An Efficient FPGA Overlay for Portable Custom Instruction Set Extensions

Dirk Koch[†,‡], Christian Beckhoff[†], and Guy G. F. Lemieux[‡]
[†]Dept. of Informatics, University of Oslo, Norway, [‡]Dept. of ECE, UBC Vancouver, BC, CA
Email: koch@ifi.uio.no, christian@recobus.de, lemieux@ece.ubc.ca

***Abstract*—Custom instruction set extensions can substantially boost performance of reconfigurable softcore CPUs. While this approach is commonly tailored to one specific FPGA system, we are presenting a fine-grained FPGA-like overlay architecture which can be implemented in the user logic of various FPGA families from different vendors. This allows the execution of a portable application consisting of a program binary and an overlay configuration in a completely heterogeneous environment.**

Furthermore, we are presenting different optimizations for dramatically reducing the implementation cost of the proposed overlay architecture. In particular, this includes the mapping of the overlay interconnection network directly into the switch fabric of the hosting FPGA. Our case study demonstrates an overhead reduction of an order of magnitude as compared to related approaches.

I. Introduction

While portability of software binaries is essential in many systems, there is a lack to transfer the same approach to reconfiguration bitstreams running on an FPGA. In the software world, portability is often achieved by using binary-compatible CPUs inside the different systems. Unfortunately for FPGAs, there is no direct path to use identical configuration bitstreams on FPGAs from different vendors. However, some related work proposed the implementation of an *overlay* (which is sometimes called *virtual FPGA* or *intermediate fabric*) that basically implements a reconfigurable architecture within the user logic of an FPGA. This can be compared with the Java virtual machine concept, where *bytecode* is shipped to users that can run on completely different CPUs. This is possible by providing a virtual machine implementation for each different CPU that can execute the same Java bytecode. With respect to FPGAs, this can be seen as an FPGA-on-FPGA implementation. And while the host FPGA bitstream is device specific, the configuration data for the overlay can be portable among different physical overlay implementations.

In [1], this concept was presented under the term *Virtual FPGA*. The proposed island-style architecture provides configurable logic blocks consisting of four 3-input LUTs and an interconnection network. While that architecture can be compiled for different targets, the implementation cost for a configurable logic block was reported to be 354 4-input LUTs on a Spartan-II FPGA which results in a physical LUT : virtual LUT ratio (PV-ratio) of more than 100x in practice. By implementing virtual LUTs directly into physical LUTs and by replacing routing multiplexers with LUTs in route through mode (see also Section III-B), a PV-ratio of 40x was achieved in [2]. As another way to reduce implementation cost, the authors of [3] restricted the routing freedom of the overlay interconnection network. For a prototype implementation of a linear systolic fine-grained overlay architecture, a PV-ratio of 16:1 was reported. While this seems better than the two other approaches, this architecture is much more restricted, which in turn, limits the target applications to very simple streaming operations.

As in particular the interconnection network of fine-grained overlays is very costly to implement, coarse grained overlay alternatives have been proposed. Here, the ratio of logic spent in the functional blocks as compared to the interconnection network is much larger. While many coarse grained architectures have been prototyped on FPGAs, only little work is targeting portability among different FPGA platforms. In [4], a domain-specific but portable course-grained overlay architecture was presented where the overhead for the interconnection network was 1/3 of the used FPGA.

A. Towards Portable Custom Instruction Set Extensions

In this paper, we investigate how custom instruction set extensions (CI extensions) of softcore CPUs can be made portable for various target FPGA families with the help of a fine-grained overlay approach. CI extensions have been presented several times before in related publications, like for example, in [5], [6], [7], [8]. It is common for CIs that little extra logic can save tens or even a hundred of machine instructions, hence, potentially resulting in substantial speed-ups. For example, adding a permute instruction that bit-swaps an input operand (such that the MSB becomes the LSB and so forth) needs only very little extra logic in the instruction decoder; while the instruction itself results basically in wiring, when implemented on an FPGA. However, even considering loop unrolling and inlining, this permute instruction would save about a hundred machine instructions on a typical RISC CPU. In addition, code will become more compact which can be another source of performance improvement.

While custom instructions were implemented as reconfigurable instructions before for dynamically adapting the ISA architecture of a CPU at run-time, no work was done so far to utilize *portable reconfigurable* CI extensions.

B. Paper Contribution

The main contributions of this paper are multilateral and separated in the following Sections. In Section II, we propose concepts for portable CI extensions. This is achieved by an overlay architecture that will be discussed in Section III. Next,

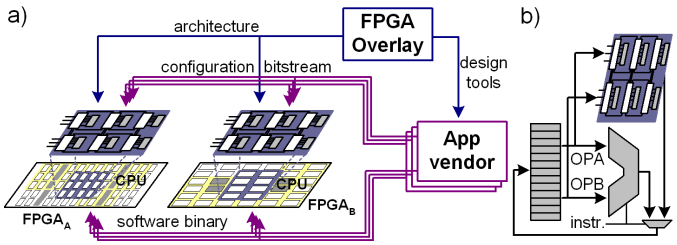


Fig. 1. a) FPGA specific overlay implementations allow porting of custom instruction bitstreams. b) Attaching a custom instruction overlay to a CPU

in Section IV, we present how such an overlay architecture can be implemented an order of magnitude more efficient than previous fine-grained approaches. We achieve this by mapping the interconnection network directly into the switch matrices of the target FPGA. After this, in Section V, we present a case study where we extend a MIPS softcore CPU with CI extensions that can be easily ported to different target platforms. Finally, we conclude this paper in Section VI.

II. Concepts for Portable CI Extensions

As described in the introduction, we achieve portability of custom instruction bitstreams by introducing an overlay architecture that will be implemented for each target FPGA as shown in Figure 1a). While portability of software binaries is achieved by implementing the same softcore instruction set architecture (ISA), our FPGA overlay specification is similarly acting as kind of an API for the CI extensions.

In this scenario, an application vendor can generate and distribute software and configuration bitstreams that can directly run on different target FPGAs. Note that this approach does not need any further target-specific synthesis or place & route steps. However, we assume a wide spectrum of possible implementation and optimization techniques. This starts from an easy portable (but more costly) RTL description of the overlay and ends with an implementation where overlay primitives are mapped directly to the entire target FPGA and where the overlay interconnection network is directly implemented inside the switch fabric of the target FPGA, as described in Section IV. Depending on the optimization level, bitstreams have to be translated separately for individual systems. Luckily, this bitstream generation is simple enough to be performed by the target FPGA itself (e.g., once when a new application is installed). For instance, if we map a virtual 4-input LUT directly to a physical 6-input LUT on the target FPGA, it is sufficient to place the 16 LUT configuration bits from the overlay configuration to the right 64 positions inside the target FPGA configuration. By allowing this process with the help of partial reconfiguration, custom instructions can be dynamically changed with the software application. The required bitstream remapping information can be derived during the implementation of the overlay architecture and this mapping is completely transparent to the application vendor. The mapping information can then be used to implement a driver which generates the configuration bitstreams for a specific target FPGA.

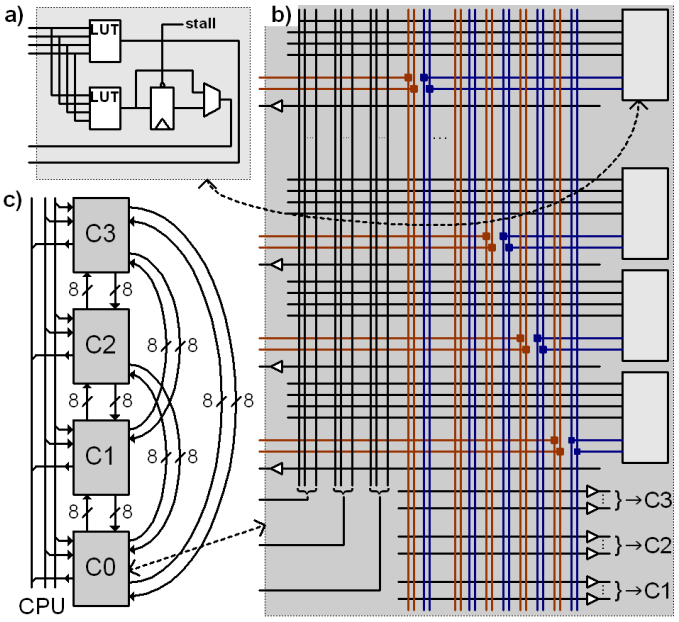


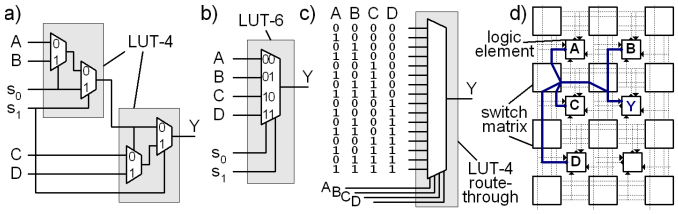
Fig. 2. a) Logic cell consisting of a pair of 4-input LUTs b) Logic cluster consisting of 8 LUT pairs, c) Overlay architecture consisting of four clusters.

As fine-grained overlay implementations introduce a logic overhead and a performance penalty, we propose to design the overlay as small as possible. In the case of instruction set extensions for softcore CPUs, this means that we synthesize an RTL specification of a CPU individually for each target FPGA. This permits high quality implementations of the CPU. To this CPU, we attach the actual overlay architecture that is intended to host relatively small CI extensions. As shown in Figure 1b), we couple this overlay directly to the CPU without additional I/O cells.

A custom instruction is executed by sending a special (not otherwise used) instruction to the CPU. Such unencoded instructions are available in most 32-bit softcore CPUs, like for example, in the MIPS instruction set that we use for the case study. There, we are actually using two unused instructions, one for accessing two registers and one for accessing a register and an immediate value which is stored in the instruction word. The instruction decoder of the softcore CPU can be configured by the CPU itself by writing to a memory mapped register. This allows us to set an extra wait state in the case that the currently configured custom instruction is too slow. This basically implements a controllable multi-cycle operation and permits to run the CPU always at its maximum speed.

III. An Overlay Architecture for Custom Instruction Set Extensions

The primary design objectives of an overlay architecture is to deliver good performance at low implementation cost for many different target FPGAs. However, the implementation cost depends on the implementation effort required to optimize the overlay for a specific target which in turn is limited to the tool capabilities and device documentation (e.g. the availability of a device architecture graph). Moreover, the overlay should be generic for allowing the implementation of a wide range of custom instructions.



	a) cascaded LUT-4	b) single LUT-6	c) route-through	d) routing fabric
Logic cost	high	high	medium	low
Latency	high	medium	medium	low
Config. time	low	low	medium	high
Impl. effort	low	low	medium	high
Conf. storage	extra	extra	bitstream	bitstream

Fig. 3. Pros and cons of different overlay multiplexer implementations.

Our fine-grained overlay architecture is consisting of a pair of 4-input LUTs, as shown in Figure 2a). We grouped 8 LUTs to a cluster with direct LUT-to-LUT routing within a cluster (Figure 2a)). We connected one byte of each CPU operand and one byte of the result vector with each cluster. Clusters can be grouped together as exemplarily shown in Figure 2c). We will discuss our design decisions in more detail in the following sections.

A. Overlay Logic Resources

When implementing an overlay LUT completely in logic, it takes for an n-input LUT 2^n flip-flops for storing the LUT table and one 2^n input multiplexer. To keep logic overhead low, we decided for 4-input LUTs, as they provide good mapping efficiency and as this allows the connection of two input bits from each operand input. Furthermore, many smaller FPGA vendors still use smaller LUTs, which is relevant, when mapping virtual LUTs directly to physical LUTs.

For implementing arithmetic functions, we decided to provide two LUTs that share the same inputs (as input multiplexers are costly). With this architecture, we can generate a sum and a carry bit or we can count the number of bits in a 4-input vector in a single logic cell. Alternatively, if one of the LUTs remains unused as a logic function generator the other LUT can still be used for routing. Note that we are not considering carry logic that is available in most FPGA architectures. This decision was made to favor portability.

We added one user flip-flop to our logic cell which can be used as an accumulator register, for pipelining, or for implementing instructions with more than two input operands.

B. Overlay Routing Network

It is ironic that despite that the basic logic building blocks of FPGAs are actually multiplexers, FPGAs are relatively weak in implementing multiplexers as the user logic. This holds in particular for multiplexers with many inputs that are common for switch matrices in commercial FPGAs. Consequently, an efficient overlay should focus on a highly efficient interconnection network while the logic utilization of the overlay is a minor concern. This is known from plain FPGA design [9] and holds even more for fine-grained overlays.

TABLE I
CLUSTER SWITCH BOX BREAKDOWN. THE VALUES IN BRACKETS [] DENOTE THE NUMBER OF PROGRAMMABLE INPUTS.

switch box inputs		switch box outputs		Mux size
operands A and B	2×8	result outputs	8	4 [32]
logic cell outputs	2×8	logic cell inputs	4×8	7 [224]
from other clusters	3×8	to other clusters	3×8	4 [96]
sum	56	sum	64	[352]

As multiplexers can be logic costly, we discuss pros and cons of different implementation alternatives in Figure 3. As can be seen, four input multiplexers are a good choice to be implemented efficiently for many different target FPGAs, regardless if they are based on 4-input LUTs or 6-input LUTs. For building larger multiplexers, multiple 4-inputs LUTs can be cascaded. For example, two 4-input multiplexers can implement one 7-input one. When targeting any newer Xilinx FPGA, which are all based on fracturable 6-input LUTs, it is possible to implement either one 7-input multiplexer or two 4 input multiplexers (whereof two inputs are shared) in a single 6-input LUT, when using the available extra input to the carry chain logic together with route through mode.

C. Switchbox Design

As switch matrix multiplexers for implementing the overlay routing can be costly, we chose a relatively sparsely connected crossbar inside each logic cluster. Figure 2b) illustrates the crossbar and a breakdown of the inputs and outputs of a cluster switch matrix is presented in Table I. Our cluster switch box consists of in total 56 inputs and 64 outputs. From these $56 \times 64 = 3584$ possible connections, 352 (9.8%) connections can be programmed. As a reference, in a Xilinx Virtex-II (Virtex-5) CLB switch matrix there exist 332 (305) inputs and 160 (172) outputs whereof 6.3% (7.5%) of the connections can be programmed [10].

The Xilinx switch matrices are significantly larger, preliminary because there is more cluster-to-cluster routing to be implemented than in our small custom instruction example. However, for scaling the custom instruction set example up to larger FPGA overlays (with many more clusters), using an extra switching layer [11], LUT input swapping, and LUT route-through routing can be used to keep implementation cost reasonable low.

IV. MAPPING OVERLAY INTERCONNECTION NETWORKS INTO FPGA FABRICS

Figure 3 points out that implementing a reconfigurable overlay routing architecture directly within the switch matrices of the target FPGA will substantially reduce the overhead for the reconfigurable overlay interconnection network. Moreover, this approach can potentially result in better timing performance because routing does not need to pass extra look-up tables and because the saved logic allows for placing the overlay logic closer together which in turn results in faster communication. However, these advantages come at the cost of a longer reconfiguration time needed to load a new instruction to the device. Consequently, direct switch matrix mapping is more suitable for applications that infrequently change custom instructions.

A. Problem Definition

In general, the problems to be solved for implementing overlay interconnection networks directly into FPGA routing fabrics are 1) to place the mapped overlay logic on the target FPGA fabric; and 2) to bind overlay routing resources to physical routing wires of the host FPGA fabric. As compared to a traditional netlist implemented on an FPGA, *the mapping graph of the overlay interconnection network has not one, but multiple sources for each input connection.* In this work, the first problem is accomplished by the vendor tools including the placer which might be guided with additional area constraints. In addition, we can add temporarily connections between the primitives to force the placer to locate physical primitives closely together as given by the architecture graph of the overlay.

The second problem is to reserve the fabric wires which implement the architecture graph of the overlay. In the physical mapping phase, the outputs of overlay primitives result in outputs of the target FPGA primitives (here LUT outputs). Regardless if an overlay LUT is implemented using several physical LUTs, only one output of these physical LUTs will represent the corresponding overlay LUT output. Opposed to this, a single overlay LUT input might be connected to multiple physical LUTs of the target FPGA. For example, when implementing the logic cell from Figure 2a) in two separate LUTs. However, if multiple physical inputs represent an identical virtual input depends on the mapping tool and the target FPGA architecture. For example, when targeting recent FPGAs from the vendor Xilinx, fracturable input LUTs (as shown in Figure 2a) can be directly implemented in a single physical 6-input LUT.

We can express the mapped overlay architecture graph by a bipartite graph with two set of nodes representing 1) the physical primitive output pins and 2) the physical input pins of the mapped overlay architecture. The edges of the graph denote programmable connections available in the overlay network. This is illustrated in Figure 4. Here, an overlay – shown in Figure 4a) – is represented by its bipartite mapping graph as depicted in Figure 4b). The node names in Figure 4b) denote a combination of the coordinate and port of the respective mapped physical primitive on the target FPGA.

Again, as compared to a traditional netlist implemented on an FPGA, the mapping graph has not one, but multiple sources for a single input connection. Each edge to an input node represents a multiplexer input of the overlay interconnection network that we want to implement directly into the host FPGA fabric. Figure 4c) shows a possible routing wire reservation for the example overlay network. Note that this is not a valid configuration because some switch matrix inputs are connected to multiple drivers. As FPGA switch matrix multiplexers are commonly implemented using pass transistors rather than using logic gates, at maximum only one input is allowed to be selected at a point in time. Otherwise, short circuit situations may occur. Note that the bitstream encoding of many FPGAs, including Xilinx devices do allow the selection of multiple inputs simultaneously [12]. This

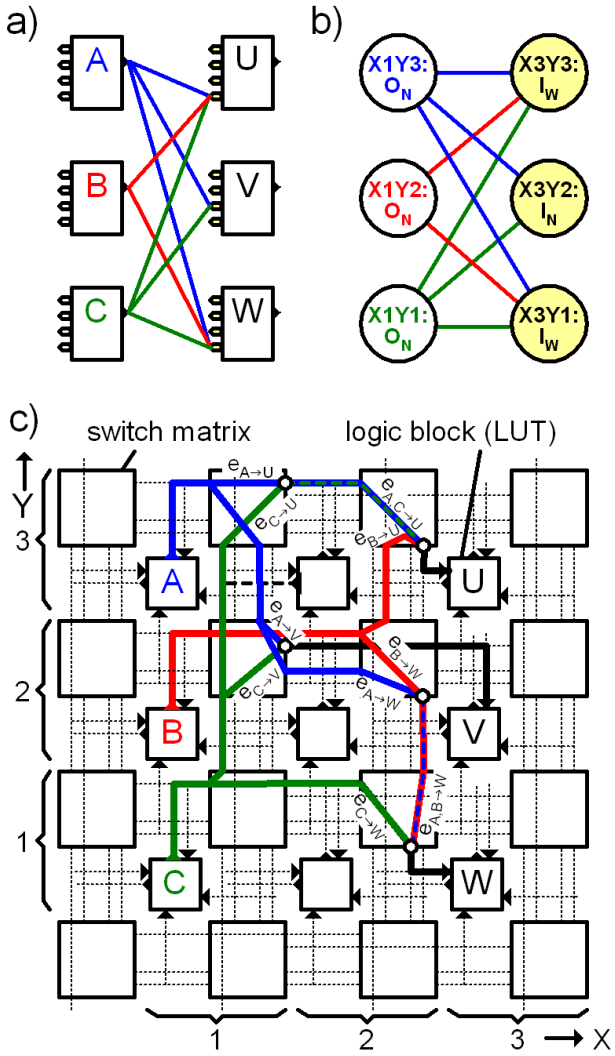


Fig. 4. a) simplified overlay architecture with six LUTs and some programmable connections; b) Bipartite mapping graph. Nodes on the left (right) hand side represent physical LUT outputs (inputs) of the mapped overlay on the target FPGA, which is shown together with the reserved routing in c).

might cause unwanted side effects or even device degradation or damages. While this situation is prevented when using the FPGA vendor tools, precautions are necessary to ensure save operation when mapping overlay interconnection networks directly onto an FPGA fabric.

Unlike traditional FPGA routing, the routing problem is not to build up individual spanning trees for each used primitive output to a set of connected primitive inputs. When reserving routing resources for the overlay routing architecture, *multiple trees have to be considered together, because multiple primitive outputs have to be connected to a single input over time* (by using partial reconfiguration).[1]

[1]This problem is related to [13], where the authors generate a reservation to connect circuit signals to trace buffers for debug purpose. When changing the set of signals to be traced, a new path is chosen from the reserved wires. However, our problem is more difficult as we have not to route to *any* primitive (trace buffer) but to *a specific* primitive which is selected during the placement of the overlay logic.

B. Customizing an Overlay Interconnection Network

As shown in the mapping graph in Figure 4b), we can identify *source graphs (or source trees) with one output primitive pin connected to multiple sinks* and *sink graphs (or sink trees) which run in opposite direction and that have one input primitive pin driven by multiple sources*. For example, there is a source tree from node X1Y3:O_N (LUT A) to the nodes X3Y3:I_W, X3Y2:I_N, and X3Y1:I_W (LUT U, V, and W). An example for a sink tree is node X3Y2:I_N (LUT V) connected to nodes X1Y3:O_N and X1Y1:O_N (LUT A and LUT C).

If we study the reserved routing example in Figure 4c), we see that implemented source and sink graphs hit in the routing fabric of the host FPGA, when paths branch. For instance, in the switch matrix X1Y3, we have a branch of the source graph, from LUT A which hits the sink graph from LUT U. Here, the shown edge $e_{A,C\rightarrow U}$ belongs to both graphs.

A branch of a source graph is identical to a branch occurring in traditional FPGA routing when connecting to multiple primitive inputs. However, branches of sink graphs denote switch matrix multiplexer settings required to be modified for routing overlay network connections. In other words, for deriving an initial configuration from the reserved routing in Figure 4c), we remove all edges, where a sink graph branches. Then, for setting connections in the overlay network, we selectively add the missing edges back to the initial configuration that implement the chosen edge of the mapping graph. By selecting only one wire exclusively at a branch, no short circuit situation can occur. In the figure, we assigned names to exactly all sink tree branches that are set or removed to configure connections from the LUTs A, B, C to the LUTs U, V, W. Note that in some cases more than one switch matrix connection has to be set for implementing a specific overlay routing path. For instance, for configuring a connection from LUT B to LUT U, we have to add the edge $e_{B\rightarrow U}$, while for setting the connection LUT A to LUT U, we have to add edge $e_{A\rightarrow U}$ and edge $e_{A,C\rightarrow U}$.

C. Creating an Overlay Interconnection Network

A straightforward method to create the overlay interconnection network is sketched in Algorithm 1. In this algorithm, we compute one link of the overlay architecture after the other. The links are given by the bipartite mapping graph which also includes the exact position of the corresponding begin and end ports on the target FPGA. Each computed path is added to the interconnection reservation graph on the target FPGA and the used wire resources are then removed from the target architecture graph for the next iteration. Consequently, individual paths will be implemented for each edge in the mapping graph on the target FPGA without sharing routing wires among multiple paths. This algorithm is easy to implement, but it allocates an unlikely large number of wires, which in turn results in a very congested or even uncomplete implementation. It should be understood that the implementation of an overlay interconnection network puts high pressure on the router as we have to consider all reconfigurable alternatives provided in the overlay architecture. In cases with high congestion, it could consequently be beneficial to combine look-up table route-through routing with the here presented direct implementation into the FPGA interconnection fabric.

Algorithm 1: Compute_Overlay_Network_Baseline

```
1  Input  : mapping graph M, target architecture graph T
2  Output : interconnection reservation graph I
3  I = ∅
4  while M ≠ ∅ do
5  {
6     m = FetchNextEdgeFromMappingGraph(M)
7     i = FindPathOnTargetFPGA(T, m)
8     I = I ∪ i
9     RemoveFromArchitectureGraph(T, i)
10 }
```

For finding better reservation graphs with less routing congestion, we developed Algorithm 2. At start, we implement a spanning tree for the first source tree of the mapping graph M (line 6). We then annotate to each used wire in the target FPGA architecture graph its usage for routing to a specific destination (line 8). Let us consider Figure 4 for an example. If the first source tree is the one starting from LUT A, then we annotate to the wire between the switch matrices X1Y3 and X2Y3 the usage {A→U}. For the wire between the matrices X1Y3 and X1Y2, we annotate the usage {A→V, A→W} because this wire implements a path to two destinations from LUT A.

After this, we run for all remaining edges a breadth-first search one after the other. In each search step, we check if we reached the destination or another path. If another paths has the present destination annotated (→destination), we incorporate the remaining other path and update the usage to all wires used for the present path (line 19 and line 20). This allows us to share wires on the path towards an input port.

In order to reuse wires from the output port side, we start the breadth-first search not only from the source output primitive pin, but in parallel from all routing resources that are already connected from that output (if existent). However, we have to restrict this to routing resources that have only one annotated source usage (source→). If there is more than one annotated source usage, this expresses that the routing resource is already shared on a path towards an input port. We identify such wires by traversing already existing paths from the source primitive towards all leaves. We include all routing resources into the overlay path search that have the source usage of the source port and we stop traversing of the already existing path, if we find another source usage. Note that this happens when an implemented sink tree branches. The branches are actually the edges used for customizing (configuring) the overlay, as described in Section IV-B.

V. Case Study

We implemented a case study consisting of a baseline MIPS CPU (no division, no interrupt, no floating point, no pipelining) attached to an overlay which follows exactly the

Algorithm 2: Compute_Overlay_Network

```
Input : mapping graph M, target architecture graph T
Output: interconnection reservation graph I
I = ∅
m = FetchFirstSourceTree(M)
i = ComputeSpanningTree(T, m)
I = I ∪ i
AnnotatePathUsageToEachWire(T, i)
while M ≠ ∅ do
{
    m = FetchNextEdgeFromMappingGraph(M)
    while state(m) ≠ routed do
    {
        i = NextBreadthFirstSearchStep(T, m)
        if ReachedDestination(i,m)
            state(m) = routed
        if HitAnotherRoutingPath(i,T)
            if PathUsage(i, T)=OnlyMyDestination(M)
                GetRemainingPath(T, i)
                UpdatePathUsage(T, i)
                state(m) = routed
    }
    I = I ∪ i     // add new path to result
    RemoveFromArchitectureGraph(T, i)
}
```

architecture shown in Figure 2. Each LUT-input consists of a 7-input multiplexer which allows each 4-input LUT pair to be connected to a set of $4 \times 7 = 28$ different signals within a cluster. For the 8 result and the 3×8 interconnection signals of a cluster, we respectively used 4-input multiplexers. We chose a relatively sparse interconnection network to keep implementation cost down. However, applying optimizations including LUT swapping inside a cluster or LUT input swapping gives significant extra freedom to implement custom instructions on this overlay. The total set of sources within a cluster consists of 2×8 operand signal wires, 2×8 LUT output wires and 3×8 wires from other clusters. This results in 56 cluster inputs (see also Table I). While this is a relatively tiny overlay, it allows us already to implement various custom instructions, for example (fractional) addition, a 64-input XOR gate over all input operand bits, or a bit permute. More instructions haven't been implemented as the place & route is currently performed manually. In this case study, we will focus on the overlay implementation itself.

A. Overlay Implementation

We examined three implementation variants: 1) a fully generic implementation which is entirely running in the user logic of the target FPGA, 2) an implementation using direct mapping of overlay LUTs to physical LUTs (and that uses route-through LUTs for the interconnection network), and 3) direct LUT mapping combined with directly using the switch matrices for the overlay network. The first variant follows the approach presented in [1], the second case [2], while the third examined variant implements our new method of mapping overlays directly to FPGA fabrics. Note that all these implementations have different pros and cons with respect to area, performance and reconfiguration cost (see also Figure 3). However, the configuration can be derived from the same overlay bitstream.

TABLE II
IMPLEMENTATION DETAILS.

	full generic	direct LUT	switch matrix	MIPS CPU
Xilinx Zynq	1247 LUTs 591 slices	274 LUTs 143 slices	96 LUTs 24 slices	886 LUTs 290 slices
X. Spartan-6	1500 LUTs 498 slices	273 LUTs 89 slices	96 LUTs 24 slices	953 LUTs 301 slices
X. Spartan-3	1440 LUTs 1644 slices	358 LUTs 191 slices	– –	1428 LUTs 844 slices
A. Stratix IV	1732 LEs	363 LEs	–	899 LEs
A. Cyclone II	1836 LEs	373 LEs	–	1282 LEs

As we target portability, we implemented our case study for different target FPGAs from different vendors, as listed in Table II. We selected 6-input LUT devices (Altera Stratix-IV, Xilinx Spartan-6, and Xilinx Zynq) and 4-input LUT devices (Altera Cyclone-II and Xilinx Spartan-3). We used latest vendor tools for the experiments and we kept the default options. Note that the synthesis results do not allow a statement about which device is best suitable to implement overlay architectures as small changes in the overlay architecture can favor heavily a specific target FPGA. We provide the here listed synthesis results with the intend to qualitatively show the trend when using different implementation alternatives.

While the fully generic test case would be functional on all target FPGAs, we only listed the synthesis results for the direct LUT implementation approach without further considering bitstream generation and configuration aspects. However, this holds for the last three FPGAs, while we implemented all three implementation variants for a Xilinx Spartan-6 FPGA (on an Atlys board) and a Xilinx Zynq FPGA (on the ZedBoard). So far, we have to generate bitstreams offline but this functionality could be implemented in a driver which generates partial bitstreams directly from the virtual overlay configuration data.

Table II points out that our direct switch matrix implementation has a $21\times$ cost advantage over the full generic and a $3.7\times$ advantage over implementing the interconnection network with route-though LUTs (in terms of used slices). Our present implementation has headroom for further optimizations because only 8 Spartan-6 slices are actually needed for implementing the overlay logic. The other 16 slices are only used to connect the overlay with the CPU. In the future, we will change this by using not only pin-to-pin routing between primitives, but arbitrary combinations of switch matrix wire and pin routing. This would give as a $63\times$ and respectively $11\times$ advantage over the related approaches [1] and [2]. Under ideal conditions, our overlay implementation approach would have the same logic cost as compared with a native implementation of a custom instruction directly to a Spartan-6 FPGA. However, in practice we estimate a penalty of $2\times$ – $5\times$ for the overlay.

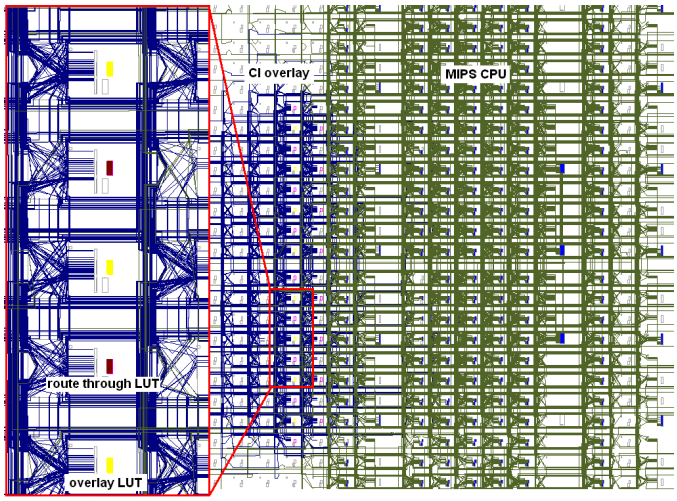


Fig. 5. FPGA editor screen shot of a MIPS CPU with an attached overlay which is implemented by directly using switch matrix multiplexers. Note that this is the overlay reservation that needs a preprocessing step to generate the initial configuration bitstream of the system.

TABLE III
OVERLAY RESERVATION ROUTING RESOURCE STATISTICS AND CPU TIME. THE PATH DELAY WAS MEASURED USING THE FPGA EDITOR. THE SPEED GRADE WAS -3 FOR SPARTAN-6 AND -1 FOR THE ZYNQ FPGA.

	number of wires	average wires	longest path	longest path delay	CPU time
Xilinx Zynq	4547	3.2	6 (31 times)	1.5 ns	35 m
X. Spartan-6	4129	2.9	7 (3 times)	1.8 ns	6.5 h

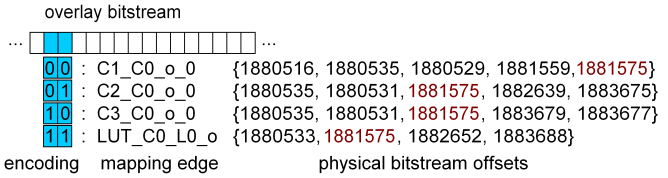


Fig. 6. Overlay bitstream encoding. The bitstream is a sequence of fields, each encoding a specific overlay resource (LUT table or switch matrix multiplexer). The example shows the encoding for a result multiplexer (the output back to the CPU). The physical bitstream offsets denote the bits to be set in the target bitstream for activating a specific multiplexer setting inside the switch matrices of the target FPGA.

B. Computing Overlay Reservations with GOAHEAD

For implementing our new direct mapping approach, we added the baseline and the advanced overlay network reservation algorithm to the tool GOAHEAD [14].[2] GOAHEAD is preliminary designed for implementing advanced systems using partial reconfiguration. The tool provides floorplanning capabilities, physical constraints generation, and various powerful netlist manipulation commands. GOAHEAD builds up the architecture graph of any recent Xilinx FPGA by reading in a device description provided by Xilinx in the XDL format (Xilinx Design Language) [15]. On an architecture graph, a path search from and to any wire port or primitive pin can be performed using the search strategies breadth-first, depth-first, and A*. XDL was also used to add the reservation graph to our static system.

The overlay reservation graph (see Figure 4) of our case study consists of 1408 mapping edges and the baseline algorithm failed to find a full reservation graph for this overlay on the used Xilinx Spartan-6 and Zynq FPGAs. When using Algorithm 2, correct implementations were found, hence demonstrating the benefit of sink and source tree sharing in the reservation graph. However, after an initial run, not all paths got reserved due to resource conflicts and we started the search again with the paths that were left over routed first (to prioritize them). While this resulted in a full reservation for Zynq after only one rip-up iteration, it took 20 rip-ups to find a full reservation for Spartan-6.

Table III lists statistics on the used routing resources and the CPU time needed to compute the reservation (on a laptop with an Intel T4200 CPU running at 2GHz). The resource results for the Spartan-6 overlay are close to the overlay implemented on a Zynq FPGA. This was expected because both FPGAs share a similar routing fabric and because the relative placement of the overlay primitives was identical in both cases. It can be seen that an overlay FPGA wire takes on average about three physical wires and the longest path delay of a single wire is about 3 times the path delay of a long physical wire. As we map overlay LUTs directly to physical LUTs, this will result in a performance of about one half to one third in clock speed as compared to a direct implementation on the target FPGA.

Figure 5 shows a screenshot of a system consisting of a baseline MIPS CPU system and an attached overlay for custom instructions implemented on a Xilinx Spartan-6 FPGA. The system was built by firstly computing the overlay and reserving all its used resources for a second incremental implementation step where we added the MIPS CPU system. GOAHEAD was used to generate all physical constraints for the Xilinx vendor tools when running the second implementation step.

C. Overlay Configuration

In this paper, we store the configuration as a bitstream of concatenated fields, each denoting the setting of a specific resource of the overlay encoded in binary format. This could be the table of a LUT in an overlay logic cell or an overlay switch matrix multiplexer. An example of the configuration of an 4-input overlay switch matrix multiplexer is shown in Figure 6.

This bitstream can be directly used with the full generic RTL implementation of the overlay (see Table II) by simply shifting in the stream into a long shift register storing the configuration values. This is possible, because the bitstream and the internal shift register have been designed to match with respect to the bit fields and the encoding. Table IV, lists a breakdown of the bits needed to configure the CI overlay. In our current implementation, we used a single sequential chain which consequently needs 1696 clock cycles for the full configuration. By using 8 (or 32) parallel chains, only 212 (or 53) shift cycles are needed to reconfigure a new custom instruction.

When considering the overlay implementation using LUTs in route-through mode (called *direct LUT* in Table II), we have

[2]Available here: http://www.mn.uio.no/ifi/english/research/projects/cosrecos.

TABLE IV
CONFIGURATION BITSTREAM BREAKDOWN FOR THE CI OVERLAY.

	minimal number of config. bits	
LUT table values	$32 \cdot 2 \cdot 16$	= 1024
logic cell muxes	$32 \cdot 1$	= 32
LUT input muxes	$32 \cdot 4 \cdot \lceil log_2 7 \rceil$	= 384
result muxes	$32 \cdot \lceil log_2 4 \rceil$	= 64
inter-cluster muxes	$4 \cdot 3 \cdot 8 \cdot \lceil log_2 4 \rceil$	= 192
sum		= 1696 (212 bytes)

to change LUT-values only (without touching the routing of the target FPGA). In the case of the Xilinx Spartan-6 FPGA, it took 89 slices for the implementation of the whole CI overlay. When constraining these slices into a rectangular region of the height of a clock region (16 CLBs in height), $\lceil \frac{89}{16} \rceil = 6$ CLB columns have to be written under ideal conditions. To be more precise, only the LUT table values have to be written which are stored in 8 130-byte frames. As a consequence, $6 \times 8 \times 130 = 6240$ byte, plus the configuration commands have to be written to the configuration port of the target Spartan-6 FPGA [16]. In order to use the route-through mode, the overlay bitstream has to be initially translated into route-through table functions for the LUT. In addition, LUT table values have to be reordered according to the target bitstream format. However, we have not further implemented this approach.

The CI overlay case study using our new implementation approach of directly utilizing the switch matrix multiplexers of the target FPGA needs the reconfiguration of 5 CLB columns (see Figure 5) for loading a new custom instruction. With 31 130-byte frames, this results in $5 \times 31 \times 130 = 20\,\text{KB}$ of data to be send to the configuration port. In our test system, we configure the FPGA in 16-bit ICAP mode at 50 MHz with the help of a dedicated DMA controller. This takes a bit more than 0.2 ms for the CI configuration process. GOAHEAD allows the generation of netlists from which we derived the bit position offsets to be set in the target FPGA bitstream in order to set a specific mapping edge of the overlay network (see Figure 6 that gives the offsets needed for one overlay multiplexer example).

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we demonstrated for the first time how to implement a fine grained overlay on a physical FPGA by mapping the overlay interconnect directly into the switch fabric of a physical FPGA. With this approach, we achieved a $21\times$ ($3.7\times$) less costly implementation than [1] and ([2]), while still having headroom for substantial further improvement. We showed that this approach can be used for implementing custom instruction set extensions that could be directly ported to various FPGA platforms.

As the next step, we want to scale our approach to much larger systems with potentially many thousands of overlay LUTs. Moreover, we aim to integrate our work into the VTR tool flow [17]. This would provide academia with a full flow allowing FPGA architecture exploration all the way down to an emulated device.

ACKNOWLEDGMENT

This work is supported in part by the Norwegian Research Council under grant 191156V30 and by the Education, Audiovisual & Culture Executive Agency (EACEA) under the Lifelong Learning Program (LLP) with project number 518565-LLP-1-2011-1-NO-ERASMUS-ESMO and project title eDiViDe. In addition, we would like to thank Michael Yue and Douglas Sim for developing a negotiation-based routing algorithm for computing overlay reservations. This research was done as part of their project work for a course on CAD for FPGAs at UBC.

REFERENCES

[1] R. Lysecky, K. Miller, F. Vahid, and K. Vissers, "Firm-core Virtual FPGA for Just-in-Time FPGA Compilation," in *Proc. of the 13th int. Symp. on Field-Programmable Gate Arrays (FPGA)*, 2005, pp. 271–271.

[2] A. Brant and G. G. F. Lemieux, "ZUMA: An Open FPGA Overlay Architecture," in *Proc. of the 20th Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 2012, pp. 93–96.

[3] L. Lagadec, D. Lavenier, E. Fabiani, and B. Pottier, "Placing, Routing, and Editing Virtual FPGAs," in *Proc. of the 11th Int. Conf. on Field-Programmable Logic and Applications (FPL)*, 2001, pp. 357–366.

[4] J. Coole and G. Stitt, "Intermediate Fabrics: Virtual Architectures for Circuit Portability and Fast Placement and Routing," in *Proc. of the 8th Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*. ACM, 2010, pp. 13–22.

[5] P. M. Athanas and H. F. Silverman, "Processor Reconfiguration Through Instruction-Set Metamorphosis: Compiler and Architectures," *IEEE Computer*, vol. 26, no. 3, pp. 11–18, 1993.

[6] J. R. Hauser and J. Wawrzynek, "Garp: a MIPS Processor with a Reconfigurable Coprocessor," in *Proc. of the 5th IEEE Symp. on FPGA-Based Custom Computing Machines (FCCM)*, 1997, p. 12.

[7] M. J. Wirthlin and B. L. Hutchings, "DISC: the Dynamic Instruction Set Computer," in *Proc. on Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing (SPIE) 2607*, J. Schewel, Ed. Bellingham, WA: SPIE – The International Society for Optical Engineering, 1995, pp. 92–103.

[8] D. Koch, C. Beckhoff, and J. Torresen, "Zero Logic Overhead Integration of Partially Reconfigurable Modules," in *Proc. of the 23rd Symp. on Integrated Circuits and System Design (SBCCI)*, 2010, pp. 103–108.

[9] A. Dehon, "Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, why you don't really want 100% LUT utilization," in *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 1999, pp. 69–78.

[10] D. Koch, *Partial Reconfiguration on FPGAs – Architectures, Tools and Applications*. Springer, 2003.

[11] G. Lemieux and D. Lewis, *Design of Interconnection Networks for Programmable Logic*. Kluwer Academic Publishers, 2003.

[12] C. Beckhoff, D. Koch, and J. Torresen, "Short-Circuits on FPGAs Caused by Partial Runtime Reconfiguration," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, September 2010, pp. 596 –601.

[13] E. Hung and S. J. E. Wilton, "Towards simulator-like observability for fpgas: a virtual overlay network for trace-buffers," in *Proc. of the Int. Symp. on Field Programmable Gate Arrays, (FPGA)*, 2013, pp. 19–28.

[14] C. Beckhoff, D. Koch, and J. Torresen, "GoAhead: A Partial Reconfiguration Framework," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, 29 2012-may 1 2012, pp. 37 –44.

[15] Christian Beckhoff and Dirk Koch and Jim Torresen, "The Xilinx Design Language (XDL): Tutorial and Use Cases," in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, June 2011, pp. 1 –8.

[16] Xilinx Inc., "Spartan-6 FPGA Configuration User Guide UG380 (v2.5)," 2013, available online: `www.xilinx.com/-support/documentation/user_guides/ug380.pdf`.

[17] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson, "The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing," in *Pro. of the 20th Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, 2012, pp. 77–86.