CONFIGURATION MERGING FOR ADAPTIVE COMPUTER APPLICATIONS

Nico Kasprzyk

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

Jan C. van der Veen

Tech. Univ. Braunschweig
Dept. for Mathematical
Optimization
Braunschweig, Germany
j.van-der-veen@tu-bs.de

Andreas Koch

Tech. Univ. Darmstadt
Embedded Systems and
Applications Group (ESA)
Darmstadt, Germany
a.koch@acm.org

ABSTRACT

We present experimental evidence that multiple compute-units, compiled from sequential high-level language input programs, can be merged into a reduced number of configurations for a reconfigurable fabric (such as a modern FPGA), thus significantly reducing the reconfiguration overhead. For cases requiring multiple configurations, both heuristical and exact algorithms to solve the configuration merging problem are described.

1. INTRODUCTION

Significant effort has been invested in the research on and the development of compilers for adaptive (reconfigurable) computing system (ACS) [1] [2] [3] [4] [5]. These tools aim to allow the programming of both the fixed-structure (software-programmable processor) and the variable-structure compute unit (reconfigurable device) of an ACS from a conventional high-level language, instead of the hardware-centric descriptions (HDL or even schematics) that are commonly used today.

The more powerful of these ACS compilers also automatically partition the input application between hardware and software execution [1] [2] [3]. In this process, computation-intense *kernels* are extracted from the description and mapped to dedicated compute units (CU) implemented on the reconfigurable fabric (rF).

Orthogonal to this flow is the manner, in which the individual CUs are scheduled onto the rF. The solution to this problem is highly dependent on the nature of the rF and affected by characteristics such as reconfiguration time, partial reconfigurability, the depth of a reconfiguration cache, and the available rF area.

For example, the Garp-CC compiler, targeting the Garp ACS [1] scheduled a single CU at-a-time onto the rF. However, due to the presence of a configuration cache, the rF could switch very quickly (order of tens of clock cycles) between a limited number of CUs. Other approaches target devices that are quickly partially reconfigurable by examining online scheduling and placement techniques [6] [7]. Here, multiple CUs are scheduled dynamically onto the rF, possibly even observing quality-of-service constraints.

However, most commercially available devices have neither a configuration cache nor rapid reconfigurability. Partial reconfigurability, if it is offered at all, is often still slow relative to the system clock and encumbered by additional limitations (lack of tool support, imposes placement constraints, etc.). Thus, our work concentrates on optimizing the complete reconfiguration approach.

In the next sections we will discuss both previous work as well as our current research to exploit high-level language programming of an ACS even under these adverse conditions (no on-chip configuration cache, only full reconfigurations supported).

2. NOVEL RECONFIGURATION STRATEGY

The compiler COMRADE [8], currently under development, is a spiritual successor to Nimble [2]. Like its predecessor, it can also target commercially available FPGAs as rF.

Nimble dynamically scheduled reconfigurations at run-time. The decision whether to realize a kernel as CU, or keep it in software, was made at compile time. Each CU was implemented as a separate configuration bitstream. However, often, a single reconfiguration obliterated the entire speed-up achievable by executing a kernel as CU on the rF.

In order to alleviate this excessive configuration overhead, a different reconfiguration strategy is employed in COMRADE: Practical experiments, both in COMRADE and in Nimble, determined that the largest CUs practically extractable from sequential programs written in the C programming language have around a hundred operators.

Assuming that no bit-width reduction occurs, most of these are 32b wide (C's native integer data type). In the common fine-grained FPGA architectures (based on 4-input look-up tables plus optional flip-flops), many of these operators will require just 32 cells (possibly extended with dedicated carry-logic for fast arithmetic). Thus, most CUs require fewer than 3000 cells on the rF.

Since even medium-sized low-cost FPGAs currently have more than 17,000 cells available, the traditional of paradigm of "one CU, one configuration", followed since Garp-CC, is extremely wasteful today.

In our new strategy, each configuration can now hold *multiple* CUs. The effect is that of a configuration cache: The rF can now quickly switch between all of the CUs packed into the current configuration. The total silicon area efficiency of this approach is of course less than that of a dedicated configuration cache (which only has to replicate configuration SRAM cells). However, an advantage lies in the variable granularity of the "cache entries": When using the rF area in this manner, the number of cacheable CUs is inversely proportional to their size. Thus, a large number of small CUs can be supported as well as a small number of large CUs. This trade-off is not possible with most of the proposed configuration caches, which support only a fixed number of cached full-size configurations.

The back-end tools of the compile flow have to be enhanced to accommodate this new use of the rF resources. In our case, this enhancement consists of a specialized floorplanning tool that places all of the individual CUs in a regular fashion on the rF, and also synthesizes/places multiplexers and their associated decoders on-the-fly to connect the individual CUs to the shared infrastructure on the rF (memory interfaces, processor interface and control logic, etc.).

The focus of the following discussion, however, are the algorithms to determine which CUs to pack into which configurations, replicating CUs as necessary to reduce configuration times even further. We will present both a heuristic for quickly generating estimates as well as an exact method for determining optimal solutions.

3. NOTATION

For the following discussion, we will use these notational conventions:

- $P = \{p_1, p_2, \ldots, p_n\}$ is a set of n CUs.
- $s : P \rightarrow \mathbb{N}$ is the size of the CU in rF resources.
- K is the total number of resources available on the rF.
- $\mathscr{C} = \{C \in 2^P : \sum_{p \in C} s(p) \leq K\}$ is the set of all feasible configurations
- $\mathscr{C}' \subset \mathscr{C}$ is the set of *sequence-maximal* feasible configurations (see Section 5.2).
- $T = (p_i, p_j, \ldots, p_k) \in P^m$ describes the dynamic execution order of the CUs as a sequence of m steps.
- $R = (\mathscr{C}_i, \mathscr{C}_j, \ldots, \mathscr{C}_k) \in \mathscr{C}^m$ is the reconfiguration sequence. $R_i \in \mathscr{C}$ is the configuration that must be loaded onto the rF before the CU T_i can be started.
- $\hat{\mathscr{C}} = \{C : C \in R\}$ is the set of unique configurations used to realize R.
- A reconfiguration occurs each time $\mathscr{C}_i \neq \mathscr{C}_{i+1}$, for $1 \leq i < m$.
- $r_i \in \mathbb{N}$ is the number of reconfigurations that have to be done if execution starts with the trace step T_i. Thus, r_1 indicates the total number of reconfigurations for the entire trace T.

4. HEURISTICS

4.1. Core data structures

The heuristics described here do not require *dynamic* kernel execution sequence data (such as Nimble's LEP) on the input program when assembling CUs into configurations. They are thus independent of the complexity (trace length m) of the application's run-time behavior. Instead, our main data structure, the Global CU Sequence Graph (GCSG), is based on a *static* view of the program. The GCSG models all *possible* execution sequences of CUs, globally across the entire program (crossing procedure call boundaries in the process). Thus, it is more detailed than the hierarchy-focussed view in Nimble's representation of the static program, the LPHG.

The nodes G_V of a GCSG G are the CUs P (which in COMRADE consist only of loops), while the directed edges G_E indicate execution (control) flow between them. The GCSG is built hierarchically, flattening the program structure in the process. Thus, edges may initially be labelled with a procedure call, which is resolved during a later pass. An edge is inserted between two nodes if, at any time, execution passes from the origin to the destination CUs in the static program structure. Cycles may occur when a procedure is called from multiple locations in the input program.

Figure 1 illustrates the transformation from input program (a) via the intermediate hierarchical GCSG (b) to the final fully flattened GCSG (c). The nodes in this final GCSG are then annotated with their resource requirements on the rF and their *execution factor* $e(l)$. The latter is the relative execution frequency as determined by dynamic profiling, defined as $e(l) = c(h(l))/\max_{i \in I} c(i)$.

where c is the execution count (as determined by dynamic profiling) of instruction i, I are the instructions of the entire program, and $h(l)$ gives the header (entry) instruction of CU (loop) l.

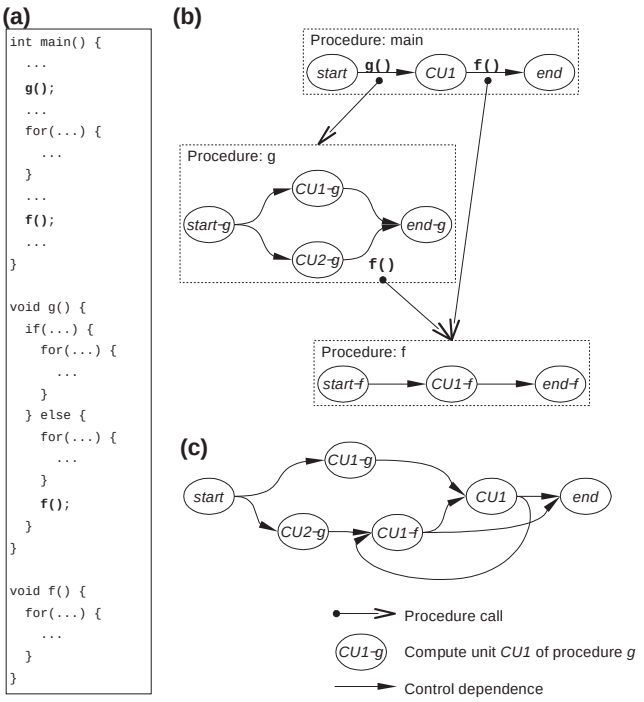

Fig. 1. Building the Global CU Sequence Graph from input program

MERGECUSINTOCONFIGS$(G,\hat{\mathscr{C}})$

1 $\hat{\mathscr{C}} \leftarrow \emptyset$
2 **for** $C \in \mathscr{C} : \forall p \in C$ are nodes on a natural loop in G **do**
3 $\quad \hat{\mathscr{C}} \leftarrow \hat{\mathscr{C}} \cup \{C\}$
4 $\hat{\mathscr{C}} \leftarrow \hat{\mathscr{C}} \backslash \{C \in \hat{\mathscr{C}} : C \subset C' \wedge C' \in \hat{\mathscr{C}}\}$
5 **for** $C \in \hat{\mathscr{C}}$ **do**
6 $\quad$ **while** $\exists p \in u(V_G, \hat{\mathscr{C}}) : (q,p) \in E_G \wedge q \in C$
$\qquad \wedge s(C) + s(p) \leq K$
$\qquad \wedge \forall p' \in u(V_G, \hat{\mathscr{C}}) : e(p) \geq e(p')$ **do**
7 $\quad\quad C \leftarrow C \cup \{p\}$
8 **for** $p \in u(V_G, \hat{\mathscr{C}}) : (q,p) \in E_G \wedge q \in \bigcup_{C \in \hat{\mathscr{C}}} C$ **do**
9 $\quad$ GROWCONFIGAROUNDCU$(p, G, \hat{\mathscr{C}})$
10 **for** $p \in u(V_G, \hat{\mathscr{C}})$ **do**
11 $\quad$ GROWCONFIGAROUNDCU$(p, G, \hat{\mathscr{C}})$

Fig. 2. Configuration merging heuristic

4.2. Algorithm

Fundamentally, a clustering algorithm (shown in Figure 2) is used in the heuristic. For brevity, we define the set of CUs from V_G not yet merged into a configuration in $\hat{\mathscr{C}}$ as $u(V_G, \hat{\mathscr{C}}) = V_G \backslash \bigcup_{C \in \hat{\mathscr{C}}} C$.

The operation proceeds in three stages: First, maximal configurations are built which cover all *natural loops* [9] in the control flow (Lines 1 to 4).

By using this dominator-based natural loop concept as our detection criterion, we are looking for correctly nested loops that actually have a single header followed by a body. If we were just determining cycles in the graph, we could stumble into cyclic sub-graphs that do not have the proper loop structure just described.

Then, CUs not part of a natural loop, but adjacent to an already covered one, get included into the loop's existing configuration in order of the highest execution factor, as long as they fit on the device (Lines 5 to 7). At this point, the loop-based configurations can no longer be grown, and new configurations are being created. This is done in two substeps: In Lines 8 to 9, the process is started at CUs that are executed immediately after one of the already-processed loops. For pathological cases (unrelated loops exceeding the device size), a last pass collects all CUs that are not adjacent to loops in the control flow (Lines 10 to 11).

The manner in which new configurations are created is shown in Figure 3: They are grown around a seed CU, with the growth proceeding alternatingly between the CUs that are following just the *seed CU* in the control flow, and those CUs following the *entire configuration-under-construction* in the control flow. In both cases, the candidate CU with the highest execution factor, but still fitting on the rF, is added first. Figure 4 shows a sample GCSG and the result of applying the heuristics for a device size of $K = 400$ to assemble the $|P| = 7$ CUs into $|\hat{\mathscr{C}}| = 3$ different configurations.

5. EXACT SOLUTION

To verify the quality of these heuristics, we also developed a number of methods to calculate the optimal solution to the problem. After initial attempts using ILP-based methods, we formulated a solution as a dynamic program. While the heuristics presented in the last section are still faster in some cases, the run-time of the exact method is still sufficiently

GROWCONFIGAROUNDCU$(p, G, \hat{\mathscr{C}})$

1 $C \leftarrow \emptyset$
2 **repeat**
3 $\quad C' \leftarrow C$
4 $\quad C = C \cup \{q \in u(V_G, \hat{\mathscr{C}}) : (p,q) \in E_G$
$\qquad \wedge s(C) + s(q) \leq K$
$\qquad \wedge \forall q' \in u(V_G, \hat{\mathscr{C}}) : e(q) \geq e(q')\}$
5 $\quad C = C \cup \{q' \in u(V_G, \hat{\mathscr{C}}) : (q,q') \in E_G \wedge q \in C$
$\qquad \wedge s(C) + s(q') \leq K$
$\qquad \wedge \forall q'' \in u(V_G, \hat{\mathscr{C}}) : e(q') \geq e(q'')\}$
6 **until** $C = C'$
7 $\hat{\mathscr{C}} = \hat{\mathscr{C}} \cup \{C\}$

Fig. 3. Growing a new configuration around a seed CU

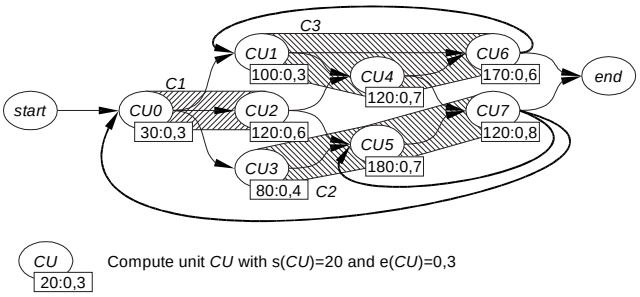

Fig. 4. Example for heuristical configuration merging

MINIMALNUMBEROFRECONFIGURATIONS$(T,\mathscr{C})$

```
1  for i ← m downto 1 do
2          r_i ← ∞
3          for C ∈ 𝒞 : T_i ∈ C do
4                  j ← i
5                  repeat
6                          j ← j + 1
7                    until j > m ∨ T_j ∉ C
8                  if j > m then
9                          R_i ← C
10                         r_i ← 1
11                 elseif r_i > r_j + 1 then
12                         R_i ← C
13                         r_i ← r_j + 1
```

Fig. 5. Calculating the minimal number of reconfigurations

short to occur within the hardware/software partitioning step of the compile flow.

5.1. Minimizing the number of reconfigurations

For a given trace T of length m, we want to determine a sequence $R = (C_i, C_j, \ldots, C_k) \in \mathscr{C}^m$ of feasible configurations that realizes T. The optimal reconfiguration sequence R will have a minimal number of reconfigurations across the entire trace T. Thus, r_1 is to be minimized. The solution to this problem is formulated as a dynamic program [11], shown in Figure 5.

The correctness and optimality of this algorithm can be proven by induction.

1. Consider a trace of length 1. Any configuration containing T_1 is an optimal configuration sequence.
2. Assume correctness and optimality for a trace of length $t-1$.
3. Consider a trace of length t that has been constructed by prepending a single CU T_i to an existing trace of length $t-1$. To compute an optimal configuration for this new trace, it suffices to examine all configurations C containing T_i. Such a configuration C possibly also contains one or more of the CUs that follow T_i in the trace. Then let j be the index of the first trace element not in C. Two cases have to be treated:
   (a) $\forall j \in \{2,3,...,t\} : T_j \in C \Rightarrow r_i = 1$. Here, C contains not only the new T_i, but also all other CUs in the trace. Thus, the number of reconfigurations from the new CU at position i to the end of the trace remains 1.
   (b) $j \leq t \vee r_i > r_j + 1 \Rightarrow r_i = r_j + 1$. Now C contains the CUs up to, but excluding T_j. Thus, a reconfiguration has to occur at this break. The total number of reconfigurations from i to the end of the trace is thus one larger than the reconfigurations from j to the end of the trace. Since $j \leq t-1$, this is optimal.

SEQUENCEMAXIMALCONFIGURATIONS$(T,\mathscr{C}')$

```
1  𝒞' ← ∅
2  for i ← 1 to m do
3          C ← ∅
4          j ← i
5          while T_j ∈ C ∨ Σ_{p∈C} s(p) + s(T_j) ≤ K do
6                  C ← C ∪ {T_j}
7                  j ← j + 1
8
9          𝒞' ← 𝒞' ∪ {C}
```

Fig. 6. Building $\mathscr{C}'$ from all sequence-maximal feasible subsets

The time complexity of the algorithm is $O(m^2|\mathscr{C}|)$, since the membership tests in Lines 3 and 7 of Figure 5 can be performed in constant time.

5.2. Restricting the search space

So far, we have used as search space the set of *all* feasible configurations $\mathscr{C} = \{C \in 2^P : \sum_{p\in C} s(p) \leq K\}$. Unfortunately, this set is far too large and can only be generated in exponential time. While the size of the set can be reduced trivially by removing all non-maximal sets (sets that are subsets of other sets), the generation time remains exponential.

However, since we are specifically optimizing for the minimum number of reconfigurations, we can restrict the search space to just the *sequence-maximal* feasible configuration subsets, named $\mathscr{C}'$ (Figure 6).

Clearly, the number of sequence-maximal feasible subsets is bounded by m, with the running time of the algorithm being polynomial in m.

This restricted search space does not degrade the quality of the solutions as long as we are only optimizing for the minimum number of reconfigurations r_1 (our current aim). If

we were also aiming to minimize the number of *different* configurations $|\hat{\mathscr{C}}|$, this would no longer hold true, though.

Operating on $\mathscr{C}'$ instead of $\mathscr{C}$, the running time of the algorithm in Figure 5 is now bounded by $O(m^3)$.

Lack of space here precludes the discussion of additional optimization techniques, such as compressing the execution traces and splitting the trace at suitable breakpoints, which can reduce the run-time even further. For consistency, the effect of these advanced techniques will not be shown in the experimental evaluation below.

6. EXPERIMENTAL RESULTS

Table 1 presents the results when applying both the heuristic and the exact solution to the configuration merging problem. We applied the algorithms to a number of real-world applications and one synthetic benchmark. The CUs were generated by the COMRADE compiler from C-language input programs, targeting as rF a Xilinx Virtex-like architecture for $K = 5120$, and an XC4000-like architecture for other values of K. K is given as CLBs, where a CLB consists of two logic cells. Note that COMRADE attempts to exploit larger devices by building larger CUs (holding more paths through the kernels), but currently does *not* optimize the reconfiguration schedule (the subject of this work).

For comparison, we have also shown the absolute minimum number of configurations to hold the CUs in P, regardless of the number of reconfigurations required. This was optimally calculated using an ILP-based bin packing formulation [10]. All run-times are given in seconds, rounded to 1/100s, on a 900 MHz UltraSPARC-III+ CPU.

The effectiveness of the methods presented here can be determined by comparing the length of the execution trace m, which would be the number of reconfigurations in the traditional approach, with the reduced numbers in the columns 'Min r_1'. In many cases, the results are obvious (all CUs could fit into a single rF configuration, thus requiring only a single reconfiguration for the application). However, even in more complex cases (CUs exceed rF area), significant savings can be realized. E.g., for $K = 1920$ in the Versatility application, instead of 5381 reconfigurations of 8 different configurations, now only four reconfigurations of four configurations are optimally required. While the heuristic result requires an additional reconfiguration, this calculation required only a fraction of the computation time (less than 1/100s) of that for the optimal solution. Note that the heuristic does *not* duplicate CUs, thus $s(\hat{\mathscr{C}}) = s(P)$.

Furthermore, it becomes clear that the current COMRADE approach of greedily moving ever larger parts of the input program to the rF needs to be guided by the configuration merging/scheduling techniques introduced here. Since they are sufficiently fast (especially the heuristic), they can be employed in the hardware/software partitioning step to trade-off somewhat smaller CUs (e.g., losing parallelism) with significantly reduced reconfiguration overhead.

7. CONCLUSIONS

We have demonstrated a novel approach to use the increasing number of resources on current rFs to significantly reduce reconfiguration times by merging multiple (possibly duplicated) CUs into larger configurations. To this end, we have described both a fast heuristic independent of the (possibly very long) run-time execution flow of the input program, as well as an optimal solution that does take the flow into account while also having polynomial run-time. Especially the heuristic is well applicable for use within the hardware/software partitioning step of the compile flow, and could significantly reduce reconfiguration times by guiding the kernel selection/synthesis.

8. REFERENCES

[1] Callahan T., Hauser J., Wawrzynek J., "The Garp Architecture and C Compiler", *IEEE Computer*, Vol. 33, No. 4, April 2000

[2] Li. Y., Callahan T., Darnell E., Harr R., et al., "Hardware-Software Co-Design of Embedded Reconfigurable Architectures", *Proc. Design Automation Conference (DAC)*, 2000

[3] Kasprzyk N., Koch A., "Verbesserte Hardware-Software-Partitionierung für Adaptive Computer", *Proc. Conf. on Architecture of Computing Systems (ARCS)*, Augsburg (Germany), March 2004

[4] Banerjee P., Shenoy N., Choudhary A. et al., " MATLAB Compiler for Distributed Heterogeneous Reconfigurable Computing Systems", *Proc. Int. Symp. on FPGA Custom Computing Machines (FCCM)*, Napa Valley (CA, USA), April 2000

[5] Goering R., "LavaLogic to field Java-based synthesis tool", *EE Times*, March 6, 2000

[6] Walder H., Platzner M., "A Runtime Environment for Reconfigurable Operating Systems", *Proc. 14th Intl. Conf. on Field-Programmable Logic and Applications (FPL)*, Antwerp (BE), August 2004

[7] Ahmadinia A., Bobda C., Fekete S.P. et al., "Optimal routing-conscious dynamic placement for reconfigurable computing", *Proc. 14th Intl. Conf. on Field-Programmable Logic and Applications (FPL)*, Antwerp (BE), August 200

[8] Kasprzyk N., "COMRADE – Ein Hochsprachen-

				Binpack	Heuristic			Optimal			
K	$\|P\|$	$s(P)$	m	Min $\|\hat{\mathscr{C}}\|$	Min r_1	$\|\hat{\mathscr{C}}\|$	t	Min r_1	$\|\hat{\mathscr{C}}\|$	$s(\hat{\mathscr{C}})$	t
Capacity [12] – Parametrized Huffman table generator											
480	7	224	42	1	1	1	0.02s	1	1	224	0s
853	7	1120	42	2	2	2	0.03s	2	2	1184	0s
1080	7	1120	42	2	4	2	0.01s	2	2	1664	0s
1333	7	2016	42	2	2	2	0.01s	2	2	2048	0s
1613	7	2016	42	2	2	2	0.01s	2	2	2592	0s
1920	7	2016	42	2	2	2	0.01s	2	2	2592	0s
5120	7	1473	42	1	1	1	0.03s	1	1	1473	0s
DES – Encryption											
480	7	448	962	1	1	1	0s	1	1	448	0.12s
853	7	448	962	1	1	1	0.02s	1	1	448	0.12s
1080	7	1696	962	2	53	2	0.02s	52	2	2016	0s
1333	7	1696	962	2	53	2	0.01s	52	2	2016	0.11s
1613	7	1696	962	2	53	2	0.02s	39	4	5168	0.01s
1920	7	1696	962	1	1	1	0s	1	1	1696	0.04s
5120	7	8688	962	2	53	2	0.01s	53	5	22344	0.01s
ManyFORs – Synthetic benchmark, many nested for loops											
480	32	2752	12668	6	3788	8	0.14s	3747	15	6384	0.21s
853	32	10816	12668	19	7588	21	0.30s	7467	22	12528	0.05s
1080	32	10816	12668	11	5403	13	0.15s	4647	25	23376	0.08s
1333	32	11632	12668	10	4490	12	0.12s	4250	22	22416	0.08s
1613	32	11632	12668	8	3456	9	0.17s	3197	27	38656	0.12s
1920	32	11632	12668	7	2981	8	0.06s	2296	25	45152	0.16s
5120	32	8140	12668	2	4	2	0.09s	2	2	10161	94.23s
Versatility [12] – Wavelet-based image compression											
480	6	432	5379	1	1	1	0.00s	1	1	432	1.74s
853	6	880	5379	2	2	2	0.00s	2	2	1456	2.73s
1080	6	880	5379	1	1	1	0.01s	1	1	880	1.77s
1333	6	2288	5379	2	2	2	0.00s	2	2	2320	2.58s
1613	8	4304	5381	4	5	4	0.01s	4	4	4880	2.58s
1920	8	4304	5381	3	5	4	0.00s	4	4	4880	2.59s
5120	8	11171	5381	3	3586	3	0.01s	2689	4	17813	0.01s
Pegwit – Elliptic-curve encryption											
480	11	1104	130760	3	84947	4	0.00s	61492	10	4264	0.96s
853	11	1472	130742	2	5951	2	0.00s	1266	7	5560	5.48s
1080	11	2184	133184	3	5077	3	0.01s	1125	9	8896	10.23s
1333	11	3064	129924	3	80634	3	0.00s	60475	4	4976	0.35s
1613	11	4056	134004	3	3900	4	0.01s	1301	6	8480	2.14s
1920	11	4056	132384	3	1286	3	0.00s	1285	8	13808	7.69s
5120	11	8706	131980	2	4733	2	0.01s	1919	9	43296	8.15s

Table 1. Experimental results

Compiler für Adaptive Computersysteme", *doctoral thesis*, Tech. Univ. Braunschweig (D), March 2005

[9] Appel A.W., "Modern Compiler Implementation in Java", p. 419, *Cambridge University Press*, 1999

[10] Martello S., Toth P., "Knapsack Problems", *John Wiley and Sons*, 1990

[11] Cormen H., Leiserson C.E., Rivest D.L., "Introduction to Algorithms", *McGraw-Hill Book Company*, 2001

[12] Kumar, S., Pires, L., Ponnuswamy S., et al., "A Benchmark Suite for Evaluating Configurable Computing Systems - Status, Reflections, and Future Directions", *Proc. Eighth International Symposium on Field-Programmable Gate Arrays (FPGA)*, Monterey (USA), 2000