MORP: Makespan Optimization for Processors with an Embedded Reconfigurable Fabric
Artjom Grudnitsky, Lars Bauer, Jörg Henkel
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{grudnitsky, lars.bauer, henkel}@kit.edu

ABSTRACT
Processors with an embedded runtime reconfigurable fabric have been explored in academia and industry started production of commercial platforms (e.g. Xilinx Zynq-7000). While providing significant performance and efficiency, the comparatively long reconfiguration time limits these advantages when applications request reconfigurations frequently. In multi-tasking systems frequent task switches lead to frequent reconfigurations and thus are a major hurdle for further performance increases. Sophisticated task scheduling is a very effective means to reduce the negative impact of these reconfiguration requests. In this paper, we propose an online approach for combined task scheduling and re-distribution of reconfigurable fabric between tasks in order to reduce the makespan, i.e. the completion time of a taskset that executes on a runtime reconfigurable processor. Evaluating multiple tasksets comprised of multimedia applications, our proposed approach achieves makespans that are on average only 2.8% worse than those achieved by a theoretical optimal scheduling that assumes zero-overhead reconfiguration time. In comparison, scheduling approaches deployed in state-of-the-art reconfigurable processors achieve makespans 14%–20% worse than optimal. As our approach is a purely software-side mechanism, a multitude of reconfigurable platforms aimed at multi-tasking can benefit from it.

Categories and Subject Descriptors
C.1.3 [Other Architecture Styles]: Adaptable architectures

Keywords
Reconfigurable Processor; Area Allocation; Task Scheduling

1 Introduction and Motivation
Reconfigurable processors allow hardware adaption to the computational profile of running applications, resulting in a platform which – unlike a processor with an application

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FPGA'14, February 26–28, 2014, Monterey, CA, USA.
Copyright 2014 ACM 978-1-4503-2671-1/14/02 ...$15.00.
http://dx.doi.org/10.1145/2554688.2554782.

specific instruction set (ASIP) [13] – is capable of accelerating a wide range of applications [8]. A widespread approach is to extend a non-reconfigurable general-purpose processor (core processor) with a reconfigurable fabric, onto which application-specific accelerators can be loaded at runtime (e.g. [26, 18, 5, 7, 17, 2]). The fabric is implemented on an embedded FPGA [23], while the core processor and other SoC components are implemented as an ASIC on the same chip. Figure 1 shows a typical architecture for such a reconfigurable processor and a possible floorplan with the embedded FPGA. A commercial example of such a reconfigurable platform is the Xilinx Zynq-7000 processor [27], which uses a dual-core ARM Cortex-A9 connected to 7-Series reconfigurable fabric on the same chip.

The instruction set architecture (ISA) of the core processor is extended with Special Instructions (SIs), which are typically used in computationally intensive kernels of the applications and provide access to the accelerators on the fabric. The latency (execution speed in cycles) of an SI typically ranges from a few cycles up to a few hundred cycles and an SI is typically executed several hundreds to thousands times per kernel (depending on the application and input data).

The high performance and flexibility of fine-grained reconfigurable accelerators comes at the cost of a considerable reconfiguration time which is typically in the range of milliseconds for complex accelerators [21]. Reconfiguration is performed using a DMA transfer from the memory location
where the accelerator bitstream is stored to the reconfiguration port of the fabric, thus the processor can continue executing applications while an accelerator is loaded. Until reconfiguration of an accelerator is complete, the application cannot use that accelerator but has to fallback on a slower SI implementation. The SI functionality can be emulated in software by using the normal ISA of the core processor, triggered either by an unimplemented instruction trap or by a conditional branch that queries whether the accelerator finished reconfiguration. That means that at least two implementations are available per SI.

We call the latency difference between the fastest possible SI implementation (selected by the runtime system of the reconfigurable processor but not necessarily reconfigured yet) and the fastest currently reconfigured SI implementation (potentially the software emulation) Reconfiguration-induced Cycle Loss (RiCL). An application exhibiting a high cumulative RiCL will take longer to finish than when running with a low cumulative RiCL. RiCL depends on the time required to complete the reconfiguration of an accelerator. The slower reconfigurations are performed, the higher RiCL.

Figure 2 presents a motivational case study that shows the RiCL over time for an H.264 Video Encoder focusing on one outer loop iteration that encodes one video frame. For each frame, the application executes three kernels after each other and each kernel uses a different set of SIs (9 different SIs are used in this example and each SI is typically executed several hundreds to thousands times per kernel). Therefore, each kernel triggers reconfigurations of multiple accelerators when it starts. In Figure 2, the starting time of the three kernels is indicated as (1), (2), and (3) and all times on the X-axis are relative to the first kernel when the application starts encoding the frame. The height of the bar at a particular point on the X-axis corresponds to the RiCL of the SIs that execute during that time (potentially zero, e.g. when no SI executes or all reconfigurations are completed) averaged over a sliding time window of 1000 cycles. Right at the beginning of the first kernel (see (1) in Figure 2), RiCL is as high as several accelerators that are beneficial for this kernel are not yet reconfigured, because the third kernel of the preceding frame just finished and the reconfigurable fabric still contains the accelerators beneficial for that kernel. As more accelerators are reconfigured for the first kernel of the new frame, RiCL reduces to zero until the second kernel starts and requests different accelerators.

Summarizing, applications that are composed out of multiple kernels always face a reconfiguration-induced cycle loss when they switch from one kernel to another (unless the reconfigurable fabric is large enough to accommodate all accelerators at the same time, which is not the case for complex applications as H.264). The green dashed line in Figure 2 shows the theoretical time when frame encoding would be finished, if the cumulative RiCL were zero (1.35 × faster for this frame). This is the potential that can be exploited by reducing RiCL.

If the system does not support multi-tasking, or only one task is executable, then RiCL can only be reduced by further improving the reconfiguration speed (we assume 66 MB/s average bandwidth for reconfiguration). However, if the task scheduler has multiple candidates that are executable, then the reconfiguration overhead of a task A can be hidden, by running a different task B while the accelerators for task A are reconfigured. But simply switching to a different task does not necessarily improve overall performance. For instance, if task A has been granted the entire reconfigurable fabric, switching to task B would actually worsen overall performance, as B would run without any accelerators and thus with a very high RiCL.

We therefore propose a novel combined task-scheduling and fabric-distribution approach called MORP (Makespan Optimizer for Reconfigurable Processors), that minimizes the makespan, i.e. the completion time of a taskset that executes on a runtime reconfigurable processor. To do so, upon anticipation of a drop in execution speed of the currently executing primary task (when switching from one kernel to another) our combined approach dynamically selects a suitable secondary task, grants it a small share of the reconfigurable fabric (fabric re-distribution), and switches execution to the secondary task (task scheduling) until the primary task completed most of its reconfigurations. This effectively hides the reconfiguration latency of the primary task at a reasonable performance (and thus reasonable RiCL) of the secondary task, thus reducing the overall RiCL of the taskset. The challenge is to identify suitable primary and secondary tasks and to identify when to switch between them and which share of the reconfigurable fabric shall be granted to the secondary task.

2 Related Work

Surveys about reconfigurable processors are available that provide good introductions as well as classifications [8, 25, 10]. Whereas early reconfigurable processors focused on accelerating single-tasking applications, recent approaches target multi-tasking scenarios. A recent survey categorizes them into reconfigurable processors with implicit, explicit, and no architectural support for multi-tasking [28].

Several reconfigurable processors with support for multi-tasking implement entire tasks either in hardware or in software. The hardware tasks execute in parallel whereas the software tasks execute sequentially on the core CPU without acceleration. A hierarchical approach to dynamically decide which task shall execute on the reconfigurable fabric is proposed in [19]. An infrastructure for transparent inter-task communication – independent on whether a task executes in hardware or software – is presented in [16]. [20] proposes OS extensions for support of reconfigurable processors, among them a hardware module that performs thread scheduling. Tang et al. [24] present a scheduler that assigns periodic tasks to heterogeneous processing elements (including a re-
configurable fabric) offline and that integrate sporadic tasks by extending the resulting schedule at runtime. However, all these approaches perform a binary decision whether to implement a task entirely in hardware or software without acceleration, i.e. some tasks are significantly accelerated and other tasks are not accelerated at all. Additionally, they perform a fixed fabric distribution per task. Each task that shall execute in hardware obtains a fixed share of the reconfigurable fabric that is typically reconfigured once when the task starts rather than being adapted to the requirements of a task that may change when executing different kernels.

Due to the limited flexibility and efficiency of such a binary decision, several reconfigurable processors use software tasks with a focus on the performance-critical parts of computational kernels that are accelerated by Special Instructions (SIs), e.g. [26, 18, 5, 7, 2]. This allows for a high efficiency as it provides several options for hardware acceleration, i.e. the decision whether an SI shall be implemented in hardware or software can be determined independent of other SIs and other tasks.

RISPP [5] presents an adaptive reconfigurable processor where a runtime system selects among different compile-time prepared SI implementations at runtime depending on the application requirements. However, it does not support multi-tasking. Proteus [7] proposes a mechanism that allows to preempt SIs during their executions (e.g. to handle interrupts or to switch to another task) and resume their execution later. However, they do not propose how to optimize the task schedule or the fabric distribution to reduce the reconfiguration-induced cycle loss. [15] presents an online task-scheduler and fabric allocator for a reconfigurable processor. The authors formulate the scheduling and allocation problem as a 2D model, where a task is defined by its runtime and required fabric area. This approach requires knowing the runtime of a task before it starts and only supports tasks that perform all their reconfigurations at the start of a task, which limits its applicability.

The reconfigurable multi-core processor presented in [2] does not use a task scheduler as at most one task executes per core. The reconfigurable fabric is allocated based on the deadlines of the parallel executing applications. Frequent fabric re-distributions are performed whenever any of the applications proceeds to its next kernel, which leads to frequent reconfigurations and thus exacerbates the need to reduce the reconfiguration-induces cycle loss. [6] proposes a runtime mapping approach of multiple tasks to a runtime-reconfigurable system, aiming to reduce reconfiguration latency. The approach aims to reuse existing configurations and thus reduce the amount of configurations during deployment of a new application, focusing mainly on the allocation/mapping of fabric, not on task scheduling. Reconfiguration latency reduction is also the goal of [21], where a design-time approach is used to map multiple applications onto the fabric. A kind of reconfiguration hiding has been explored in [11] and [4] for soft real-time systems, thus their main goal was minimizing deadline misses or tardiness (i.e. accumulated time by which tasks miss their deadlines). None of these approaches aim at minimizing the makespan of a taskset or aim at reducing the reconfiguration-induced cycle loss.

A compiler-assisted scheduling approach for the MOLEN reconfigurable platform [26] is presented in [22]. The compiler provides information about the temporal distance between kernels to the runtime system, a technique which is also utilized by our approach. For the actual task scheduling, the approach in [22] uses Round Robin and we compare our approach with [22] in the results. For a non-reconfigurable single-core processor, Shortest Processing Time (SPT) scheduling is optimal for makespan minimization [14]. This is not the case for reconfigurable processors as we demonstrate with our comparison in the results.

3 System Definition

3.1 Architecture & Application Assumptions

This work targets reconfigurable processors, capable of loading hardware accelerators at runtime onto a reconfigurable fabric. The fabric is partitioned into reconfigurable containers, i.e. designated regions on the fabric where accelerators can be loaded. Figure 1 shows a simplified example of such an architecture where the processor pipeline and the reconfigurable fabric are tightly coupled to accelerate applications. The containers are interconnected using the 1D fixed-size area model [12] and they are implemented on an embedded FPGA. Loading an accelerator takes a considerable amount of time (in the order of milliseconds, depending on accelerator complexity), but can be performed without stalling the pipeline using a DMA transfer.

Applications that are implemented for reconfigurable processors typically have computationally intensive kernels that benefit from hardware acceleration. They are accelerated by invoking Special Instructions (SIs) that are implemented in multiple alternatives: one software implementation and at least one hardware implementation (using the reconfigurable accelerators). In general, applications run faster when more reconfigurable fabric is allocated to them (until a certain amount of parallelism is utilized).

The operating system (also called runtime system) is taking care of task scheduling and management of the reconfigurable fabric. The decision which SIs shall be implemented in hardware (given a certain amount of reconfigurable containers) is performed at runtime and can be determined automatically by a runtime system (as in [17, 5, 2]) or explicitly by the application (as in [26, 18]).

3.2 Task Properties

We assume the following task properties and requirements to the system (for in-depth details on scheduling background, see [14]): The problem is to schedule a taskset $T$, consisting of tasks $T_i$. Each task $T_i$ has a finite Completion Time $C_i$, i.e. the time when $T_i$ finishes its execution. The optimization goal of MORP is to minimize the makespan (see Eq. (1)), i.e. the time when all tasks of the taskset completed.

$$makespan := \max \{ C_i \mid \forall \text{Tasks } T_i \in T \}$$  \hspace{1cm} (1)

We assume that the system supports task preemption, i.e. that the currently executing task can be interrupted at any time by the operating system, so that the task scheduler can decide which task to run next, and switch to a different task if required.

Before a task can use the reconfigurable fabric, the application issues a prefetch system call that triggers the reconfigurations of the accelerators. Prefetches are used to inform the runtime system about the SIs the application is about to execute. The runtime system can then decide which SI implementation (software or one of the hardware implementations) to use and start loading the corresponding
accelerators as in [5]. Alternatively, prefetches can directly contain the information which accelerators shall be configured as in [26]. Once a task has finished all its hardware accelerated kernels, it can use a designated prefetch called **last prefetch** to inform the runtime system that it will not execute further SIs. This last prefetch can be added to the application in the same manner as regular prefetches, e.g. by the compiler or a profiling tool.

Depending on the kernel execution behavior, MORP assigns tasks into one of the following three task categories. This categorization is done offline using a profiling tool.

**Multi-Kernel Task (MKT):** An MKT executes more than one hardware accelerated kernel (i.e. kernels that are accelerated by invoking SIs) over its lifetime. The kernel execution can happen periodically, e.g. the H.264 Video Encoder example in Figure 2 executes three different kernels sequentially to process a frame.

**Single-Kernel Task (SKT):** An SKT executes one single hardware accelerated kernel and one or more SIs. After an MKT or SKT executes its dedicated last prefetch, it behaves like a ZKT.

### 3.3 Required Metrics

In addition to the task category, MORP uses further task-specific metrics for its decisions. Tasks benefit to a different degree from the reconfigurable fabric. ZKTs run at the same speed for all fabric sizes (i.e. number of reconfigurable containers), SKTs and MKTs will benefit from increasing fabric sizes up to a saturation point (as is shown in Figure 5 in Section 5). This fabric-size dependent characteristic is called **Task Performance**. Task performance is profiled offline by executing the task alone (i.e. no multi-tasking) on different fabric sizes (i.e. allocating different numbers of containers to it). The relative task performance \( RTP_{i,n} \) of task \( T_i \) on a fabric of size \( n \) is defined as shown in Eq. (2), where \( C_{i,j} \) is the completion time of \( T_i \) executing in single-tasking mode on a fabric of size \( j \).

\[
RTP_{i,n} = \frac{C_{i,n}}{C_{i,0}}
\]

The relative task performance is normalized into the range \([0,1]\) once the task enters the system to allow comparing performances of different tasks. The normalized task performance \( TP_{i,n} \) is shown in Eq. (3), where \( N \) is the fabric size of the actual system.

\[
TP_{i,n} = \begin{cases} 
1.0 & \text{if } n \geq N \\
RTP_{i,n}/RTP_{i,N} & \text{otherwise}
\end{cases}
\]

Tasks that use accelerators (i.e. MKT and SKT) run at different speeds depending on how many of the prefetched accelerators already finished reconfiguration. When no prefetched accelerators are available (e.g. short time after the prefetch was issued) the speed is low which leads to a large RiCL as shown in Figure 2. To make these different speeds comparable among different tasks, we use the relative **Task Efficiency** metric, which is defined as follows: When the current prefetch for a task is completed, i.e. there are no pending reconfigurations for this task, then the task efficiency for this task is defined to be 1.0. If some reconfigurations are not yet completed for this task, then not all SIs of the currently executing kernel can be executed in the targeted implementation (decided by the runtime system or directly determined by the prefetch instruction) and thus the task efficiency is lower. Let \( SI.C \), \( SI.T \), and \( SI.SW \) be the latency (i.e. execution time in cycles) of the current, target, and software implementation of \( SI_i \), respectively, then the following relation holds: \( SI_i.T \leq SI_i.C \leq SI_i.SW \). The SI latencies are weighted by the average number of executions for this SI \( w_i \) (from offline profiling). To compute the task efficiency for kernel \( K_x \), first \( \Delta L \) is computed as the accumulated weighted difference between the latency of the current SI implementations and the target SI implementations for all \( SI_i \in K_x \). The worst possible latency for the prefetched SI, i.e. the latency of their software implementations, constitutes \( L_{\text{worst}} \). The task efficiency \( TE \) is then defined as in Eq. (4).

\[
\Delta L = \sum_{SI_i \in K_x} w_i(SI_i.C - SI_i.T) \\
L_{\text{worst}} = \sum_{SI_i \in K_x} w_i SI_i.SW \\
TE = 1.0 - \frac{\Delta L}{L_{\text{worst}}}
\]

The concept of task efficiencies is inspired by [4]. The difference between task performance and task efficiency is that task performance is a static characteristic (usually implemented as a lookup table), while task efficiency is a dynamic value, changing with each prefetch and with each reconfiguration that loads or removes accelerators beneficial for the currently executing kernel.

If a task does not run at an efficiency of 1.0, then \( \Delta L \) is larger than 0. We call \( \Delta L \) the **Reconfiguration-induced Cycle Loss (RiCL)**, compared to the task running at maximal efficiency. The RiCL metric denotes how many cycles a task losses accumulated over all its SIs over all its execution time due to reconfiguration delays compared to a hypothetical situation where the accelerators that are requested by a prefetch would be immediately available (i.e. under the assumption that the reconfiguration would happen instantaneously). Reducing the RiCL is the goal of our proposed scheduling approach.

For each kernel \( K_x \) that performs a prefetch, the **Average Time Between Prefetches (ATBP)** determines the time between that prefetch and the following prefetch. The ATBP is determined offline by a profiling tool.

### 4 MORP: Scheduling and Fabric Allocation

The overall idea is to hide the prefetches of a primary task by executing a secondary task, while the reconfigurations for the primary task are performed. To do so, the system selects a secondary task when the primary task is approaching its next kernel, i.e. its next prefetch. Tasks that do not execute SIs (i.e. ZKTs) or tasks that reach a high task efficiency with only few reconfigurable containers are good candidates for secondary tasks. They may also receive a small amount of reconfigurable containers (called *reallocations*) before the primary task reaches its next kernel to run much more efficiently, at the cost of slightly reduced performance of the primary task. When the primary task issues its next prefetch, the system switches to the secondary task while the prefetch for the primary task is performed. The RiCL of the primary task is reduced (because its reconfiguration time is hidden) without significantly increasing the RiCL of the secondary task, thus a net reduction of RiCL is achieved and thereby the makespan is reduced.

Figure 3 shows the flow-chart of our approach, MORP. The system is usually either in the **Primary Task Execution** state or the **Secondary Task Execution** state.
First, a primary task \( pt \) is selected from the current taskset \( T \). The primary task is preferably a MKT, characterized by a high RiCL and is selected using the following method:

1. Select MKT with highest RiCL from \( T \). If no MKTs are left in \( T \), then
2. Select SKT with highest RiCL from \( T \). If no SKTs are left in \( T \), then
3. Select any ZKT from \( T \). If no ZKTs are left in \( T \), then

Taskset \( T \) has been finished, exit.

The system allocates the entire fabric to \( pt \), starts it, and enters the Primary Task Execution state (see Figure 3).

During compilation and profiling, each program has been annotated with ATBP (see Section 3.3), which is used to start selection of the secondary task. MORP uses the average time required to load one accelerator as the basic time unit. In the following, time specifications such as “\( N \) reconfigurations” (see also Figure 3) mean “the time required to load \( N \) accelerators”.

\( N \) reconfigurations before the next prefetch of the primary task \( pt \), the system decides what to do after that prefetch. It either continues executing \( pt \) as primary task (potentially switching to a secondary task for a short time to hide the reconfiguration latency of the prefetch) or it switches to a different primary task \( pt' \). \( N \) is chosen as half the size of the reconfigurable fabric, i.e. when deciding to switch to a new primary task \( pt' \), then there is sufficient time to reconfigure half of the reconfigurable fabric towards \( pt' \) to reduce its RiCL. MORP selects the task with the highest efficiency as the next primary task. A good candidate for \( pt' \) with high efficiency is a task that can utilize accelerators that are currently loaded on the fabric.

If the efficiency of \( pt' \) is higher than the threshold \( TH_{E_1} \), \( pt' \) is provided as the only input candidate to the “Select Secondary Task” function (see Figure 3). Due to its already high efficiency, \( pt' \) will be selected by the “Selected Secondary Task” function, and additional fabric will be reallocated to \( pt' \) to improve its efficiency.

\( TH_{E_1} \) needs to be fairly high, as primary tasks tend to use a lot of accelerators (in general equal to the number of reconfigurable containers). Switching to a new primary task with low efficiency would require multiple reconfigurations until the new task reaches full efficiency. During this time RiCL would be high, thereby increasing the makespan. If the efficiency of \( pt' \) is lower than the threshold \( TH_{E_1} \), a secondary task is selected out of all MKTs and SKTs (except the current primary task) to hide the reconfiguration latency of the primary task.

Secondary task selection is shown in Algorithm 2. The goal is to select the next task \( nt \) (along with the number of accelerators \( a \) that shall be reallocated to it) that provides the highest total performance for the primary task \( pt \) and the secondary task \( nt \). Taking performance for both tasks into account is crucial, as reconfigurations for \( nt \) start while \( pt \) still executes, thus \( pt \) will run with reduced efficiency for a short time until switching to \( nt \). The performance of all kernels of all tasks for different fabric sizes is known from offline profiling and is used in Algorithm 2 to search an appropriate secondary task. The task that maximizes performance is returned as the secondary task \( nt \), along with the number of accelerators \( a \) that are required for providing that performance. If reallocation would cause the performance of either the secondary or the primary task to drop below the threshold \( TH_P \), reallocation of reconfigurable fabric is not considered an option. A secondary task may still be used later, but without reallocation.

If the parameter \( TH_P \) is chosen too high, it would discard most secondary task candidates, e.g. if set to 1.0, no efficiency loss for the primary task would be tolerated, thus only very few (if any) containers would be considered for reallocation to the secondary task. The secondary task would have to reach maximum performance with those few containers. Meeting such tough constraints is almost impossible, thus the system would not perform any reallocation at all. Setting \( TH_P \) too low would allow for reallocation of too many containers to the secondary task. The primary task usually...
Algorithm 1: Select Next Task

**input:**
- Taskset $T$
- Previously/initally selected secondary task $nt$
- Originally executed primary task $pt$
- Efficiency Threshold for primary task $TH_{E2}$
- Efficiency Threshold for secondary task $TH_{E3}$

**output:** next task to execute

// Resume to the primary task pt when its efficiency is good if no outstanding reconfigurations or $pt.efficiency > TH_{E2}$ then return $pt$

// Execute those tasks that are awaiting their first prefetch
next_task := MAX{$t.issued | t \in T.MKT \cup T.SKT$ and not $t.issued.prefetch$}
if next_task $\neq$ NULL then return next_task

// Execute secondary task, if available
if $nt \neq$ NULL then if $nt > TH_{E3}$ then return $nt$

// Execute ZKT
next_task := T.ZKT.pop()

// Execute low-efficiency secondary task, if no ZKT left
if next_task $=$ NULL then next_task := $nt$

return next_task

has more time-consuming kernels than the secondary task, thus reduced performance of the primary task has a strong negative impact on the makespan, more than can be made up for by improved secondary task performance.

After Secondary Task Selection the system resumes executing $pt$ until ‘a reconfigurations’ before its next prefetch. Reconfigurations for $nt$ are then started and the system resumes executing $pt$ until it issues its next prefetch.

The events before and after the prefetch are sketched in Figure 4 for illustration. Task efficiency of the currently running task (sometimes primary, sometimes secondary) is shown with a solid line, while a dashed line is used for the non-running task. Ideally, reconfiguration for the secondary task (see (1) in Figure 4) is finished immediately before the primary task issues its next prefetch. Due to input-data dependent behavior of the primary task, its prefetch may be issued earlier or later than expected (differences between expected prefetch time from offline profiling and actual prefetch time are evaluated in the results) and thus reconfigurations of the secondary task may be finished earlier or later than the prefetch of the primary task. If the reconfigurations are finished earlier, RICL for the primary task before the prefetch will be increased more than expected by Algorithm 2 due to the longer time the primary task runs with less reconfigurable fabric. If reconfigurations for the secondary task are not yet finished at the time the primary task issues its prefetch, its pending reconfigurations are aborted (to start the new reconfigurations for the primary task), and the secondary task executes with reduced efficiency until switching back to the primary task.

Once the primary task issued its prefetch, its efficiency will drop as it requires accelerators that are not yet loaded on the reconfigurable fabric (see (2) in Figure 4). MORP now exits the Primary Task Execution state (see Figure 3). Before Secondary Task Execution can start, the system handles the following special cases: If the primary task issued its Last Prefetch or terminates, then a new primary task needs to be selected and its pending reconfigurations need to be started before proceeding to the Secondary Task Execution state. If the next task $nt$ that was selected during the Primary Task Execution state was a primary task, then the system directly switches to the new primary task, re-

Algorithm 2: Select Secondary Task

**input:**
- taskset $T'$
- currently running task pt
- Performance Threshold $TH_P$

**output:** reallocation amount $a$, secondary task $nt$

$nt := NULL$, $a = 0$, $best = 0$

foreach $i$ in Fabric.size/2 do

if $t \in T.ZKT$ or not $t.issued.prefetch$ then continue

// Performance of primary task if $i$ containers of fabric are taken away
$P = pt.current.kernel.Perf(Fabric.size - i)$

// Performance of secondary task if $i$ containers of fabric are provided
$N = t.current.kernel.Perf(i)$

if $P < TH_P$ or $N < TH_P$ then continue

// Score if $i$ reconfigurable containers are reallocated to task $t$
if $P + N > best$ then

$best = P + N$, $a = i$, $nt = t$

return $(nt,a)$

In the Secondary Task Execution state, the next task is selected using Algorithm 1. If the primary task efficiency is above the threshold $TH_{E2}$, it is considered sufficiently high and the system switches back to the Primary Task Execution state. Otherwise, tasks that have not yet issued their first prefetch (e.g. that are still in their initialization phase) are selected as the next task. We call this preferred selection of not yet initialized tasks ‘task warm up’. The goal of this warm up is to have a prefetch ready for all tasks (where possible), as without a prefetch the system does not know which accelerators are needed for the task and therefore no reallocation can be performed. If all tasks are already warmed up, the secondary task is scheduled, if such a task is available and its efficiency is higher than $TH_{E3}$. Otherwise a ZKT (Software-task) is scheduled, and if none are available, the secondary task is scheduled even if its efficiency is lower than $TH_{E3}$.

The system proceeds with the Secondary Task Execution, hiding the reconfiguration latency of the primary task (see (3) in Figure 4), until the primary task has sufficiently high

Figure 4: Efficiency of primary and secondary tasks in the proximity of a primary task prefetch.
Table 1: Benchmarking Setup

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluated Schedulers</td>
<td>SPT, RR, MORP</td>
</tr>
<tr>
<td>Number of Tasks</td>
<td>3</td>
</tr>
<tr>
<td>Number of Tasks per Taskset</td>
<td>3 – 5</td>
</tr>
<tr>
<td>$TH_{E1}$, $TH_{E2}$, $TH_{E3}$</td>
<td>0.9, 0.8, 0.9</td>
</tr>
<tr>
<td>Total Number of Simulations</td>
<td>135</td>
</tr>
</tbody>
</table>

Table 2: Tasks used during evaluation

<table>
<thead>
<tr>
<th>Taskset</th>
<th>H.264</th>
<th>H.264</th>
<th>H.264</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKT</td>
<td>SUSAN</td>
<td>SUSAN</td>
<td></td>
</tr>
<tr>
<td>SKT</td>
<td>AdPCM</td>
<td>RiJndael</td>
<td></td>
</tr>
<tr>
<td>ZKT</td>
<td>SHA</td>
<td>SHA</td>
<td>SHA</td>
</tr>
</tbody>
</table>

5 Results

Evaluation was done with our cycle-accurate SystemC-based simulator, which models the reconfigurable processor shown in Figure 1 and described in Section 3.1. We envision that such a reconfigurable processor is implemented using an embedded FPGA for the reconfigurable fabric and the core processor is implemented as an ASIC on the same chip, similar to the Xilinx Zynq-7000 (with the difference that the fabric is loosely coupled in the Zynq, while we assume that the fabric is tightly-coupled to the core CPU).

As a proof-of-concept, we have implemented the reconfigurable processor as an FPGA-only prototype, based on a Gaisler SoC, consisting of a LEON 3 [1] SPARC V8 CPU with MMU, DDR Controller and peripherals. The pipeline of the LEON CPU is connected to our runtime-reconfigurable fabric. Special Instructions executing in the fabric can receive input data from the register file, the memory hierarchy, or an on-chip Scratchpad Memory that is connected to the fabric with two 128-bit wide ports (see Figure 1). We used a Xilinx Virtex-5 LX110T based evaluation board for the implementation. Running Linux 2.6.36.4 on the prototype, we have measured the reconfiguration time required for loading one accelerator to be 0.6 – 0.7 ms and context switching time approximately 12.5 $\mu$s. Depending on accelerator complexity, the number of FPGA Slices required to implement one accelerator was between 12 and 147 with a median of 51 slices. The simulator configuration parameters for the benchmarks are summarized in Table 1.

We have evaluated 3 different tasksets, shown in Table 2. The tasksets comprise applications from the MiBench-suite [9] and an H.264 video encoder. The tasks H.264 (video encoding) and SUSAN (image recognition) are multi-kernel tasks, i.e. for each outer loop iteration, multiple distinct prefetches are performed. AdPCM (audio encoding) and RiJndael (cryptography) are single-kernel tasks, i.e. only one prefetch at the start of the program is performed. SHA (cryptographic checksum) is a software task, i.e. no prefetches are performed and thus no reconfigurable fabric is used. Although SHA is a good candidate for hardware-accelerated execution, we chose to use a software-only version to show how our system handles heterogeneous tasksets. Figure 5 shows the single-tasking execution time of the tasks when executed on systems with different fabric sizes. A normalized version of these curves is provided to MORP for making scheduling decisions.

For task scheduling policies we use Shortest Processing Time (SPT, optimal for makespan minimization of non-reconfigurable single-core processor [14]), Round Robin (RR, used in the reconfigurable Molen processor [22]), MORP and Optimal. We define “Optimal” as SPT scheduling with a RiCL of 0 and no overhead apart from context switching time. We can obtain such results from our simulator by configuring it to assume zero-overhead reconfiguration time. However, makespans produced by the “Optimal” scheduler are not achievable in most cases, and should be only regarded as a lower bound for the makespan.

In addition to scheduling, the distribution of the reconfigurable containers, i.e. the share of the reconfigurable fabric assigned to a task at a given point in time, needs to be determined. SPT runs a task to completion, therefore during the time a task runs, all reconfigurable containers are assigned to this task. This policy is not beneficial for RR, as it switches frequently between different tasks and would have to reconfigure potentially all containers after each context-switch (multiple ms additionally; which is an infeasible overhead). Instead, for RR it is beneficial to divide the reconfigurable fabric between the currently executing tasks, i.e. a particular task $T_i$ has a certain share of the reconfigurable fabric where it can reconfigure. Another task $T_j$ has its own share that it can reconfigure, but it can not overwrite the share of $T_i$. RR partitions the fabric equally among the tasks, however, not exceeding the amount of accelerators a task can make use of, e.g. AdPCM achieves its highest performance with 2 reconfigurable containers, while performance of H.264 just saturates at 19 containers. MORP uses its integrated fabric distribution, described in Section 4.

For accurate overhead analysis, the C implementation of MORP was fed with test input data and the execution time...
Taskset 1
Taskset 2
Taskset 3

Reconfigurable Fabric size
[# Reconfigurable Containers]

Figure 6: Makespan of 3 tasksets when scheduled by different schedulers.

Reconfigurable Fabric size
[# Reconfigurable Containers]

Figure 7: Relative speed of MORP, Round Robin and SPT schedulers compared to Optimal.

5.1 Evaluation & Discussion

The makespan results of the benchmarks characterized by Table 1 are shown in Figure 6. Figure 7 shows the results of the benchmark as relative speed of MORP, Round Robin; and SPT when compared to Optimal. Makespans scheduled by MORP are 2.8% slower than Optimal (median; mean: 5.8% slower), whereas makespans of SPT and Round Robin are 13.8% and 19.6% slower than Optimal (median; mean: 12.2% and 19.6%), respectively. Scheduler performance varies with the taskset. MORP achieves its best results with Taskset 3, as does Round Robin, while SPT favors Taskset 2. For each of the 3 tasksets for all sizes of the reconfigurable fabric MORP yields better results than SPT and Round Robin in 35 out of 45 benchmark configurations. In the cases where MORP does not yield the best makespan, its result is on average 1% worse than that of the best scheduler for the particular configuration. Most of the configurations where MORP did not achieve the best result were in Taskset 1, which is disfavorable to MORP, as explained further below. In the remaining configurations in Tasksets 2 and 3 where MORP does not yield the best makespan, RiCL was already very low, thus the RiCL reduction by MORP could not offset the makespan increase due to its overhead.

The following two non-scheduler related effects can be observed: (i) The more reconfigurable fabric is available for accelerators, the faster the taskset is completed – this is due to more kernels being executed in hardware, as more fabric is available. (ii) As the fabric size increases, the makespans of all schedulers converge. The reason is that once sufficient reconfigurable containers are available, a task will have all accelerators for all of its kernels loaded on the fabric and will no longer perform any reconfigurations. Once this fabric size is reached, RiCL is nearly 0 and the scheduler no longer has an effect on the makespan (apart from its overhead).

For very small fabric sizes (1–2 reconfigurable containers) MORP and SPT produce similar results, as reallocating from the already small fabric would slow down the primary task significantly, thus MORP behaves similar to SPT. Round Robin will generally yield a worse makespan for small fabric sizes. If multiple tasks would benefit from hardware acceleration, then RR will force some of them to execute in software as not enough reconfigurable containers are available for all of them. For instance, in Taskset 3 only the H.264 task is hardware accelerated when using RR. On larger fab-
Table 3: Average amount of reconfiguration data transferred.

<table>
<thead>
<tr>
<th>Scheduling Strategy</th>
<th>Transferred Configuration Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round Robin</td>
<td>24,022 kB</td>
</tr>
<tr>
<td>SPT</td>
<td>25,800 kB</td>
</tr>
<tr>
<td>MORP</td>
<td>28,318 kB</td>
</tr>
</tbody>
</table>

Figure 8: Overhead of MORP as share of the total makespan.

The overhead required for MORP (already included in the benchmark measurements) is approximately 1% of the total makespan. Figure 8 shows boxplots for the overhead for each of the 3 tasksets. The runtime of the MORP components mainly depends on the number of tasks in the tasksets and the size of the reconfigurable fabric. Table 3 shows the amount of bytes transferred through the reconfiguration port per simulation run. MORP keeps the reconfiguration port busier than the other schedulers, as it deliberately performs prefetches when from a primary to a secondary task to reduce the total makespan. The increased utilization of the reconfiguration port may have a negative impact on power utilization, however, as the makespan of a taskset is reduced, the energy consumption may still be lower.

5.2 In-Depth Analysis

In this section we focus on one run of Taskset 2 on a fabric with 10 containers. In Figure 9, the top stripes marked “Reconfiguration Trace” show the reconfigurations performed by the system and the areas marked “Scheduling Trace” show the scheduled task. The colors indicate which task recon-
In the direct comparison in Figure 9, MORP reduced the RiCL value down to only 6% of the SPT RiCL value. RiCL in Figure 9b) at 3 is due to the difference between the Average Time Between Prefetches (ATBP) from offline profiling (see Section 3.3) and the actual time between prefetches during the execution of the taskset. ATBP is provided for each kernel, and is only of interest for MKTs (as SKTs only issue one prefetch). For the H.264 MKT in Taskset 2 on a fabric with 10 reconfigurable containers, offline determined ATBP for the first kernel is off by 13%, for the second kernel by 21% and for the third kernel even by 72%. The third kernel has a very short duration, compared to the other two, thus RiCL reduction and execution time variation due to input data have a stronger effect. As evidenced by the overall RiCL reduction and the resulting makespan improvement, MORP is quite robust against these mispredictions in ATBP.

6 Conclusion
In this work we present an approach for reducing makespan on reconfigurable processors. To achieve this goal, our proposed approach, MORP, tackles both scheduling and allocation of the reconfigurable fabric. We identify the Reconfiguration-induced Cycle Loss (RiCL) as the main contributor to makespan on reconfigurable processors. RiCL occurs after a task has issued a reconfiguration prefetch, thus our technique focuses on improving system performance at these particular points during task execution. We reduce RiCL through hiding reconfiguration latency of a primary task by choosing a fitting secondary task and re-allocating fabric between both tasks in a way as to optimize performance of both tasks.

Our MORP scheduler achieves an average makespan reduction by 6.5% and 20.3%, when compared to the SPT scheduler (Shortest Processing Time, optimal for makespan minimization of non-reconfigurable single-core processors) and Round Robin (used in the reconfigurable Molen processor), respectively. Compared to the makespan of an optimal schedule, MORP produces results which are only 2.8% (median; mean: 5.8%) worse than optimal, while other evaluated schedulers produce schedules with makespans that are 14%-20% worse than optimal.

7 Acknowledgments
This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center “Invasive Computing” (SFB/TR 89).