ARCHITECTURE EXPLORATION AND TOOLS FOR PIPELINED COARSE-GRAINED RECONFIGURABLE ARRAYS

Florian Stock [*]

Tech. Univ. Braunschweig
Dept. for Integrated
Circuit Design (E.I.S.)
Braunschweig, Germany
stock@eis.cs.tu-bs.de

Andreas Koch

Tech. Univ. Darmstadt
Embedded Systems and
Applications Group (ESA)
Darmstadt, Germany
koch@esa.cs.tu-darmstadt.de

ABSTRACT

We present a heavily parametrized tool suite that allows the modeling and exploration of heterogeneous, coarse-grained, heavily pipelined reconfigurable architectures. Our tools perform a simultaneous mapping and pipelining-aware placement, which is then followed by a congestion-avoiding router. Initial experiments show that this flow can succeed in implementing applications with smaller track count and reduced connectivity than existing commercial tools, suggesting changes to the original array architecture. The placer can reduce pipeline latency mismatches on converging paths, simplifying the problem for a pipelining-aware routing step.

1. INTRODUCTION

When designing a new reconfigurable device architecture, the quality of the architecture is linked tightly with the quality of supporting design tools. However, effective tool development is commonly performed only for stable target architectures.

To solve this chicken-and-egg problem, highly parametrizable tool suites exist that can quickly adapt to architectural changes by using flexible architecture definitions instead of hard-coded descriptions. While less efficient than dedicated tools (that can special-case architectural features), tool suites for architecture exploration often trade run-time for flexibility. Note that the quality-of-results should *not* be compromised by the exploratory capabilities of the software, since the conclusions drawn from the results will in turn guide further development of the architecture.

In contrast to previous efforts [1] [2], which targeted fine-grained FPGA-like devices, our efforts deal with coarse-grained heterogeneous fabrics that have word-wide ALUs as processing elements. Furthermore, we also consider pipelining effects due to registers both in the processing elements as well as in the interconnect. Specifically, our work uses a device structure similar to that of the PACT XPP2 [3] as basis. However, many of our results will be generally applicable to coarse-grained arrays with pipelined interconnect.

From that starting point, we evaluate both architectural extensions as well as suitable tool algorithms to improve the quality of the current, unmodified state of both architecture and tool flows. We examine how to improve the silicon area usage of the architecture and how to improve the processing throughput.

2. BASE ARCHITECTURE

The base architecture consists of a matrix of tiles, where each tile contains both interconnect as well as compute blocks (called *objects*). Both the number as well as the nature of the objects within a tile may vary from tile to tile. The arrangement in Figure 1 is typical, however: The primary compute element is an ALU (having a word width of 24...32 bits), that can also include a multiplier. In addition to the word-wide *data* operations, the ALU can also operate on boolean single-bit `event` values. The latter are used for condition codes and synchronization information. Secondary objects supply registers or simple logic functions on events.

Connections are made using horizontal tracks running above and below the rows of objects, and through the register objects (which connect vertically across a row). The objects reach the tracks by going through configurable connection points. Tracks are unidirectional and propagate their data either to the left or to the right. The two value types (data and event) are carried on separate tracks in the interconnect network. Each track runs only within its own tile, at the boundaries they connect to an adjacent tile via a segment switch (which also registers for the signals).

Figure 2 depicts a complete array and shows how the tracks

[*]Supported by PACT XPP Technologies AG, Germany.

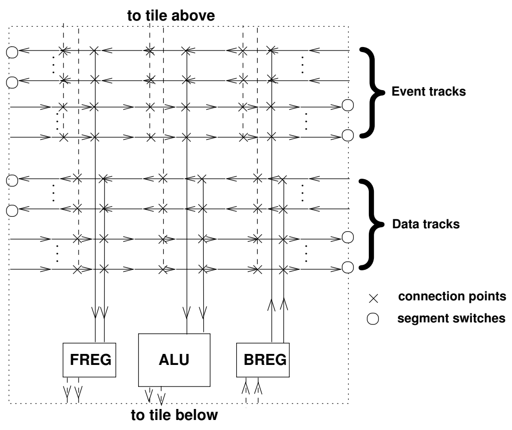

Fig. 1. Simplified architecture tile

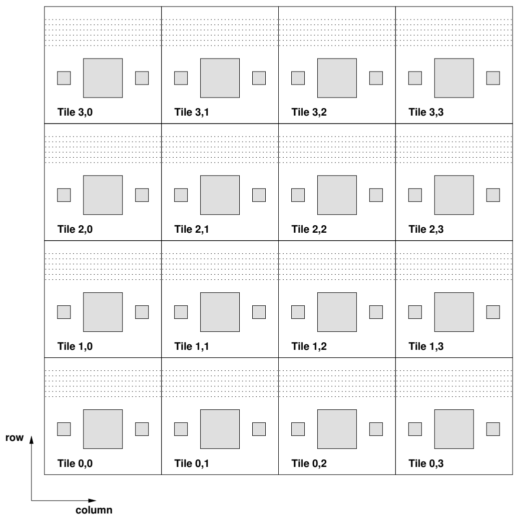

Fig. 2. A sample 4×4 array

in the abutting tiles form segmented busses spanning the entire width of the array.

Note that our work treats an object as a black-box, we do not consider its internal structure. However, our tool flow *does* consider how to map the abstract logical elements, given in the form of a structural, netlist-like description, to the actual physical blocks of the device (see below).

For high performance, the array is heavily pipelined: The output of each operator is registered, explicit registers exist in the form of the FREG/BREG objects, and each segment switch also contains a register (as described above). The FREG and BREG blocks are generic storage elements, capable of handling both data and event information. They differ only in their propagation direction (forward up-down and backward down-up).

An actual silicon device containing the XPP array as an synthesized IP block is assumed to run at a fixed clock rate, matched to the inter-register delay achievable on the desired target technology. Thus, the practical performance is dependent on the sequential latency and throughput of the placed/routed application. The throughput, in turn, depends on equalizing the latencies on all paths converging on a single object.

To this end, we consider expanding the base architecture with FIFOs for introducing additional latency into signal paths. These FIFOs will be placed at two places in the architecture: First, PINFIFOs at the *inputs* of objects control the individual latency on a per-sink basis. Second, the segment switches are also expanded beyond the initial single register stage with SEGFIFOs. This two-pronged approach distributes the available de-skewing latency across the fabric, instead of concentrating it just at the objects. Note that placing FIFOs at the output ports of objects is not efficient: All fan-outs of a source would be delayed by an equal amount, which does not help when balancing the latency for each separate path. The mapping tools also need to be made aware on how to employ these hardware features to best effect.

3. ARCHITECTURAL PARAMETERS

Specific instances of the base fabric are defined in architecture definition files. They describe the following characteristics:

The names of functions. These are the elements the input netlists are composed from. Examples of functions are ALUs, various LUTs, registers, RAM blocks, and I/O streams.

Mapping rules from functions to actual physical objects. For example, a DREG function specifically stores data values (word-wide). It can be implemented either in a physical DFREG data register within an FREG object, or in a physical DBREG data register in a BREG object.

The number of routing tracks in each direction for both data and event tracks. In the original XPP2, this was set to 8 and 6 respectively, leading to total number of 16 data and 12 event horizontal tracks per tile. However, as will be shown in Section 5, with our improved tools, this can generally be reduced to 4 tracks each for data and event lines in most applications examined. For more concise architectural descriptions, named subgroups of tracks can be defined and referred to later when describing the detailed connectivity.

The physical objects implemented in this architecture instance. These are described by a name, their connectivity and mapping characteristics. In the connectivity section, each port on the object is identified by a number and its type (data, event), its direction as well how it connects to the hor-

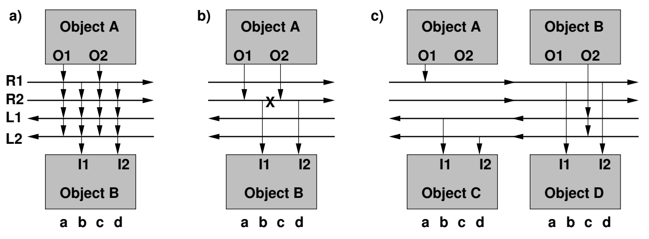

Fig. 3. Interconnect details

```
{DFREG}:
  ( DFREG.<D.Q>; <0.4>,<1.5>,<2.6>,<3.7> );
{EFREG}:
  ( EFREG.<E.R>; <8.12>,<9.13>,<10.14>,<11.15> );
{DFREG, EFREG}:
  ( DFREG.<D.Q>; <0.4>,<1.5>,<2.6>,<3.7> );
  ( EFREG.<E.R>; <8.12>,<9.13>,<10.14>,<11.15> );
```

Fig. 4. Logical-to-physical object mapping for FREG

izontal tracks. The latter is achieved by giving the location (top, bottom) of the port and its horizontal position. Figure 3.a shows examples for such positions labeled with a. . . d, their actual use will be described later. Additional specifications describe the maximum depth of FIFOs on input ports (PINFIFOs).

For **mapping**, each physical object describes which logical function(s) it can realize, and the required assignment of function ports to physical pins, if any. This information is used during a combined mapping / placement step: Here, we simultaneously map a logical function to a suitable kind of object, and place that at a specific geometric position on the array. Figure 4 is an excerpt of such an architecture definition for a physical FREG object: The object can implement DFREG (data forward register) and/or EFREG (event forward register) functions. Up to four instances of each function can be implemented in the FREG. As an example for a pin assignment, consider an instance of a DFREG function. It will have to use pin 0 for the logical D port (input), and pin 4 for the logical Q port (output). Other instances will use the other pin/port combinations. Analogously for the EFREG event register function, with its input port E and output port R.

A final section describes the **composition of the entire reconfigurable array** by listing, for each tile in the matrix, the contained physical objects. Note that the linear order of the objects is relevant (see below).

In addition to these detailed characteristics, various aspects can be set globally for the entire array:

Control, whether the tracks within a tile can be **segmented between connection points**. In effect, this determines whether the use of a track for a connection blocks this track for other, unrelated connections. An example of this is shown in Figure 3.b. The independent nets (ObjectA.O1,ObjectB.I1) and (ObjectA.O2,ObjectB.I2) can only be routed together on Track R2 if the track is segmented at 'X'. Otherwise, the second net has to be routed on R1. The re-usability of tracks on a per-segment basis is dependent on the horizontal order of connection points on the tracks.

Another parameter controls whether **fan-out can occur only at the source port** of a net, or whether it can occur at any track at the connection point-level. Figure 3.c shows the difference: In the first case a net sourced by ObjectB.O2 and sunk at ObjectC.I1 and I2 requires two tracks L1 and L2. In the second case, a similar net connecting ObjectA.O1 to ObjectD.I1 and I2 requires only the track R1.

A last parameter describes the maximum depth of FIFOs in the segment switches (**SEGFIFOs**)

4. CORE ALGORITHMS

The parametrized tool flow consists of a combined mapping/placement step followed by a routing procedure.

Mapping is handled simultaneously placement, which in itself is realized using the self-adaptive simulated annealing suggested by VPR [1]. The rules in the architecture definition files describe which logical functions are compatible with which physical objects, and thus determine the valid moves / locations for each function. The placement step thus also performs the mapping / packing of logical functions to physical objects as described above.

Ordinarily, the cost function in VPR is the weighted sum of a delay and a wiring length term. However, due to the fixed-frequency nature of our target architecture, delay is no longer relevant. On the other hand, the balancing of sequential pipeline delays is crucial and must be considered in the placement cost function. Thus, our "raw" *cost* function consists of a weighted sum of the balancing latency mismatch and the wiring length, with the weight factor λ controlling the trade-off between both terms. This raw cost is then self-normalized (as in VPR) to get the actual cost. As will be shown in Table 2, increasing λ does indeed lower the balancing mismatch, but excessive values lead to a deterioration of result quality.

Figure 5 shows the cost function used in our placer. The converging path sets C of paths to balance are specified by the user as part of the circuit description. They are part of the application, since an automatic approach could not distinguish between intentional latencies in a path (e.g., for a FIR filter), and mismatched latencies that it needs to balance. Since the actual routing is only performed after placement, the cost function is only an estimate, both for the wiring and the balancing parts. The first uses the same heuristics as VPR (fanout dependent correction factor on bounding box),

$$
\begin{aligned}
\mathit{cost}(\text{placement}) &= \lambda \cdot \mathit{balcost}(\text{placement}) \\
&+ (1-\lambda) \cdot \mathit{wirecost}(\text{placement}) \\
\mathit{balcost}(\text{placement}) &= \sum_{\text{Converging path sets } C \text{ in placement}} \mathit{mismatch}(C) \\
\mathit{mismatch}(\{P_1,\ldots,P_k\}) &= \max(0, \max_{1\le j\le k}(\mathit{latency}(P_j) \\
&\qquad - \min_{1\le j\le k}(\mathit{latency}(P_j) + \mathit{fifos}(P_j)))) \\
\mathit{latency}((e_1,\ldots,e_l)) &= d_{\text{manhattan}}(e_1,e_l) \\
\mathit{fifos}((e_1,\ldots,e_l)) &= |\mathit{column}(e_1) - \mathit{column}(e_l)| \cdot \text{SEGFIFOS} \\
&\quad + |\mathit{row}(e_1) - \mathit{row}(e_l)| \cdot \text{PINFIFOS} \\
\text{SEGFIFOS} &= \text{max. depth of FIFOS in segment switches} \\
\text{PINFIFOS} &= \text{max. depth of FIFOS in object input pins}
\end{aligned}
$$

Fig. 5. Pipeline balancing in placer cost function

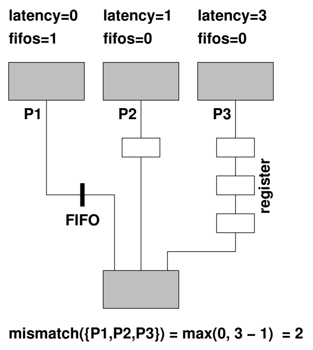

Fig. 6. Pipeline latency balancing

the second estimates the maximum difference of converging paths. This is done using the Manhattan distance as a base for computing the latency, and then attempting to extend shorter paths by activating additional FIFO stages in the segment switches (up to the parametrized maximum SEGFIFOS) and object input pins (up to the parameter PINFIFOS).

Figure 6 shows an example. Here, we assume a converging path set of $C = \{P_1, P_2, P_3\}$. Each path has a number of registers, and a maximum number of available FIFO stages. In the example, P_3 has the longest latency of 3. To determine the mismatch, we now compute the shortest path that results even if we slow down the paths as much as possible using FIFOs (e.g., in the segment switches or in the input ports). In this case, P_1 ends up at a total latency of 1 (due to switching on the single available FIFO stage), while P_2 contains a single register, but no FIFO stages. In the final result, the difference between the slowest and the fastest paths (the balancing mismatch) is 2.

In this first stage of our work, one of our aims was to determine whether reducing the generous routing structure of the original commercial architecture was possible without

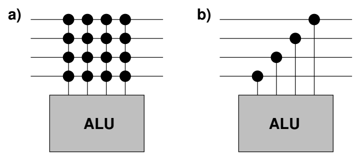

Fig. 7. Comparison of normal and depopulated architecture

adversely affecting routability. To this end, we implemented a negotiation-based router in the PathFinder [4] fashion, improved by adding the closest-sink first heuristic of [5]. Note that, in its current stage, the tool flow does *not* attempt to consider pipeline balancing during routing, but only during placement. However, we do already perform an exact balancing analysis after routing, retrieving the largest difference between converging paths as well as the sum of differences to judge the overall quality.

5. EXPERIMENTAL RESULTS

In our first set of experiments, we focused on reducing the device area by relying the improved router to succeed with fewer tracks and more limited connectivity. To test the latter, we depopulated the connection point patterns between ALU objects and the horizontal tracks. Instead of the full connectivity shown in Figure 7.a, which is also employed in the original architecture, we employed the more restricted pattern in Figure 7.b. The ALUs were modified to make the input pins commutative. Furthermore, we enabled intra-tile segmented wires and allowed fan-outs at connection points[1].

Table 1 presents the results of our approach for a number of sample applications. These were either compiled from C to hardware using a vectorizing C compiler [6], or have been manually described in the structural netlist format. When considering the complexity of these sample circuits (given as number of objects and nets), remember that the array is a *coarse-grained* architecture with data flow control integrated directly into the fabric. For comparison, [7] discusses the practical use of a 4x3 array with just 12 ALUs to accelerate various video decoders. The target array we used here has a dimension of 8x8 tiles, with each tile having the structure shown in Figure 1.

As can be seen in the Table, almost all of the samples are routable on the depopulated architecture. Furthermore, in most cases, they require considerably fewer tracks than the original array design. Also, circuits that turned out to be unroutable on the depopulated connection pattern are also unroutable using the original, fully populated one.

[1]According to PACT XPP Technologies AG, these two options do not lead to significant increases in silicon area.

.grph-file (xppvcgen)	#Objects	#Nets	Routability	.grph-file (manual)	#Objects	#Nets	Routability
adpcm_decoder.grph	44	55	4444s	wavelet/main.grph	14	12	4444
arrayfir2.grph	17	20	4444	2dfir/sampfilt.grph	134	136	5555s
arrayfir3.grph	38	46	4444	2didct/invdct8.grph	259	264	x
arrayiir.grph	29	34	4444	compcorr/comp_corr.grph	99	125	4444
chendcttest2.grph	37	46	4444	single_conf_DCT32/main.grph	200	232	6688s
chendcttest.grph	36	44	4444	DCT_2D_XPP20/dct_main.grph	76	94	4444
condit3.grph	29	41	4444	enhance/main_5_dead_MAIN_5_DEAD.grph	63	92	4444
condit.grph	20	28	4444	fft1024/main.grph	61	93	4444
corr16_pipe.grph	60	74	4444	fft256/main.grph	42	56	4444
dct1.grph	91	118	4444	fft_64/fft64.grph	136	155	5555
dct2.grph	69	88	4444	fft64_complex/fft64_fft64_mod.grph	26	35	4444
dct3.grph	70	93	4444	fir/fir_16.grph	35	33	4444
edge3x3.grph	53	68	4444	h264/forward.grph	140	140	4444
ellip1.grph	14	24	4444	IDCT_2D_XPP20/idct_main.grph	88	120	4444
filt.grph	25	33	4444	IDCT_2D_XPP64/main_idct_main.grph	169	223	6688
iquant.grph	25	30	4444	idct8x8/idct8x8.grph	259	264	x
iwt2DShore.grph	64	79	4444	jacobi/jacobi_MAIN.grph	83	107	4444
memdistr.grph	88	103	4444	median/median_main.grph	59	81	5555
mult.grph	70	91	4444	mpeg4_all/XPP_mpeg4_convert.grph	107	136	4444
pipetest2.grph	31	37	4444	mpeg4_all/XPP_mpeg4_yuv2yuv.grph	68	89	4444
pipetest.grph	12	13	4444	mpeg4_all/XPP_mpeg4_decoder_conf1_mod.grph	190	250	6688s
simpedge2.grph	48	60	4444	mpeg4_all/XPP_mpeg4_decoder_conf2_mod.grph	76	97	4444
simpedge.grph	65	75	4444	mpeg4_all/XPP_mpeg4_dummy_mod.grph	3	1	4444
sort7.grph	61	75	4444	mpeg4_all/XPP_mpeg4_ME.grph	105	148	5555
streamfir2.grph	22	24	4444	mpeg4_all/XPP_mpeg4_framediffmod.grph	13	20	4444
streamfir.grph	9	9	4444	RS_encoder/rs_encoder.grph	11	12	4444
test10.grph	7	7	4444	rake1/DeScrambleDeSpread.grph	72	88	4444
test11.grph	8	8	4444	rake2/DeScrambleDeSpread.grph	26	35	4444
test12.grph	9	10	4444	MPEG4_Huffman_Dec/ivop_ReadMPEGData.grph	122	151	5555s
test13.grph	11	13	4444				
test15.grph	7	8	4444				
test16.grph	10	12	4444				
test17.grph	19	23	4444				
test3.grph	10	12	4444				
test4.grph	8	8	4444				
test5.grph	6	7	4444				
test6.grph	7	11	4444				
test7.grph	10	15	4444				
test8.grph	24	27	4444				
test9.grph	5	4	4444				
testbool.grph	15	16	4444	**Legend**			
test.grph	9	11	4444	*ddee*: Data/event tracks Left/right			
vecaddnew.grph	53	64	4444	**s**: Placer run in high-quality mode			
vecadd.grph	16	20	4444	**x**: unroutable in 8866s			

Table 1. Placement and routing results

Application	Balancing		Mismatch $\lambda = 0$		Mismatch $\lambda = 0.75$		Mismatch $\lambda = 0.9999$		XMAP	Inherent Mismatch	
	PathSets	Nets	Sum	Max	Sum	Max	Sum	Max	xmap	Sum	Max
adpcm_decoder	1	34	33	17	28	14	28	14	x	10	3
arrayfir2	1	18	23	13	21	19	22	13	x	9	3
arrayfir3	3	38	21	10	76	14	62	15	x	27	3
arrayiir	1	18	8	4	15	5	13	4	ok	11	3
chendcttest2	3	20	0	0	0	0	0	0	ok	0	0
chendcttest	3	17	0	0	0	0	0	0	ok	0	0
compcorr__comp_corr	5	73	46	9	49	12	31	5	x	18	2
condit3	1	2	0	0	7	4	1	1	ok	0	0
condit	1	12	8	7	4	4	5	4	ok	4	2
corr16_pipe	3	50	326	18	143	7	489	25	x	178	8
dct1	5	128	1168	79	850	45	891	56	x	339	17
dct2	3	94	1280	84	649	46	819	60	x	335	22
dct3	4	38	14	14	11	10	19	18	x	7	4
edge3x3	2	43	27	8	16	5	39	9	x	17	3
enhance__main_5	2	3	0	0	0	0	0	0	ok	0	0
fft64_complex	1	17	11	11	10	10	6	6	ok	4	4
filt	1	13	4	3	12	11	11	10	ok	6	4
iquant	1	15	0	0	3	3	3	3	x	5	3
iwt2DShore	1	41	44	27	40	20	37	23	x	12	6
memdistr	5	125	1278	71	521	41	709	48	n.r.	308	17
mpeg4_decoder_conf2_mod	1	46	165	23	120	17	57	10	-	63	5
mpeg4_framediffmod	1	13	0	0	0	0	0	0	-	0	0
mpeg4_yuv2yuv	1	33	1	1	1	1	1	1	-	2	1
mult	4	38	0	0	6	5	1	1	ok	1	1
pipetest	1	4	0	0	0	0	0	0	ok	0	0
simpedge2	3	35	13	10	8	8	14	11	x	4	3
simpedge	3	18	16	8	9	9	9	6	x	8	4
sort7	2	14	5	5	5	5	4	4	ok	4	2
test8	1	5	0	0	0	0	0	0	ok	0	0
test	1	7	0	0	0	0	0	0	ok	0	0
vecadd	2	12	0	0	0	0	0	0	ok	0	0
vecaddnew	1	4	0	0	0	0	0	0	ok	0	0
Sums			4491	422	2604	315	3271	347		1372	120

Table 2. Balancing-oriented placement

In the next set of tests, we examine the quality of our balancing-oriented placement in Table 2. Of course, we consider only the sub-set of applications that actually has balancing constraints. For each applications, we give the number of path sets C and the total number of nets affected by them. Then, we show the sum and maximal values of the mismatches for three values of λ. For comparison, we show results obtained when using a commercial, mainly balancing-oriented router in the column labeled XMAP. Since XMAP does not produce detailed mismatch data, we use a coarser presentation: 'ok' means that the router was able to achieve perfect balancing, 'x' indicates it failed to do so, 'n.r.' stands for "not routable", and '-' signifies that no data was output by XMAP. In the last two columns, we show the inherent balance mismatches in the circuit (without considering placement and routing). For these benchmarks, we used SEGFIFOS=1 and PINFIFOS=0, which is similar to the setup of the original architecture.

From these numbers, it is obvious that the balancing-oriented placement can indeed reduce both the maximal and the overall mismatches. Compared to the purely wiring-driven solution ($\lambda = 0$), a value of $\lambda = 0.75$ leads to a reduction of the summed maximum mismatch by 25%, and reduces the overall mismatches by 42%. On the other hand, it is also apparent that a balancing-aware router is sorely needed: Even with these reduced values, the summed maximum balancing mismatch increases over that already present in the circuits almost by a factor of three. Only in very rare cases can the balancing-aware placement actually decrease the inherent mismatch by approriately using the interconnect registers. As the XMAP column shows, even a simple balancing-aware router can significantly reduce the mismatches.

We also measured the quality of our latency estimation during placement by actually performing a full routing for each step of the simulated annealing. While this is unsuitable for production use (due to excessive run-times), it does produce precise wire-length and latency values. However, this approach does not improve the quality of results beyond those achievable by the estimation.

6. CONCLUSION AND FUTURE WORK

Our work uses a combination of techniques to improve the area efficiency of the architecture family we have examined, and the quality of results of the design tools. The first is accomplished by applying suitably modified routing techniques from fine- to coarse-granular target architectures. The second is achieved by a simultaneous mapping/placement pass, refined to include balancing estimates into the placer cost function. This can significantly reduce both the maximum as well as the overall latency mismatches. Of course, the improvements in throughput required the introduction of balancing FIFOs into the fabric, offsetting the area gains by the better router. Our entire experimental flow was implemented in a very flexible manner, allowing fast and easy exploration of architectural variations by just altering the architecture definitions.

The next step in tool development will be a pipelining-aware router. While approaches such as [8] for the RaPID architecture could be used as a starting point, the techniques are not entirely applicable: RaPID has a simpler, linear arrangement with fewer latency-inducing so-called *bus connectors*. Furthermore, the RaPID tools expect as input a re-timed netlist that *explicitly* enumerates pipeline registers. These are then directly mapped to appropriate bus connectors during placement (similar to our simultaneous placement/mapping step). In our two-dimensional XPP2-related target architecture, signals incur extra latency at tile boundaries in a less predictable fashion (due to the two-dimensional arrangement).

Our current planning is to employ an A^*-like strategy, where the router estimates the latency towards the next sink (similar to the placer) and then routes signals along a least-congestion-cost detour until a sufficient number of registers has been picked up. The balancing can further be improved by adding deeper FIFOs to the input pins of physical objects, this is already covered in our architecture definition model. However, since these FIFOs will increase the overall silicon area, they should be carefully sized only *after* the tool chain has been made fully balancing-aware.

Acknowledgments: The authors would like to thank Daniel Bretz, Martin Vorbach, and Markus Weinhardt, all of PACT XPP Technologies AG, for providing details on the XPP2 architecture and netlists of the sample applications.

7. REFERENCES

[1] V. Betz, J. Rose, and A. Marquardt, "Architecture and CAD for Deep-Submicron FPGAs", Kluwer Academic Publishers, 1999

[2] A. Danilin, M. Bennebroek, and S. Sawitzki, "A Novel Toolset for the Development of FPGA-like Reconfigurable Logic", *Proc. Intl. Conf. on Field-Programmable Logic (FPL)*, Tampere, 2005

[3] `www.pactxpp.com`, 2006

[4] C. Ebeling, L. McMurchie, S. A. Hauck, and S. Burns, "Placement and Routing Tools for the Triptych FPGA", *IEEE Trans. on VLSI*, Dec. 1995, pp. 473-482.

[5] J.S. Swartz, V. Betz, J. Rose, "A Fast Routability-Driven Router for FPGAs", *Proc. Intl. Symp. on FPGAs (FPGA)*, Monterey, 1998

[6] J. M. P. Cardoso, M. Weinhardt, "PXPP-VC: A C compiler with temporal partitioning for the PACT-XPP architecture", *Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL)*, Montpellier, 2002

[7] `http://www.pactxpp.com/xneu/download/PACT_Video_Decoding_White_Paper.pdf`, 2006

[8] A. Sharma, K. Compton, C. Ebeling, S. Hauck, "Exploration of Pipelined FPGA Interconnect Structures", *Proc. Intl. Symp. on FPGAs (FPGA)*, Monterey, 2004