, but also allow us to use a non-reconfigured real-time clock and control time at a much finer granularity. An alternative implementation could employ an embedded CPU to execute the runtime system.

9 Acknowledgment

This work was supported by the German Research Foundation under project number PL 471/2-1 within the priority program ‘Rekonfigurierbare Rechensysteme’.

10 References

11 Danne, K.: ‘Memory management to support multithreading on FPGA based systems’, Proc. Int. Conf. on Reconfigurable Computing and FPGAs (ReConFig), Mexico City, Computer Science, SMCC, 2004
21 Xilinx LogicCore 1-D Discrete Cosine Transformation V2.1, Xilinx, 2002
302