

Relocatable Hardware Threads in Run-Time Reconfigurable Systems

Alexander Wold¹, Andreas Agne², and Jim Torresen¹

¹ Department of Informatics, University of Oslo, Norway

² Computer Engineering Department, University of Paderborn, Germany
{alexawo,jimtoer}@ifi.uio.no, agne@upb.de

Abstract. Run-time reconfiguration provides an opportunity to increase performance, reduce cost and improve energy efficiency in FPGA-based systems. However, run-time reconfigurable systems are more complex to implement than static only systems. This increases time to market, and introduces run-time overhead into the system. Our research aims to raise the abstraction level to develop run-time reconfigurable systems. We present operating system extensions which enable seamless integration of run-time reconfigurable hardware threads into applications. To improve resource utilization, the hardware threads are placed on a fine granularity tile grid. We take advantage of a relocatable module placer targeting modern field programmable gate arrays (FPGAs) to manage the reconfigurable area. The module placer accurately models the FPGA resources to compute feasible placement locations for the hardware threads at run-time. Finally, we evaluate our work by means of a case study that consists of a synthetic application to validate the functionality and performance of the implementation. The results show a reduction in reconfiguration time of up to 42% and more than double resource utilization.

1 Introduction

The design and implementation of field programmable gate array (FPGA)-based systems which use run-time reconfiguration is significantly more complex compared to purely static systems. Run-time reconfigurable systems require careful floorplanning to partition the device into static and reconfigurable regions. In addition, the communication infrastructure that allows for communication between the static and the run-time reconfigurable regions, introduces complexity which increases development time, and introduces run-time overhead into the system. The run-time overhead includes management of reconfigurable resources, reconfiguration time and unused resources due to fragmentation. If the complexity associated with partial run-time reconfiguration is not addressed, the advantages offered may be nullified. It is therefore an attractive proposition to address these challenges in order to allow systems to be implemented at lower cost.

For many years, the implementation of partial run-time reconfigurable systems with many relocatable hardware threads has provided a challenge to engineers. Recently however, improved tools which target partial run-time reconfigurable

systems have become available [1, 2]. These tools have simplified the design of reconfigurable systems. In particular, new features enable implementation of systems which allow partially reconfigurable (PR) modules to be relocated at run-time.

Previous work related to both design time aspects and run-time aspects for PR implement a coarse grained tile grid to place the PR modules [3, 4]. As device size and number of PR modules increase, it will be increasingly difficult to maintain resource utilization on a coarse granularity tile grid used in earlier work. This requires a fine granularity tile grid to place the PR modules. Part of the theoretical foundation presented in these publications still applies to modern devices. However, modern FPGAs are both larger and more heterogeneous than earlier devices.

In this work, we present improvements to both the development framework and the run-time environment. We aim to raise the abstraction level for developing such systems. A higher abstraction level has the potential to shorten development time. This requires both a flexible development framework and an operating system to manage the reconfigurable resources at run-time.

Our work targets ReconOS, an operating system and programming model which supports heterogeneous applications [5]. These heterogeneous applications consist of both hardware and software threads. ReconOS provides a unified programming interface for both software and hardware threads. To raise the abstraction level, we hide implementation details related to scheduling and placement of reconfigurable hardware threads behind the operating system’s application programming interface (API). This allows the operating system to manage, not only static hardware threads, but also hardware threads which are reconfigurable at run-time.

Relocatable hardware threads are modules which can be placed at different locations in the reconfigurable area during run-time. However, a number of constraints must be met to place a hardware thread. The resources required by the hardware thread must be available at the location, the area must be free (i.e. not used by another hardware thread) and communication has to be routed. To compute placement locations where these constraints are met, we have integrated a module placer into the ReconOS operating system.

The module placer implements a realistic constraint model to accurately model the fabric of the FPGA. Computation of feasible placement locations is based on constraint satisfaction theory [6]. This allows us to implement hardware threads with a complex layout. In addition, the module placer supports multiple alternative layouts for a single hardware thread. Multiple layout variants increase the number of feasible placement locations as reported in [6, 7].

In addition, we have implemented a communication infrastructure according to the zero logic overhead concept introduced by Koch et al. [8]. To provide a communication infrastructure between the static and the reconfigurable area, routing of the communication wires is critical. There is no built-in support in the vendor tools to enable fine granularity routing constraints (i.e. mapping of a

signal to a physical wire). Therefore, this has to be undertaken with dedicated tools, for example GOAHEAD [2] and OpenPR [1].

The remainder of the paper is organized as follows: The implementation flow is introduced in the following Section 2. In Section 3, we introduce the run-time environment. This is followed by experimental results in Section 4, and a conclusion in Section 5.

2 Run-time Reconfigurable System Implementation

To implement run-time reconfigurable systems, the design is partitioned into static and dynamically reconfigurable regions. In this work, we have partitioned the device according to the GOAHEAD floorplan flow presented in [2]. GOAHEAD works in conjunction with the Xilinx tools by generating placement and routing constraints. The GOAHEAD flow covers system partitioning and signal-to-wire mapping of wires crossing the boundary of the reconfigurable region. The signal-to-wire mapping is required to implement communication between the regions. In addition, GOAHEAD supports routing of the clock nets. It is a prerequisite to implement identical clock net routing for the static and partial run-time reconfigurable region.

Subsequent to the essential steps of system partitioning, signal-to-wire mapping and clock net routing, the static design can be implemented independently of the hardware threads. Independent implementation of the static design and the hardware threads is a feature supported in both OpenPR [1] and GOAHEAD, however not in PlanAhead according to [2]. This is of particular importance in this work, as we implement many hardware threads in small bounding boxes to reduce fragmentation. This significantly increases the tool time to place and route the hardware thread. It is therefore essential to be able to implement multiple hardware threads concurrently.

OpenPR and GOAHEAD allows design changes to be made to the static design without incurring a reiteration of place and route of the relocatable hardware threads.

2.1 System Partitioning

We aimed to minimize the static region and maximize the run-time reconfigurable areas, since this allows the maximum number of hardware threads to run concurrently. The maximum number of concurrent hardware threads is however restricted to the number of fast simplex link (FSL) ports supported by the MicroBlaze processor. Currently, this is limited to 14 ports.

The static region is not required to have a rectangular shape. For example, it is possible to define reconfigurable areas which are \sqcap and \sqcup shaped in addition to rectangular areas. GOAHEAD allows definition of placement and routing constraints with a polyomino (e.g. \sqcap) shaped layout, and modeling this layout is supported by the module placer.

The size, shape and location of the static region is determined by two factors. Size is determined by the resource requirements. The shape and location

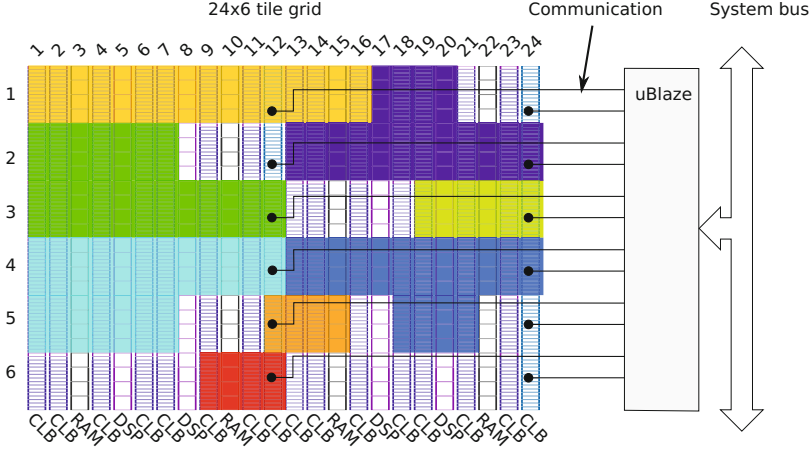


Fig. 1. 8 ReconOS hardware threads placed on a 24×6 tile grid. The placement depicted is computed by the module placer.

is constrained by the IOs to external peripherals. For example, the location of IOs for the PCI-Express interface and memory controller typically reside in the static partition. On the Virtex-6 ML605 evaluation kit, used in this work, the external memory is connected to IOs in the center of the device. In Figure 1, the floorplan of a fully placed and routed design is depicted. To the right of the device depicted in the figure, dedicated interfaces such as high speed serial PCI-Express links are located.

2.2 Relocatable Hardware Threads

We have implemented the run-time reconfigurable area with a fine granularity tile grid, 24×6 . 24×6 tiles means the FPGA is divided into a fine granularity grid of 24×6 tiles. However, communication is implemented on a coarser granularity tile grid, 2×6 . This allows the implemented hardware threads to have little internal fragmentation. However, unused area between the hardware threads decreases resource utilization for the reconfigurable area. To address this, we support hardware threads with polyomino shapes, as depicted in Figure 1. For example, hardware threads can be of \square , \sqcap , \sqcap and \sqcap shape. These hardware threads span more than one configuration frame in height, also depicted in Figure 1.

The layout for a hardware threads are defined in GOAHEAD. GOAHEAD can be used to generate constraints to allow a hardware thread to be implemented with a particular shape. This forces the Xilinx tools to implement the hardware thread within the defined bounding box. As the Xilinx placer does not support an API to guide the routing decisions, the size of the area defined to implement the hardware thread may be too large or too small. A larger area results in internal fragmentation and a larger bitstream. A smaller area leads to unsuccessful

implementation, for example a hardware thread which does not meet timing constraints.

Hardware threads can be made relocatable by modifying the frame address registers (FARs) within the generated bitstreams on the fly during reconfiguration. This is accomplished by storing the addresses where the FARs are located in a list. Since this is done once for each module at design-time, we do not have to parse the entire bitstream at run-time to find the FARs. The FARs are updated to their new values at run-time when the hardware thread position is known.

3 Run-time Environment Implementation

In this section we present our extensions to the ReconOS operating system for the support and management of relocatable hardware threads. The extensions consist of integration of a reconfiguration manager to manage the partial run-time reconfigurable area, and operating system support to schedule hardware threads at run-time. We have encapsulated low level implementation details into a high level thread API. This provides the necessary abstraction level to aid developers to use relocatable hardware threads. The high level API is exposed to the application as a set of system functions to create, suspend, resume and terminate threads.

3.1 ReconOS API Extensions and Scheduling

The ReconOS system function *hwt_create* creates a hardware thread. In Figure 2 API extensions and the life cycle of a hardware thread is depicted. Each hardware thread has a software delegate thread to manage communication with the other threads of the application. Similar to software threads, hardware threads have a thread control block (TCB). The TCB is used by the hardware thread scheduler and contains pointers to the bitstream layout variants, the address of the FSL communication port, and the current scheduling state. The FSL port address is required to allow the hardware thread to communicate with the delegate thread.

The reconfiguration manager is invoked through the hardware thread scheduler to compute a feasible location for the hardware thread's bitstream variants. If a feasible location exists, the FARs in the bitstream are updated and the bitstream is transmitted to the internal configuration access port (ICAP) port of the FPGA. The TCB is updated with the current FSL port address and the scheduling state is set to RUNNING. If a feasible location does not exist for any of the hardware thread layout variants, the scheduling state is set to WAITING. It can then be placed at a later time when the hardware thread constraints are met in the run-time reconfigurable region. We follow the software methodology of adding (hardware) threads to a waiting queue if they cannot be placed. This allows for the creation of more hardware threads than can currently fit into the reconfigurable region.

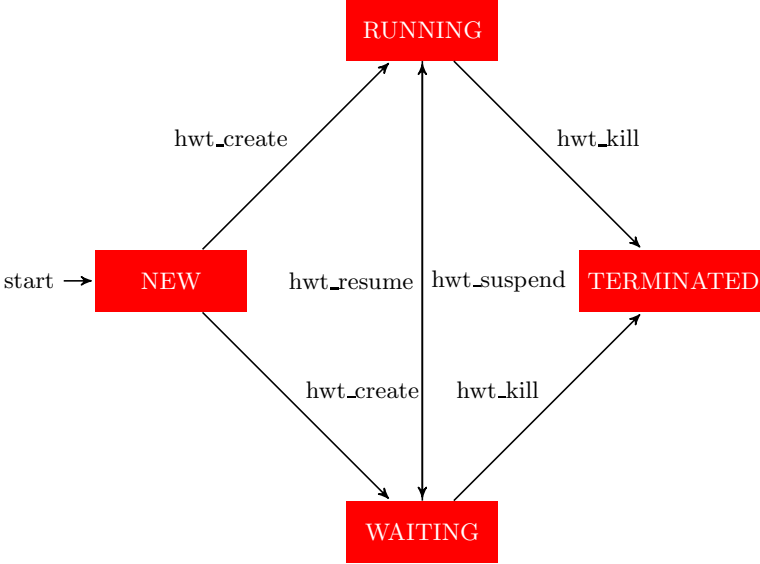


Fig. 2. API extensions to control the life cycle of a hardware thread

Hardware threads can be suspended and resumed through the API functions *hwt_yield* and *hwt_resume*. In this work, we consider a co-operating scheduling strategy of the hardware threads. A co-operative, or run to completion, schedule allows a hardware thread to finish its execution before it is suspended.

When a hardware thread is suspended (*hwt_yield*), the reconfigurable area is marked as unused, and the TCB is updated to reflect the new state. The area can then be used by another hardware thread. If there are threads in the waiting queue, the reconfiguration manager is invoked by the scheduler to compute a feasible placement location for the waiting hardware threads. Hardware threads which have a feasible placement location are placed in the reconfigurable area, and the TCBs are updated with the new FSL port address and state.

Finally, the *hwt_kill* function terminates both, the hardware thread and the delegate thread. The reconfiguration manager is then updated to allow the area to be reused by another hardware thread.

3.2 Reconfiguration Manager

In order to find suitable placements for hardware threads, we integrated a module placer into the ReconOS system. The module placer takes unused area, hardware thread layouts, communication interface and the heterogeneous tile grid into account to compute feasible placement locations. This is accomplished according to the placement model formulated in [6]: A module (hardware thread), M , consist of a sequence of one or more layouts, $M = \{L_1, \dots, L_n\}$. A layout, L is a implementation variant, consisting of a sequence of one or more tile resources,

$L = \{R_1, \dots, R_n\}$. The tile resource sequence, R , is the arrangement of tiles required by the layout variant. There are different types of tile resources, representing physical FPGA resources such as logic and memory. R is defined as a bounding box, $R(x_0, y_0, x_1, y_1, k)$, in which x and y represent the bounding box coordinates. k is a sequence of elements denoting the type of the tile resource column on the FPGA (e.g. a logic or memory column). Similarly, the run-time reconfigurable area is modeled as a sequence of tile resources, $\text{FPGA} = \{R_1, \dots, R_n\}$.

To compute feasible placement locations, the constraint solver evaluates each constraint (i.e. unused area, resources). This results in a sequence for each constraint, \mathcal{C} . The intersection of all sequences form a sequence, \mathcal{P} , of feasible placement locations: $\mathcal{P} = \{\mathcal{C}_{area} \cap \mathcal{C}_{communication} \cap \mathcal{C}_{tilegrid}\}$. \mathcal{P} is computed by our constraint solver using a branch and bound depth first search function.

An up to date scheduling state of all placed hardware threads in the tile grid is kept by the module placer. This state is updated whenever hardware threads are created, suspended, resumed or terminated. When a hardware thread is to be placed, the module placer is invoked by the hardware thread scheduler. The module placer computes a feasible placement location for one of the hardware thread layout alternatives. The computed placement is then returned to the scheduler which updates the bitstream with the new location and writes the updated bitstream to the ICAP port.

4 Experimental Results

To assess our approach, we have implemented the system presented in the previous sections and performed an experimental evaluation. The experiments have been carried out on a Xilinx ML605 Board. Our implementation supports up to 12 active hardware threads located within a 24×6 tile grid (2×6 for communication). The implemented design is depicted in Figure 3.

We have implemented a benchmark application which consists of hardware and software threads. The application creates 20 hardware threads, 5 for each function listed in Table 1. The hardware threads are scheduled and placed by the operating system. This allows us to verify the correct operation of the system when threads are suspended and resumed. In addition, the application allows us to evaluate the effect of multiple layout variants for the hardware threads on a fine granularity tile grid through experiments rather than simulation.

For comparison, we have also done experiments on two coarser tile grids, 1×6 and 2×6 . The height of each tile is a single configuration frame. For the 1×6 tile grid (i.e. slot style), a single tile contains 9920 LUTs (1240 CLBs), 32 BRAMS and 32 DSPs on our XC6VLX-240T FPGA. At this coarse granularity the tiles are still homogeneous. For the 2×6 tile grid, we have a tile size of 4960 LUTs (620 CLBs), 16 BRAMS and 16 DSPs. Note that our communication infrastructure is implemented according to the zero logic overhead concept for all tile grids - a communication infrastructure that uses only routing resources. Therefore, all other reconfigurable resources in the tiles are available to the hardware threads. For the 24×6 tile grid (2×6 tile grid for communication), the

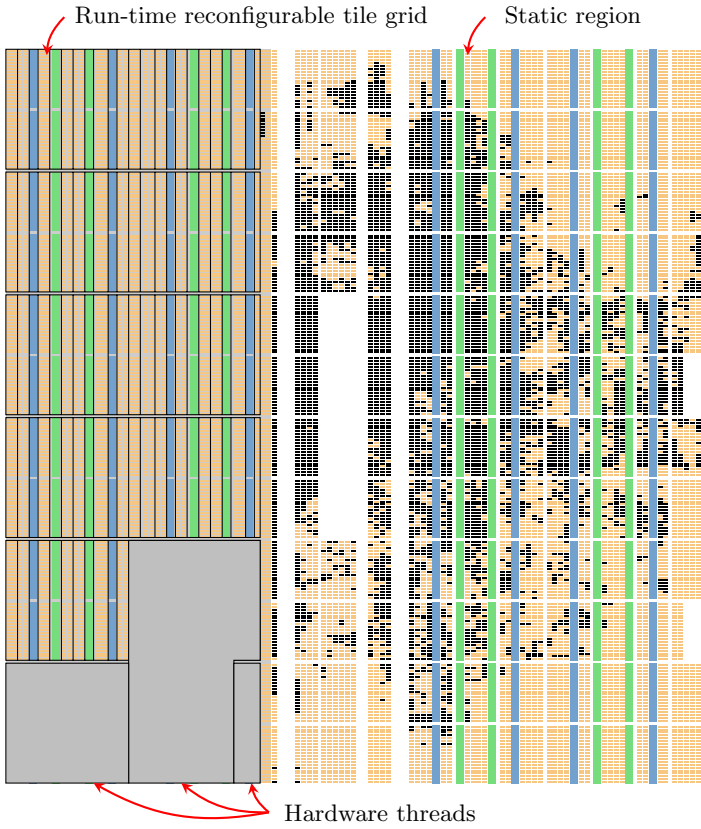


Fig. 3. Implementation of a the 24×6 (2×6 for communication) tile grid on a Virtex-6 (xc6vlx240t) FPGA. The depicted FPGA fabric is generated from the XDL file of a fully placed and routed system.


















tile grid becomes heterogeneous and not every tile contains CLB resources. At this granularity the tile grid contains either 640 LUTs (80 CLBs), 8 BRAMs or 8 DSPs.

4.1 Hardware Thread Implementations

In Table 1, the available hardware thread layout alternatives are listed. The hardware threads were implemented independently of the static system as described in Section 2. The selection of hardware threads include accelerators for computation of square root (SQRT), SHA256 hash function, single precision floating point (FP) addition/subtraction, and fast Fourier transform (FFT).

Resource utilization within the layout bounding box (i.e. the internal fragmentation) is listed as the percentage of the total amount of LUTs used within

Table 1. Relocatable Hardware Threads

Function	Layout	Tile Grid	Tiles	Usage LUTs	LUT-s/REGs	Tool Time	Thread Size
SQRT		1×6	1	11%	1162/1390	21m	342KB
SQRT		2×6	1	23%		22m	206KB
SQRT		24×6	5	62%		2h49m	91KB
SHA256		1×6	1	67%	6664/6971	6h24m	392KB
SHA256		2×6	2	67%		6h24m	392KB
SHA256		24×6	19	80%		16h35m	376KB
SHA256		24×6				16h34m	376KB
SHA256		24×6				16h34m	376KB
SHA256		24×6				16h35m	376KB
SHA256		24×6				16h36m	376KB
FP		1×6	1	4%	410/629	22m	209KB
FP		2×6	1	8%		25m	111KB
FP		24×6	4	21%		26m	59KB
FP		24×6	5	21%		21m	73KB
FFT		1×6	1	21%	1996/2737	22m	384KB
FFT		2×6	1	40%		26m	246KB
FFT		24×6	8	62%		6h40m	172KB

the bounding box. A higher utilization is not always possible due to the resource demands of the hardware thread, the granularity of tile grid and routing constraints. For example, hardware threads such as the square root have a high amount of unused resources when implemented on a coarse granularity tile grid. If the hardware thread does not fit on a single tile, two or more tiles are used. For example, the SHA256 hardware thread does not fit in a single 2×6 tile, and therefore requires two 2×6 tiles. The number of tiles listed for the 24×6 grid include BRAM and DSP tiles.

In our work, we have considered layout variants, but not design variants of the hardware threads for the different tile grids. In many cases it is possible to exploit the module design space to better utilize tile resources. For example, various levels of transformations (e.g. loop unrolling and pipelining) can be applied to the SHA256 hashing algorithm to obtain design variants with different resource requirements. In Figure 4, placed and routed layout variants of a functionally equivalent hardware thread is depicted.

We observe that a smaller layout bounding box increase the tool time to implement the hardware thread. In particular the place and route time increases. The tool times as measured on a Xeon X5690 server are listed for each hardware thread, together with the size of the generated bitstream. We also find that a smaller layout bounding box reduce run-time reconfiguration time. This is to be expected, as the resulting bitstream has fewer configuration frames and thus has a reduced size.

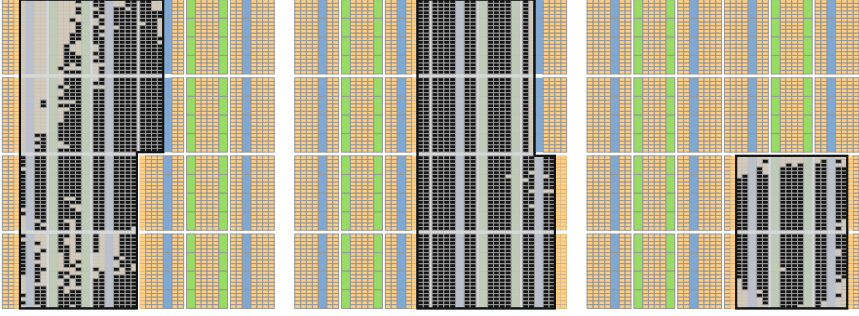


Fig. 4. A hardware thread implemented with multiple layout variants

Table 2. Bitstream Data for all Hardware Threads on Different Tile Grids

Tile Grid	Bitstreams Size	Bitstreams	Average LUT Usage
1×6	6635KB	20	26
2×6	4775KB	20	35
24×6	3855KB	55	56

4.2 Run-time Experiments

In order to evaluate the placement, the application creates 20 hardware threads. Each hardware thread is executed 5 times. After each execution the thread yields, and is removed from the reconfigurable area. If the hardware thread has design alternatives (i.e. the SHA256 and FP), each design alternative is evaluated by the module placer to find one that is feasible.

In Table 2, the hardware thread bitstream data is listed for tile grids of 1×6, 2×6 and 24×6 (2×6 for communication) granularity. On a 1×6 tile grid, a maximum of 6 hardware threads can be executed concurrently. On a 2×6 and 24×6 tile grid, up to 12 hardware threads can be executed at the same time.

A fine granularity tile grid allows significant reduction in the total bitstream size as shown in Table 2. The results show a reduction in bitstream size of up to 42% between a 1×6 and a 24×6 tile grid. Besides reduced storage requirements, the main benefit of this is the decrease in configuration time, which is proportional to the bitstream size. In Table 3, the total bitstream data transferred during execution of the application is listed. On a 24×6 tile grid we have reduced the reconfiguration time by up to 46% compared to a 1×6 tile grid. Using external SPI flash memory at 100MHz, this translates into an improvement from 2.6 seconds (33175KB) down to 1.4 seconds (17800KB).

In addition, we have measured the effect of layout alternatives on resource utilization. While hardware thread layout alternatives consume additional memory

Table 3. Run Time Experiment Results

Tile Grid	Bitstream Data Transferred
1×6	33175KB
2×6	23850KB
24×6	17800KB

for storage, FPGA resources are significantly more expensive than non-volatile memory¹. It is therefore beneficial to use hardware thread layout alternatives to increase resource utilization, even for hardware thread alternatives which are rarely used.

Through our experiments, we found that even with a coarse granularity 2×6 communication tile grid, it is possible to achieve improved resource utilization when combined with multiple hardware thread layout alternatives and a fine granularity resource tile grid (24×6).

5 Conclusion

In this work, we have presented our research on integrating support for relocatable hardware threads in the ReconOS operating system. Our aim has been to raise the abstraction level to develop run-time reconfigurable systems. We have achieved this with extensions to ReconOS which encapsulate implementation details into a high level thread API. This allows seamless integration of run-time reconfigurable hardware threads into applications.

To manage the reconfigurable area, we have implemented a module placer targeting modern FPGAs. The module placer accurately models the reconfigurable resources. This is utilized by the operating system to compute feasible placement locations for the hardware threads. The computed placement positions are then used to relocate the hardware threads by updating the frame addresses in the respective bitstreams.

The system floorplan has been created with GOAHEAD, which enables the implementation of a zero logic overhead communication infrastructure. We have used the tool-flow to develop, synthesize, place, and route the hardware threads independently of the static system as well as other hardware threads. Thus, our improvements allow hardware threads to be implemented independently of each other, similar to the threads of a pure software application.

Our experiments were performed on a Virtex-6 device with a implementation that supports a fine granularity 24×6 tile grid and multiple hardware thread layout variants. This combination enables an efficient use of reconfigurable resources at the cost of additional non-volatile memory to store layout variants.

¹ The price of a Virtex-6 (xc6vlx195t) FPGA is 2210 Euro, and the price of non-volatile memory (16GB compact flash), 214 Euro. The prices have been obtained from <http://de.rs-online.com/web/>.

Acknowledgment. This work is funded by THE RESEARCH COUNCIL OF NORWAY as part of the Context Switching Reconfigurable Hardware for Communication Systems (COSRECOS) [9] project, under grant 191156V30, and by the European Union Seventh Framework Programme under grant 257906, as part of the Engineering Proprioception in Computing Systems (EPiCS) [10] project.

References

1. Sohanguhpurwala, A.A., Athanas, P., Frangieh, T., Wood, A.: OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, Number Xdl, pp. 228–235. IEEE (May 2011)
2. Beckhoff, C., Koch, D., Torresen, J.: Go Ahead: A Partial Reconfiguration Framework. In: 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, pp. 37–44. IEEE (April 2012)
3. Jara-Berrocal, A., Gordon-Ross, A.: An integrated development toolset and implementation methodology for partially reconfigurable system-on-chips. In: ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors, pp. 219–222. IEEE (September 2011)
4. Wang, Y., Zhou, X., Wang, L., Yan, J., Luk, W.: SPREAD: A Streaming-Based Partially Reconfigurable Architecture and Programming Model. IEEE Transactions on Very Large Scale Integration (VLSI) Systems (99), 1–14 (2013)
5. Lubbers, E., Platzner, M.: ReconOS: An RTOS Supporting Hard- and Software Threads. In: 2007 International Conference on Field Programmable Logic and Applications, pp. 441–446. IEEE (August 2007)
6. Wold, A., Koch, D., Torresen, J.: Enhancing Resource Utilization with Design Alternatives in Runtime Reconfigurable Systems. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, Anchorage, pp. 264–270. IEEE (May 2011)
7. Koester, M., Luk, W., Hagemeyer, J., Pormann, M., Ruckert, U.: Design Optimizations for Tiled Partially Reconfigurable Systems. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 19(6), 1048–1061 (2011)
8. Koch, D., Beckhoff, C., Torresen, J.: Zero logic overhead integration of partially reconfigurable modules. In: Proceedings of the 23rd Symposium on Integrated Circuits and System Design - SBCCI 2010, p. 103. ACM Press, New York (2010)
9. COSRECOS: Context switching reconfigurable hardware for communication systems (cosrecos), <http://www.mn.uio.no/ifi/english/research/projects/cosrecos>
10. EPiCS: Engineering proprioception in computing systems (epics), <http://www.epics-project.eu>