A METHOD FOR FAST HARDWARE SPECIALIZATION AT RUN-TIME

Karel Bruneel[*], *Peter Bertels*[†], *and Dirk Stroobandt*

Department of Electronics and Information Systems
Ghent University
Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium
{kbruneel, pbertels, dstrooba}@elis.ugent.be

ABSTRACT

Dynamic hardware generation is a powerful technique that can substantially reduce both the required hardware resources and the time needed to perform a calculation, reflected in an improved functional density. This performance improvement is a result of additional run-time optimizations enabled by the knowledge of values at certain inputs at run-time. However, due to the large overhead conventional hardware generation tools incur, the usability of dynamic hardware generation is limited. We present a dual approach that combines compile-time generation of generic hardware and run-time specialization. This drastically decreases the dynamic generation overhead. Our approach is used for dynamic generation of FIR filters and compared to a static and a conventional dynamic implementation. The experiments clearly show that the dual approach improves the usability of dynamic hardware generation.

1. INTRODUCTION

During the design process of a hardware component designers aim at an optimal implementation, taking into account the specific conditions of the application. While a broad range of optimizations is possible at design time many more emerge only at run-time when even more information about the specific application becomes available. One of these optimizations is constant propagation of specific parameters and/or inputs which leads to a more efficient use of the Field Programmable Gate Array (FPGA) fabric.

In this paper we use the design of the FIR filter shown in figure 1 as a running example. At design time the number of filter taps is known as well as the word length of its coefficients. The word length of the input data is also a design constraint. This specification leads to an implementation using generic multipliers. If the designer would also know the exact values of the coefficients, these values could be propagated into the design leading to a specific implementation which is more compact and possibly faster. In applications where the filter coefficients are not fixed, constant propagation is only possible at run-time.

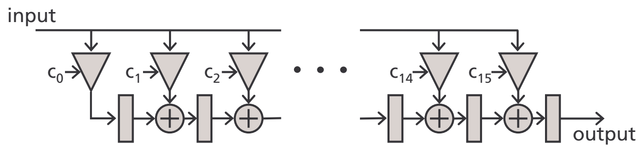

Fig. 1. Finite Impulse Response Filter

Run-time switching between specific implementations is possible due to the reconfigurability of FPGAs. For applications where all possible configurations can be generated at design time, this run-time switching is a sufficient solution. In this paper we focus on applications where more flexibility is needed. The number of specific implementations becomes unmanageable for these applications. Hence it might be more efficient to generate the specific implementations at run-time.

Run-time optimization improves the quality of the generated hardware, but also incurs an overhead. Functional density, introduced in section 2, is a measure for how efficient the FPGA fabric is used. Functional density takes both intrinsic hardware properties and run-time optimization overhead into account. Using functional density we can show under which conditions the optimization gain outweighs the hardware generation overhead.

Standard hardware synthesis tools are very slow, which makes these conditions too stringent for many practical situations. To alleviate these conditions we propose a faster method for hardware generation at run-time in section 3. We achieve this by splitting up the hardware generation in an offline initial compilation and a fast run-time refinement of this compilation. A profound description of two techniques we use for this refinement is given in section 4.

Experimental results for the optimization of FIR filters in section 5 show that our method can effectively increase the functional density.

[*]Karel Bruneel is supported by a BOF grant from Ghent University.
[†]Peter Bertels is supported by a PhD grant of IWT-Vlaanderen.

We situate our work in the broad field of run-time hardware generation in section 6 and formulate some conclusions in section 7.

2. FUNCTIONAL DENSITY

We claim run-time generated circuits can use the FPGA fabric more efficiently than generic, offline generated circuits. A metric to measure this efficiency is the functional density D. This metric combines two important properties of a calculation in hardware: the number of hardware resources, reflected in the area A, and the time T needed to perform this calculation. Functional density is defined as follows:

$$D = \frac{1}{AT}. \tag{1}$$

For the static case where generic hardware is generated at compile-time, A_s is the total hardware cost and $t_{s,exec}$ is the execution time of the hardware:

$$D_{static} = \frac{1}{AT} = \frac{1}{A_s t_{s,exec}}. \tag{2}$$

For dynamically generated hardware as we propose in this paper, T consists of the execution time of the hardware ($t_{d,exec}$) and the time needed to generate the hardware ($t_{generate}$) and to configure the FPGA (t_{conf}). The area is denoted A_d. This leads to the dynamic functional density $D_{dynamic}$, introduced by Wirthlin et al [1].

$$D_{dynamic} = \frac{1}{AT} = \frac{1}{A_d(t_{d,exec} + t_{generate} + t_{conf})} \tag{3}$$

When the generated hardware component is reused several times (n), the overhead of generating this component is amortized over several executions. This results in:

$$D_{dynamic} = \frac{1}{A_d\big(t_{d,exec} + \frac{t_{generate}+t_{conf}}{n}\big)}. \tag{4}$$

For large n the overhead of run-time hardware generation becomes negligible:

$$D_{dynamic} \approx \frac{1}{A_d t_{d,exec}}. \tag{5}$$

At run-time more optimization possibilities emerge resulting in potentially smaller and faster hardware:

$$A_d t_{d,exec} < A_s t_{s,exec}. \tag{6}$$

Under this condition, we can see that $D_{dynamic}$ overtakes D_{static} for large n. We can calculate the break even point N as:

$$N = \frac{A_d(t_{conf} + t_{generate})}{A_s t_{s,exec} - A_d t_{d,exec}}. \tag{7}$$

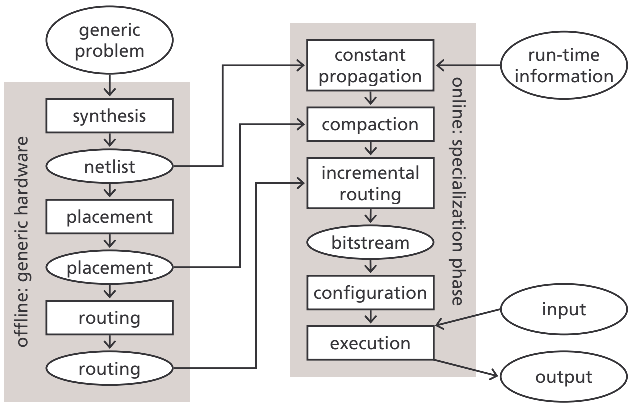

Fig. 2. Method for run time hardware generation

As can be seen in section 5, the hardware generation time $t_{generate}$ is very large for a conventional hardware synthesis tool chain. The break even point N can be reduced by speeding up run-time synthesis. This enables the use of run-time hardware generation in a new range of applications where n is relatively small.

3. DUAL APPROACH

Dynamic hardware generation enables exploitation of optimizations that only emerge at run-time. For some applications, optimizing for specific parameters, requires a completely different design. We aim for another class of applications where the optimizations can be seen as a transformation of an existing design. In this paper we focus on constant propagation. Example applications are: adaptive filtering, key-specific encryption and many others.

These applications enable our dual approach where we combine compile-time generation of generic hardware with run-time specialization as shown in figure 2. As the optimizing transformations are far less complicated than a complete redesign, large speed-ups can be achieved over a conventional hardware synthesis approach. By shifting hardware generation time from run-time to compile-time, we improve the functional density where $t_{generate}$ was the obstructing factor as can be seen from equation 4.

For the offline generation of the generic design, we start with a high level description of the generic problem. This description is synthesized with a conventional tool chain consisting of three consecutive steps: Synthesis, Placement and Routing. This results in a generic netlist, and generic placement and routing information used as a headstart for the online specialization steps.

At run-time we perform the following steps: constant propagation, compaction and incremental routing. Each of these online steps corresponds to an offline preparation step and builds on its results.

The constant propagation step uses the run-time information about constant input values to transform the generic netlist into a specialized netlist. This is done by propagating the constants into the circuit and simultaneously simplifying the circuit. Some logic blocks are removed due to this simplification.

The compaction step combines the generic placement and the specialized netlist and produces a compact specialized placement. This is done in two steps. First the generic placement is pruned only retaining the logic blocks present in the specialized netlist. The emerging free space is fragmented and therefore cannot be used efficiently. Secondly this sparse placement is compacted to reduce the fragmentation, while trying to preserve the placement quality. Because compaction decreases the bounding box area, functional density is improved as can be deduced from equation 4.

Optimally compacting a sparse placement is as hard as finding the optimal placement itself. There is no strict correlation between the place of a logic block in the sparse placement and its place in the optimal placement. Nevertheless we can use the sparse placement as a headstart for a new suboptimal placement considering the generic placement as a condensed representation of the connectivity information of logic blocks. Based on this observation we designed a fast and efficient online heuristic for compaction, presented in section 4.2.

The incremental routing step concludes the online hardware generation. Only those interconnections that were broken during constant propagation and compaction are rerouted.

The bitstream resulting from our online hardware specialization can be used to configure the FPGA after which the hardware component can start executing. During execution the dynamic input is processed by the hardware component generating the desired output.

4. ALGORITHMS

4.1. Constant propagation

The constant propagator takes a netlist and a list of the constant inputs and their constant values. In the netlist logic blocks are annotated with the truth tables of their Look Up Tables (LUTs). The output is a specialized netlist.

First we read the generic netlist and build an internal data structure containing information about logic blocks – consisting of LUTs and Flip Flops (FF) – inputs, outputs and nets. This representation facilitates complex transformations on the circuit.

Secondly we propagate constant inputs one by one in the circuit. The truth tables are simplified simultaneously. In some cases this leads to opportunities to prune the circuit. We use eight simple pruning rules:

1. If a LUT becomes independent of one of its inputs, this input is removed from the sink list of the driving net.
2. If a net has no sinks, its driving LUT or FF is removed.
3. If a LUT is removed, its inputs are removed from the sink list of its respective input nets.
4. If a FF is removed, its input is removed from the sink list of its input net.
5. If a LUT becomes a constant generator, it is removed except for when it drives an output. The constant output of the LUT is propagated.
6. If a FF has a constant input it is removed and the constant input is propagated.
7. If a LUT has only one input and it simply transfers the value to the output, the input net and the output net are merged and the LUT is removed.
8. If all LUTs and FFs of a logic block are removed the logic block itself is removed.

The eighth rule removes logic blocks from the circuit. These blocks are pruned from the netlist but are still present in the generic placement file. The compaction step will solve this problem by removing them from the placement.

Another important property to note is that none of the rules create extra logic blocks. This proves that the set of logic blocks in the specialized circuit after constant propagation will be a subset of the logic blocks in the original design. As a consequence, each logic block in the specialized netlist is still linked to a place in the generic placement. This results in a valid initial placement for all logic blocks, which will be optimized by our compaction algorithm.

4.2. Replacement via Compaction

Some logic blocks have become redundant after constant propagation. Removing these blocks from the generic placement, results in a sparse specialized placement that uses less hardware resources than the original placement. Because of fragmentation the vacant logic blocks do not contribute to an improved functional density.

This sparse specialized placement is transformed into a compacted version by our compaction algorithm. The resulting placement can effectively improve the functional density if it has a reduced bounding box and still a good placement quality. Important factors are routability and the length of the resulting critical path.

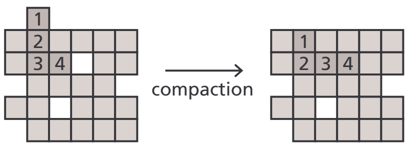

Fig. 3. Principles of Compaction

As stated in section 3 we designed a fast online heuristic for compacting while trying to preserve the placement quality. The generic placement is used as a condensed representation of the connectivity information of logic blocks. This placement is produced by a placement algorithm that obtains good quality by placing strongly connected blocks closely together. We assume that this property still holds for the sparse placement. Hence good placement quality can be achieved by keeping blocks that are closely together in the original placement closely together during compaction.

Our compaction algorithm is a fast heuristic for achieving this goal. The pseudo code is shown in figure 4. We first calculate the bounding box of the sparse placement. Then we iterate over all logic blocks on the border of this bounding box and shift them into the bounding box. This is done by finding the closest free space in the bounding box and then shifting the logic blocks that are in between at a right angle (figure 3). By shifting a logic block over not more than one position every iteration, logic blocks that were closely together stay closely together during an iteration.

```
BoundingBox b = findBoundingBox();
LogicBlock l = b.firstLogicBlock();
while (b.freeSpace() != 0) {
   Place f = b.findClosestFreeSpace(l);
   shift(l, f);
   l = b.nextLogicBlock();
   if (l == null) {
      b = findBoundingBox();
      l = b.firstLogicBlock();
   }
}
```

Fig. 4. Pseudo code of the compaction algorithm

As can be intuitively seen this algorithm is very fast compared to more conventional placement algorithms based on simulated annealing. In section 5 we also show the placement quality can be preserved.

5. RESULTS

Our implementation is based on VPR (Versatile Place and Route) [2]. After synthesis we use Vplace for placing the generic circuit in the offline phase. For the online specialization we use our own constant propagation and compaction algorithms, described in sections 4.1 and 4.2 respectively. Currently we have not yet implemented an incremental routing algorithm. Therefore the specialized circuit is fully routed at run-time with Vroute, the VPR router.

Although the approach proposed in this paper is architecture independent, we focus on an implementation for a specific FPGA in this section. We have chosen a simple architecture[1] with logic blocks containing one 4-LUT and a FF. In the interconnection network the wire segments only span one logic block and the channel width is fixed to 20.

FIR filtering is used for validation of our dual approach. In our experiments we use a generic 16 tap FIR filter with 8 bit coefficients and an 8 bit input, shown in figure 1. The generic multipliers are ripple carry array multipliers.

A generic netlist for this FIR filter was manually constructed. It contains 2704 logic blocks, 137 inputs (one for the clock, 8 for the input, and 16 times 8 for the coefficients) and 31 outputs. In the netlist every LUT was annotated with its truth table. Vplace was used with default settings for the offline placement of the generic design.

In order to test the online specialization and constant propagation, we randomly generated 100 different sets of 16 coefficients. The next subsections present results of experiments with these 100 FIR filters.

All experiments were executed on an Intel Core 2 processor running at 2.13 GHz with 2 GiB of memory.

5.1. Experiment 1: Compaction

Our compaction algorithm can produce a placement of equal quality compared to Vplace in a much shorter time. We use the critical path length after routing as the quality measure.

Vplace can be tuned to trade quality for execution time with the `inner_num` parameter. We ran Vplace for this parameter ranging from 0.2 to 10 (default Vplace setting) for all coefficient sets. The critical path length and the placement time were averaged over these FIR filters. The resulting data points are plotted in figure 5.

The compaction algorithm was also run for all 100 coefficient sets. The results were averaged over all FIR filters. The resulting data point is shown in figure 5. Our compaction algorithm on average takes 43 ms and produces FIR filters with an average critical path length of 119.3 ns.

The figure clearly shows that our compaction algorithm can produce a placement with equal quality to VPR in a shorter time. We measure an average speedup of 45.8 for an average critical path of 119.3 ns.

We can also see that our compaction algorithm produces FIR filters with a critical path length that is 36.1% longer on average compared to Vplace.

[1]A description of this architecture is provided with the VPR tool suite in `4lut_sanitized.arch`.

Note that Vplace cannot produce routable placements for our FIR filters if `inner_num` is smaller than 0.2.

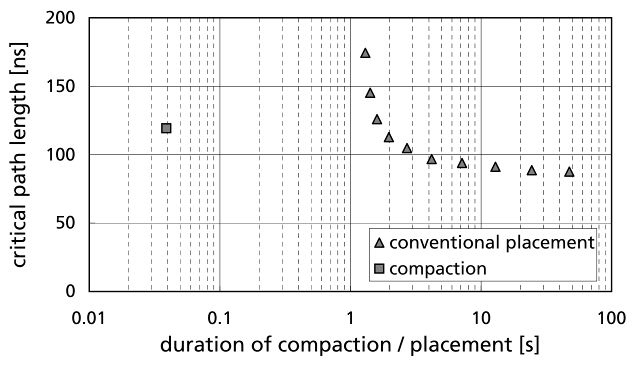

Fig. 5. Placement quality versus generation time

5.2. Experiment 2: Intrinsic hardware properties

In our second experiment we compare the intrinsic hardware properties of the hardware produced by three different design methods: our dual approach, a full online FPGA tool chain and a static implementation.

In the dual approach the hardware is generated online in three steps: constant propagation, compaction, and Vroute. The full online FPGA tool chain also produces the specialized hardware in three steps: Synthesis, Vplace and Vroute. Because we did not have a synthesis tool available that fits in the VPR tool chain we replaced it with the constant propagator. No online hardware generation is done by the static implementation. To achieve the same functionality, the static hardware implements a generic FIR filter.

Table 1 shows the average area and critical path length for these three design methods. We see that on average the static implementation uses 81.4% more logic blocks than the two dynamic implementations. On average the hardware produced by the dual approach is 5.3% faster than the static implementation and 36.1% slower than the hardware produced by the online VPR tool chain.

Table 1. Properties of the generated hardware

	Dual Approach	VPR	Static
Area (logic blocks)	1491	1491	2704
Critical path (ns)	119.3	87.6	126.0

5.3. Experiment 3: Hardware generation time

In the third experiment we compare the generation times of the dynamic design methods described in section 5.2

Table 2 gives the average total hardware generation time and its decomposition. We see that the dual approach is on average 12 times faster than the full VPR tool chain and that the bulk of the generation time of this approach is routing time. Incremental routing could reduce this routing time and make our method even faster.

Table 2. Duration of the hardware generation process

	Dual Approach	VPR
Constant propagation (ms)	128	—
Compact / Place (ms)	43	47557
Route (ms)	3889	2275
Total generation time (ms)	3932	49832

Note that no measurements are filled out for the synthesis time because we did not have a synthesis tool available that fits in the VPR tool chain. Adding the synthesis time will increase the total time needed for the online VPR approach, thus enhancing the performance advantage of our dual approach.

5.4. Experiment 4: Functional Density

Finally we compare the functional density of the three design methods described in section 5.2. We calculated the functional density as shown in section 2 for several values of n and averaged the resulting functional densities over all 100 FIR filters. The result is shown in figure 6. In our FIR filter example n denotes the number of input samples that are filtered in between coefficient changes.

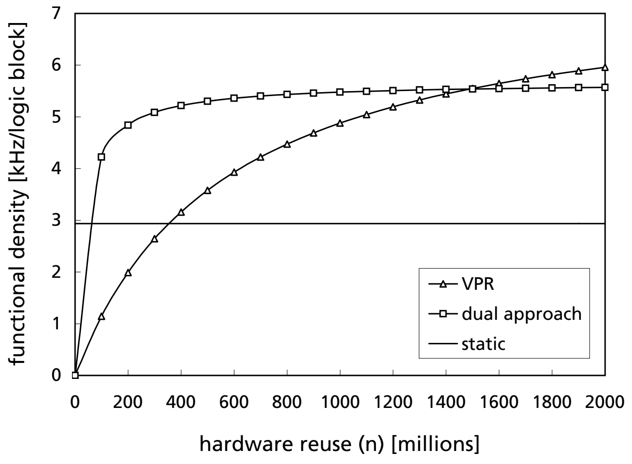

Fig. 6. Functional density versus hardware reuse

For ($n >$ 40 million), our dual approach outperforms the static approach because of the improved area and timing with a relatively low overhead. The VPR flow has an even better performance — in area and timing — but with a much larger overhead. Therefore only for very large reuse values ($n >$ 1.5 billion) its functional density is higher than the one for our approach.

The range where our dual approach is beneficial can be split up into two subranges. In the range (40 million $<$ $n <$ 360 million) dynamic hardware generation is not beneficial without our new approach. In this range our specialization technique enables dynamic hardware generation for a new range of applications. In the second range (360 million $< n <$ 1.5 billion) the functional density for online hardware generation is improved by our method.

6. RELATED WORK

The exploitation of the reconfigurability of FPGAs to improve performance has been studied intensively in the past decade.

Run-time reconfiguration of the FPGA by switching between several statically generated bitstreams has lead to interesting results in the field of neural networks, template matching, DNA sequencing and many others [1, 3]. Because the hardware is generated offline, in contrast to our approach, only the reconfiguration incurs an overhead. Of course, offline generation provides less optimization opportunities.

Other related work was done in the field of hardware generation at run-time. For applications with a quasi-static behavior a full hardware generation with a conventional tool flow was successfully used to improve overall performance. Examples are key-specific DES [4], the subgraph isomorphism problem [5], boolean satisfiability [6] and many others. This approach cannot be extended for applications with a more dynamic behavior — the applications focused on in our work — because the conventional hardware generation process is far too slow. In the WARP processor [7] frequently used code is dynamically moved from processor to FPGA. The hardware generation is done by lean versions of the conventional FPGA tool chain.

Several authors proposed run-time hardware specialization on the netlist level based on partial evaluation [4, 8]. McKay et al [9] also reuse placement and routing information but do not compact the sparse placement obtained after constant propagation.

7. CONCLUSIONS AND FUTURE WORK

In this paper we proposed a dual approach for run-time hardware generation. Our approach combines the generation of generic hardware at compile-time with run-time specialization. This allows us to use fast tools at run-time while maintaining the hardware quality. Hence the hardware generation overhead can be reduced extensively. We described two of those tools: the constant propagator and the compactor. The incremental router in our approach is not yet implemented.

We validated our approach on FIR filters with changing coefficients. A static generic implementation is compared to run-time hardware generation, both with a full tool chain at run-time (VPR) and our faster dual approach. We have shown that run-time hardware generation can effectively improve functional density if the generated hardware is sufficiently reused. By reducing the hardware generation overhead our dual approach makes run-time hardware generation profitable for a new range of applications.

Although the tools we have implemented are sufficient for proving the concept of our dual approach many improvements can be made. The next step in our research is therefore the implementation of an incremental router. This could substantially improve functional density because the bulk of the run-time generation time is due to routing. We also plan an in depth study of constant propagators and compaction algorithms.

8. REFERENCES

[1] M. J. Wirthlin and B. L. Hutchings, "Improving functional density using run-time circuit reconfiguration," *IEEE Trans. VLSI Syst.*, vol. 6, no. 2, pp. 247–256, 1998.

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

[3] J. Villasenor, B. Schoner, K.-N. Chia, C. Zapata, H. J. Kim, C. Jones, S. Lansing, and B. Mangione-Smith, "Configurable computing solutions for automatic target recognition," in *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 1996.

[4] J. Leonard and W. H. Mangione-Smith, "A case study of partially evaluated hardware circuits: Key-specific DES," in *Proc. International Workshop on Field-Programmable Logic and Applications (FPL)*, 1997, pp. 151–160.

[5] S. Ichikawa and S. Yamamoto, "Data dependent circuit for subgraph isomorphism problem," in *Proc. International Conference on Field-Programmable Logic and Applications (FPL)*, 2002, pp. 1068–1071.

[6] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Accelerating boolean satisfiability with configurable hardware," in *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 1998, p. 186.

[7] R. Lysecky, G. Stitt, and F. Vahid, "Warp processors," *Transactions on Design Automation of Electronic Systems*, vol. 11, no. 3, pp. 659–681, July 2006.

[8] S. Singh, J. Hogg, and D. McAuley, "Expressing dynamic reconfiguration by partial evaluation," in *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 1996.

[9] N. McKay and S. Singh, "Dynamic specialisation of XC6200 FPGAs by partial evaluation," *Lecture Notes in Computer Science*, vol. 1482, p. 298, 1998.