EXECUTING HARDWARE TASKS ON DYNAMICALLY RECONFIGURABLE DEVICES UNDER REAL-TIME CONDITIONS

Klaus Danne        Roland Mühlenbernd        Marco Platzner

Computer Science Department
University of Paderborn

ABSTRACT
This paper presents a prototype system that executes a set of periodic real-time tasks utilizing dynamic hardware reconfiguration. The proposed scheduling technique, MSDL, is not only able to give an offline guarantee for the feasibility of the task set but also minimizes the number of device configurations. After describing this technique, we extend the schedulability analysis to include different runtime system overheads, including the device reconfiguration time. Then we detail a light-weight runtime system that performs the online part of the MSDL scheduling technique. The runtime system is entirely implemented in hardware. Finally, we outline the corresponding synthesis tool flow and report on the overhead posed by the runtime system.

1. INTRODUCTION
Reconfigurable hardware devices, such as FPGAs, are increasingly used in embedded systems that operate under real-time conditions. In general, mapping real-time computations to reconfigurable hardware is inherently more complex than mapping such workloads to microprocessor-based systems. While real-time scheduling has been intensively studied for microprocessor-based systems [1], the corresponding task scheduling and placement strategies for reconfigurable hardware devices have mostly focused on non real-time application models, e.g., [2, 3, 4, 5].

In this paper, we consider a memory-limited embedded system with a dynamically reconfigurable hardware device as the sole processing element. The system executes a set of periodic real-time tasks. As the overall logic requirement of the task set exceeds the device capacity, we utilize full dynamic reconfiguration to feasibly map all computations to hardware.

The application consists of a set of periodic tasks Γ, where each task \( T_i \in \Gamma \) is characterized by three parameters: the period \( P_i \), the worst-case execution time \( C_i \), and the required amount of hardware resources \( A_i \). The tasks have deadlines equal to periods. The reconfigurable hardware device is characterized by the amount of hardware resources it offers. All hardware resource figures, also denoted as area parameters, are normalized to the device area without loss of generality. Hence, the device offers an area of 1, and the single tasks are assumed to have area requirements \( A_i \in [0 \ldots 1] \).

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

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

For the special case that all tasks are executed sequentially, \( U^T \) is the fraction of time the device spends executing tasks whereas \( 1 - U^T \) is the idle time. As a more expressive metric in case several tasks are executed in parallel, we define the system-utilization factor of a task set \( \Gamma \) as

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

The objectives for devising appropriate scheduling techniques are threefold: First, the scheduler must generate feasible schedules, i.e., 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 a feasible schedule will result at runtime or not. Third, we seek to minimize the number of device configurations (bitstreams) in order to reduce the system’s memory requirements.

In Section 2 we briefly review MSDL, a server-based scheduling technique for periodic task sets. Then, we discuss an extension of the MSDL schedulability analysis to include various runtime overheads. An all-hardware implementation of the runtime system is discussed in Section 3, followed by an outline of the corresponding synthesis tool flow in Section 4. Finally, in Section 5 we report on the overhead posed by the runtime system.

The main contributions of this paper are i) the extension of a previous scheduling technique to include runtime overhead analysis and ii) the design of an all-hardware runtime system and synthesis tool flow.
2. SERVER-BASED SCHEDULING

2.1. The MSDL Technique

Starting from the well-known single processor EDF (earliest-deadline first) scheduling policy, we presented several EDF extensions for reconfigurable hardware in [6]. One of the techniques, Merge Server Distribute Load (MSDL), is especially suited to embedded reconfigurable systems as it works with full device reconfiguration and leads to a very small number of configurations. MSDL bases on the concept of server tasks to construct a schedule. A server is a periodic task that reserves execution time and device area for other tasks. We define a server as $S_i = (R_i, P_i, C_i, A_i)$, where $R_i = \{T_a, T_b, \ldots\} \subseteq \Gamma$ 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 of the server, respectively. The area of a server is set to equal the sum of the areas of tasks represented by the server, $A_i = \sum_{T_k \in R_i} A_k$. Consequently, whenever the server $S_i$ runs, all tasks it represents can run.

The rationale of the MSDL algorithm is to construct a set of servers $\Omega$ from the original task set $\Gamma$, such that any feasible schedule for $\Omega$ implies a feasible schedule for $\Gamma$. More specifically, MSDL constructs a set of servers $\Omega$ by properly merging tasks together for parallel execution. The resulting servers are then scheduled for sequential execution on the device with single processor EDF. Feasibility for the resulting set of servers is thus efficiently checked by the EDF utilization test: $U^T(\Omega) \leq 1$. For further details on how servers are selected for being merged and how the remaining computation times can be determined, we refer to [6].

The example task set in Table 1 illustrates the MSDL principle. The original task set in the top-third of Table 1 shows a time utilization exceeding one, rendering a direct sequential EDF schedule infeasible. MSDL starts by turning all tasks into servers. In the first iteration, the servers $S_1$ and $S_2$ are merged into $S_4$, resulting in a deleted $S_1$ and an $S_2$ with reduced computation time. As the time utilization is still larger than one, MSDL merges the residual $S_2$ and $S_3$ into $S_5$. The final time utilization of one implies a feasible EDF schedule. MSDL tries to group servers together for parallel execution in order to reduce the overall execution time. In other words, MSDL trades an increase in system utilization for a reduction of the time utilization. Fig. 1 presents the resulting schedule for the example.

2.2. Runtime System Requirements and Overheads

An implementation of the MSDL technique requires a dynamically reconfigurable hardware device operated in the full reconfiguration mode and a preemptive runtime system. In a preemptive runtime system, a running instance of a task (server) can be preempted by another task (server) before its completion and, later on, be resumed.

A preemptive runtime system for MSDL will lead to several overheads that should be included in the schedulability test. First, there is a time overhead associated with the reconfiguration process. Reconfiguration times can be rather long with up to some tens of milliseconds for large FPGAs. Therefore, including this overhead in the schedulability analysis is of utmost importance. Let the full device reconfiguration time be denoted as $t_{rc}$. In an MSDL schedule, the set of periodic servers is scheduled by sequential EDF. Hence, we can use a result from EDF theory [1], that bounds the number of times a server $S_i$ can get preempted to

$$N_i = \sum_{S_j \in \Omega} \left\lfloor \frac{P_i}{P_j} \right\rfloor - 1 \quad (3)$$

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 most for $N_i$ times, when it resumes from preemption. We now add the maximum reconfiguration time to the server’s original computation time:

<table>
<thead>
<tr>
<th>$S_i$</th>
<th>$R_i$</th>
<th>$P_i$</th>
<th>$C_i$</th>
<th>$A_i$</th>
<th>$U^T_i$</th>
<th>$\bar{U}^T_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$T_1$</td>
<td>4</td>
<td>2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/4</td>
</tr>
<tr>
<td>$S_2$</td>
<td>$T_2$</td>
<td>6</td>
<td>5</td>
<td>1/4</td>
<td>5/6</td>
<td>5/24</td>
</tr>
<tr>
<td>$S_3$</td>
<td>$T_3$</td>
<td>12</td>
<td>3</td>
<td>3/4</td>
<td>1/4</td>
<td>3/16</td>
</tr>
</tbody>
</table>

Figure 1 presents the resulting schedule for the example.

Table 1. Example server set generated by MSDL

Fig. 1. Schedule for the example of Table 1
\[ \tilde{C}_i = C_i + (1 + N_i) \cdot t_{rc} \]  

Using the increased server computation times \( \tilde{C}_i \), we re-calculate the total time-utilization of the server set \( \Omega \):

\[ \tilde{U}_T(\Omega) = \sum_{S_i \in \Omega} \frac{\tilde{C}_i}{P_i} \]  

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, i.e., the context save and store operations. Similar to [7], our implementation of the runtime system requires the single tasks to include logic that stores/loads the context to/from an external memory (see Section 3). 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 Eq. 4.

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 due to loading and storing the context of the runtime system, is again added to the server execution times.

### 3. SYSTEM ARCHITECTURE

#### 3.1. An All-Hardware Runtime System

We have 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, minimize the runtime overhead and deliver maximum scheduling performance. In a processor-based real-time runtime system, a periodic timer triggers a kernel function that maintains time in terms of system ticks. The smaller the timer period is, the finer is the system’s time resolution but the higher is the runtime overhead. 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 optimization of parts of the runtime system to specific servers. On the other hand, we also have to save and restore the context of the runtime system.

Our prototype is based on a CELOXICA RC203 board. Fig. 2 shows the system architecture consisting of an FPGA, a CPLD, a flash memory card and two banks of SRAM. The CPLD connects to both FPGA user I/Os and the FPGA configuration port, as well as to the flash memory card. This way, the CPLD acts as 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. 2 comprise the timer, the scheduler, the task management and an optional memory management unit (MMU).

**Timer:** The timer provides the time reference for the system 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.

**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 executing on the FPGA. A server (task) in the READY state has been released and has not yet terminated, but it is not currently executing 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.

**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 of
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 (speckled objects in Fig. 2) implements 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 optimizations in the compilation process.

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 an MMU, such as the one described in [8], the SRAM banks are divided into pages which can be shared among the runtime system and the user tasks.

3.2. 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 several data structures. The server control block array (SCBarray) holds for each server the following constants: ID, period, computation time, the address of the corresponding bitstream file in the flash memory, and the set of tasks which constitute the server. The variable part of the data structure stores the server’s current server state, its absolute deadline, and its remaining computation time. The register runningServer contains the ID of the server in execution.

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 the following constants: ID, period, and the size of the context. The variable fields 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 single register, readyQueueHead, that points to the entry in SCBarray holding the server with the currently earliest deadline. Additionally, each element of SCBarray contains a field which points to the next server in the ready queue.

3.3. The 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 reconfiguration is needed or not. These two cases are outlined in Fig. 3.

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. As a consequence, these tasks receive a signal to start or resume their execution. In the example of Fig. 3, S1 starts execution. Then, the scheduler idles until almost the end of the system tick period. During this time interval, the user tasks execute and some may terminate.

The first scheduler phase after the delay 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 scheduler phase wake up 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 its deadline is set to $d_i = d_i + P_i$. In case of a server, 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. 3-(a), the Save System Context (SSC) phase is entered next. If the currently running server is determined to continue execution, shown in Fig. 3-(b), a delay phase is entered next.

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 sever running. The following RT phase starts the next scheduler cycle (system tick period).

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

In case no device reconfiguration is intended, the scheduler enters another delay 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 taken from Fig. 3, 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 the time keeping and scheduling processes, 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.

4. SYNTHESIS TOOL FLOW

Fig. 4 shows the basic steps of the synthesis tool flow that generates the configuration bitstreams for a particular application. We use CELOXICA’s hardware description language Handel-C to specify the runtime system and the user tasks.

The application, i.e., the task set $\Gamma$, 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 $\Omega$ (cmp. Section 2). If such a feasible set of servers exists, the relevant task and server parameters are written into the Handel-C header file task_serv_data.hch. This file contains mainly preprocessor macros, defining the tasks’ data and for each server the ID, period, computation time, set of included tasks, and the address of the configuration bitstream in the flash memory card.

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. This server is run as initial configuration after system startup and when all other servers are idle. When the synthesis tool flow is completed, all $m + 1$ server bitstreams are stored onto the flash memory card at their respective addresses. After system power-on, the CPLD automatically configures the idle server into the FPGA. The scheduler takes over control and decides which server to reconfigure next according to the MSDL schedule.

User tasks are free to use logic, memory and I/O resources of the FPGA, as long as they implement the pre-defined interface to the runtime system. This interface 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.

5. RESULTS

We have 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 SmartMedia flash
memory card. An FPGA configuration bitstream has a size of approximately 1.25MB, which allows us to store up to 50 server bitstreams on the flash memory card.

To determine the area requirement for the runtime system, we have 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) grows slower and 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 first glance. However, this number corresponds to the rather high number of 15 servers and 15 tasks, respectively. Then, these figures have been derived from a first, unoptimized 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 overhead for such a runtime system.

<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 2. Runtime system size for various numbers of tasks and servers

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. 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 3 servers and 9 tasks, we achieved a maximal clock frequency of about 50 MHz. Although the Virtex II 3000 could 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.

We further measured the overhead of the logic inside the user tasks needed for storing and loading the tasks’ contexts by synthesis experiments. We organized the context regis-

ters in form of a shift register chain and measured the number of additional LUTs required to implement the load and store functions. The results reveal an overhead of 2 LUTs per bit for a context register of 32 bit. This overhead decreases to 1.2 LUTs per bit for context register sizes of 256 and more.

6. CONCLUSION

In this paper, we presented a system that executes periodic real-time tasks on dynamically reconfigurable hardware. We used the MSDL scheduling technique which offers an efficient offline schedulability test and requires 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.

For the future, we plan to port the system to an FPGA platform with shorter reconfiguration time. Although 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, we also intend to investigate partial device reconfiguration. A natural way to utilize partial reconfigurability is to keep the runtime system permanently in the device and only reconfigure between different servers. Alternatively, an embedded CPU could be used to implement the runtime system.

7. REFERENCES