Feasibility of High Level Compiler Optimizations in Online Synthesis

Lukas Johannes Jung and Christian Hochberger
Department for Electrical Engineering and Information Technology
Computer Systems Group, TU Darmstadt
Email: {jung, hochberger}@rs.tu-darmstadt.de

***Abstract*—High-level synthesis approaches are currently very popular. They use specifications on rather high abstraction levels like C or SystemC to describe the required system functionality and automatically generate a hardware implementation for this specification. Shifting this process to the runtime of the system leads to so called online synthesis. The advantage of this approach is that only heavily used parts for the current run of the software need to be synthesized and also profiling information of the current run can be used to optimize the implementation, which enables an adaptive system behavior. Additionally, synthesis will only use the amount of currently available resources. Obviously, in this case the synthesis algorithms must run sufficiently fast such that the hardware implementation is created quickly enough. In this contribution we discuss the feasibility and the efficiency of different high-level optimizations which are usually performed by sophisticated compilers and synthesis systems.**

I. Introduction

Chip manufacturing cost increases exponentially with the feature size of the employed technologies. Thus, in the future individual chips can only be manufactured for high quantity chips. One way to deal with this limitation is the usage of reconfigurable chips in the form of field programmable gate arrays (FPGAs) or coarse grained reconfigurable arrays (CGRAs). Both technologies offer the option to change their functionality after production.

Currently, many designs for these chips are created through high-level synthesis approaches. In this case a rather abstract description of the required functionality is given by high-level languages like C or SystemC. This specification is then automatically[1] transformed into a HW implementation. While this is a viable solution, it requires massive rewriting and adaption of existing applications. Furthermore, it must be executed for each and every variant of the target platform.

We can also shift the synthesis process towards the runtime of the system, which is often called *online synthesis*. Online synthesis has advantages and drawbacks compared to offline approaches.

Among the advantages, particularly the following should be considered:

- No modification of the application is required. All applications can benefit from the synthesis directly.
- A current profile of the application is available. The profile is specialized with respect to the input data of the application and also with respect to the overall system situation (what other applications are running, how many resources are available).
- The synthesis process can be tailored towards the available resources *in the target*. This means that different target variants (#LUTs in an FPGA or #PEs in a CGRA) can be supported without changing the application.
- Also, the synthesis process can respect the local usage situation, e.g. sharing the reconfigurable fabric with other applications to maximize the system performance.

The major drawbacks of online synthesis are the requirements: It must be sufficiently quick enough (otherwise the approach may not pay off in many situations) and the synthesis must be carried out on a resource constrained device, which limits the amount of code that can be invested. Also, a user intervention is not possible in the case of online synthesis.

In the light of this discussion, the central challenge of online synthesis is to produce high quality results with a small amount of runtime of the synthesis algorithm.

In this contribution we discuss which compiler/synthesis optimizations can be carried out in the target system and what effect they have on the synthesized HW. As a basis for these experiments we use the AMIDAR processor architecture which is particularly created to support runtime synthesis. It identifies and transforms candidate instruction sequences fully automatically into HW configurations of our reconfigurable fabric.

The rest of this paper is structured as follows: The next section explains the AMIDAR processor concept. In section three we discuss the optimizations considered in this paper. Following, we evaluate runtime and effect of all combinations of these optimizations. Then we show related work, which leads to the conclusion and an outlook onto future work.

II. The Processing model

The processing model used in this work is based on the AMIDAR model developed by Gatzka et al. [1].

A. General Model

Execution of instructions in AMIDAR processors differs from other execution schemes. Neither microprogramming nor pipelining are used to execute instructions. Instead, instructions

[1]This is the ideal case. In many systems, the user still has to help the system in this process with fundamental design and application knowledge.

978-1-4673-9406-2/15/$31.00 ©2015 IEEE

are broken down into a set of tokens which are distributed to a set of functional units (FU). These tokens carry the information about the type of operation that shall be executed, the version information of the input data that shall be processed (called *tag*), and the destination of the result.

Tokens which do not require input data can be processed immediately. Otherwise, FUs wait for input data that has the appropriate tag information. Once the right data is available, the operation starts. Upon completion of the operation, the result is sent to the FU that was denoted in the token as destination.

In this version AMIDAR processes Java bytecode directly. A detailed explanation of the model, its application to instruction sequence execution and its specific features can be found in the original work.

B. Adaptivity in the AMIDAR Model

The AMIDAR model exposes different types of adaptivity. Firstly, the communication structure can be adapted to minimize the bus conflicts that occur during the data transports between the FUs. Secondly, the characteristics of the FUs can be changed to optimally suit the needs of the running application. FUs can either be latency optimized or throughput optimized.

Finally, one can augment the processor with some specialized FUs that implement often used code sequences in an optimized way. The identification of candidate sequences is done automatically by a profiler implemented in hardware [2]. It counts instructions of loop bodies on all loop nesting levels and triggers a software thread when a particular sequence exceeds a predefined threshold of the execution time. This thread can then optimize the execution of the sequence. The optimization can be done in two different ways:

- *Synthilation* [2] [3] maps the sequence to be accelerated to the existing FUs resulting in an optimized tokenset
- *Synthesis* maps the sequence to be accelerated to a CGRA resulting in an specialized FU

The result of the mapping can be stored and following occurrences of the sequence can make use of the optimization result.

Both optimization methods can lead to substantial speedup, but in this work we make only use of *Synthesis* in order to evaluate the feasibility of high level compiler optimizations in online synthesis. We extended our high level synthesis algorithms described in [4] in order to support the following optimizations.

- Speculative method inlining
- Partial loop unrolling
- Common subexpression elimination

A detailed description can be found in Section III.

[2]The term synthilation was coined for this particular mixture of methods between hardware synthesis and classical compilation.

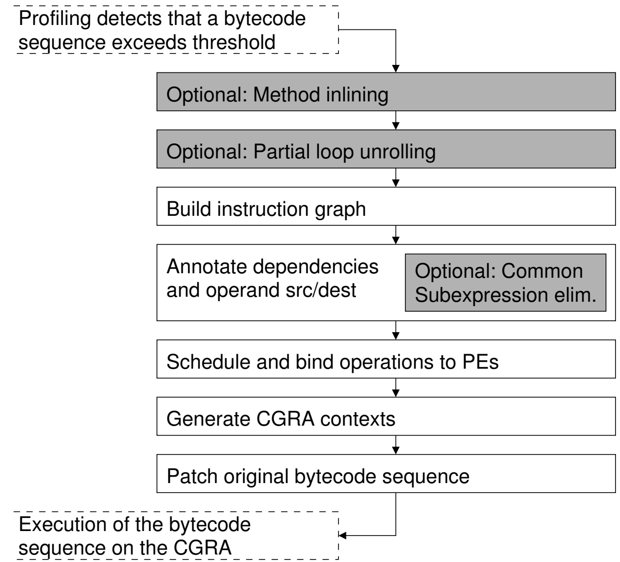

Fig. 1. Processing steps for the synthesis of an CGRA accelerator

C. Synthesis in AMIDAR

When the hardware profiler mentioned in the previous paragraph identifies a candidate sequence, the synthesis process shown in Fig. 1 is started. The first two steps in synthesis are optional. They perform method inlining and loop unrolling. From the resulting bytecode sequence an instruction graph is created. Afterwards, this graph is annotated with dependency information and information regarding sources and destinations of operands. This step can include common subexpression elemination. The annotated graph is then subject to scheduling and binding.

D. CGRA Model

The CGRA model used in this work assumes fully connected processing elements (PE). Local variables are stored in a central register file. All PEs can access the register file directly.

In contrast to previous AMIDAR versions the CGRA can now access arrays and objects on the heap via DMA. Thus, now it is possible to synthesize java bytecode containing multidimensional arrays. The heap can be accessed by all PEs via a given number of memory ports. Thus, the number of memory ports limits the number of parallel memory accesses in one time step. We assume that each port has a dedicated optimal cache so that a DMA access takes one clock cycle.

In this work we use 16 PEs in order to be able to evaluate the compiler optimizations purely without imposed hardware constraints. Usually smaller CGRAs of size 4 to 8 PEs are used. Preliminary estimations show that more than four caches are hard to manage in terms of cache coherence. Thus, the number of memory ports is set to 4. Each memory port is accelerated by a dedicated cache.

Algorithm 1: Minimal Code Example

1 **for** $(i = 0;\ i < 10;\ i{+}{+})$ **do** $a[i] = i;$;

Algorithm 2: Code Example visualizing loop unrolling

1 **for** $(i = 0;$ $i < 10;$ $i{+}{+})$ **do**
2     $a[i] = i;$
3     $i{+}{+};$
4   **if** $i < 10$ **then**
5         $a[i] = i;$
6         $i{+}{+};$
7   **if** $i < 10$ **then**
8             $a[i] = i;$

III. Synthesis Optimizations

A. Speculative method inlining

Method inlining is done as a first step on bytecode level. Static methods and private object-methods can be inlined directly in Java, as the corresponding class is known a priori. Public object-methods on the other hand need a special handling as the class of the current object can only be determined during runtime. Thus, we implemented a speculative method inlining mechanism. We assume that the class of the calling object is the same as the declared class and no subclass of it. In order to verify this assumption, the class-index of the declared class is stored in the CGRA as a constant. During runtime the actual class-index of the current object is loaded via DMA from the heap. If both indexes differ, the assumption was wrong and a rollback mechanism has to be invoked. In our current version no rollback mechanism is implemented yet. Instead the simulation is simply aborted. This case did not occur in any of the benchmarks used for this paper.

In order to keep the synthesis times small we only inline methods whose code is shorter than 1000 bytes. In all our benchmarks this limit was never reached.

B. Partial Unrolling of Inner Loops

Experiments showed that the PE utilization can be very low for real life applications. Thus we extended the original AMIDAR synthesis algorithm so that the innermost loop of nested loops can be unrolled partially by a predefined factor u. The unrolling is done on bytecode level by copying the loop body u-times into the loop. The loop body of Algorithm 1 consists of the array assignment plus the comparison of the loop condition and the increment of the index variable which are both implied in the loop definition. The partial loop unrolling of Algorithm 1 with $u = 3$ results in a code which has the same behaviour as Algorithm 2.

The corresponding dependency graph is shown in Fig. 2. Comparison, array assignment and index increment are mapped to the Nodes `CMP(i)`, `f(i)` and `i++` respectively.

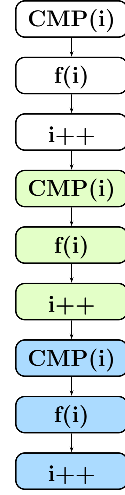

Fig. 2. Dependency graph of Algorithm 2

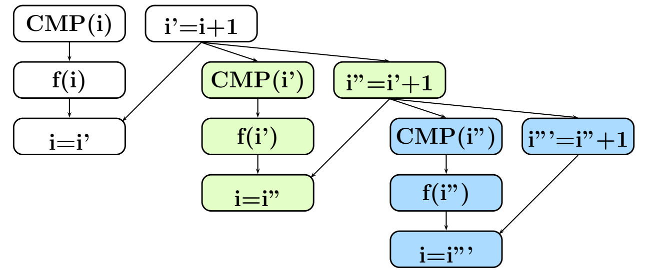

Fig. 3. Improved dependency graph of Algorithm 2

Scheduling this graph would not lead to a substantial speedup as the different loop iterations (marked in white, green and blue) are strictly dependant. To overcome this circumstance the instruction `i++` is split into two instructions `i'=i+1` (*calculation*) and `i=i'` (*store*) during graph generation. So different versions of the variable `i` are created which leads to an improved dependency graph shown in Fig. 3.

Both store instructions `i=i''` and `i=i'''` only receive a write-enable signal if the corresponding `CMP` instruction returns true.

The splitting of instructions into *calculation* and *store* is done for each local variable to decrease dependencies between adjacent loop iterations.

In the worst case the number of loop iterations N is no multiple of u. Then $u-1$ unrolled iterations of the original loop body (green and blue) are executed unnecessarily. The number of unnecessarily executed loop iterations is $e = u \cdot \lceil \frac{N}{u} \rceil - N < u$. From this follows that if $N \gg u$ the effect is negligible. If N is known in advance u should be chosen to be a divider of N so that $e = 0$.

1) Relation to Modulo Scheduling: This procedure makes use of the same parallelism buried in the code as modulo scheduling with reduced effort. If the unroll factor u happens to be the same as the actual number of loop iterations N,

both procedures are equivalent in terms of execution time. Otherwise the extended loop body has to be executed $p = \lceil \frac{N}{u} \rceil$ times. This means that the pro- and epilog are executed $p - 1$ times more than in modulo scheduling. In order to minimize p the unroll factor u should be chosen as high as possible while keeping the time needed for processing the extended loop body and the error e in mind.

2) Upper limits of the Unroll Factor: In the AMIDAR simulation framework the user can define an unroll factor u_{user} which will be applied to each loop when it does not exceed the upper bounds that will be described below. Thus, the unroll factor for a loop is

$$u = min(u_{user}, u_{max1}, u_{max2}).$$

Using a high value of u for large loop bodies with length b leads to long bytecode sequences which cannot be processed in a reasonable amount of time for online synthesis. Thus, we defined a maximum bytecode length b_{max} which should not be exceeded by unrolling. From this an upper bound for the unroll factor follows:

$$u_{max1} = max\left(1, \frac{b_{max}}{b}\right).$$

Tests also showed that the synthesis time increases significantly with the unroll factor when many dependencies are present in the datagraph. Additionally, if there are many dependencies between two adjacent loop iterations, it is not possible to overlap the execution of both iterations. Thus, it is possible to reduce the unroll factor u in those cases in order to reduce synthesis time without sacrificing performance. Unfortunately dependencies can only be evaluated after the generation of the data graph (thus after unrolling). Hence, we have to estimate the dependencies with a heuristic.

In this paper we chose a heuristic which uses the number of array accesses in the given bytecode sequence as a measure for the number of dependencies in the datagraph. This is reasonable as in online synthesis it is not possible to analyse memory aliasing thoroughly. Thus, successive array accesses will be considered as dependent in most cases. All the array accesses in the given bytecode sequence (n_a) and the array accesses in the loop body to be unrolled are counted (n_l). With an unroll factor u the total number of array accesses can be calculated with $n = n_a + n_l \cdot (u - 1)$.

With a given value n_{max} we determine a second upper bound

$$u_{max2} = max\left(1, \frac{n_{max} - n_a}{n_l} + 1\right).$$

Empirical tests showed that a value of $b_{max} = 3000$, $n_{max} = 125$, and $u_{user} = 5$ lead to an average decrease of $22.6\,\%$ in synthesis time for all benchmarks compared to applying u_{user} to all loops globally. The synthesis time for the worst case even decreases by $94.6\,\%$. The speedup decreases only by $3.4\,\%$ on average. These parameters will be used in the rest of the paper.

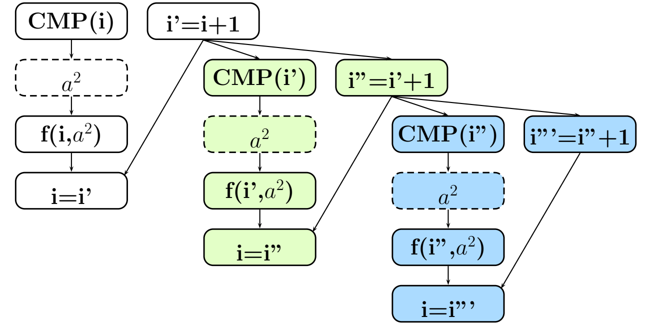

Fig. 4. Dependency graph with common subexpressions (dashed)

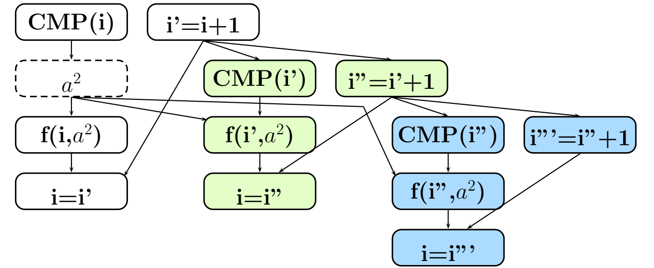

Fig. 5. Dependency graph with eliminated common subexpressions

C. Common Subexpression Elemination (CSE)

We detect common subexpressions during data flow graph generation iteratively. Before a node is inserted in the graph, all nodes with the same operation are examined whether they have the same predecessor nodes. If so, the node is not inserted in the graph but the node already in the graph will be reused.

Normally CSE is performed by the compiler so performing CSE alone during dataflow graph generation does not contribute significant speedup. This is even the case when using the javac compiler which does not optimize.

Fig. 4 and Fig. 5 show that when CSE is used in combination with partial loop unrolling new common subexpressions are generated which cannot be detected during compile time. In this case CSE has a similar effect as loop-invariant code motion while it is not necessary to determine whether a code is actually loop invariant. The invariant code will now be executed p times instead of N times, while with true loop-invariant code motion it would only be executed once.

IV. Evaluation

To evaluate the feasibility of the high level compiler optimizations described in Section III we used a set of benchmark applications listed in Table I. All benchmarks used unaltered kernel code with the exception of Serpent cipher. The code was improved by compacting several lines of code in one loop. The synthesis algorithm used in this paper also benefits from this modification. The sources were taken from [5], [6] and [7] and were compiled with *javac* compiler.

In order to cancel out the runtime of initialization code,

we execute all benchmarks in a long and a short version and calculate the difference of both runtimes.

To obtain the time needed for synthesis we used the Java-function `System.nanoTime()`. As for now the (single-threaded) synthesis algorithm is still executed on the host as no AMIDAR hardware is available yet. To obtain runtimes that are comparable to the runtimes on a future AMIDAR implementation we executed all tests on a CuBox I2W-300-D computer with a i.MX6 dual core processor with 1 GHz and 1 GB of RAM. All simulations are executed five times and the minimum of the synthesis times for each kernel is determined in order to cancel out artifacts induced by the OS scheduler on the host PC. The synthesis time for one benchmark is the sum of the synthesis times of all kernels in this benchmark.

We simulated all benchmarks with all eight combinations of the three optimizations (inlining = I, unrolling = U and CSE = C) switched on or off respectively. For example $IUC = 000$ means all optimizations are switched off whereas $IUC = 001$ means that just CSE is switched on. Table I shows the speedup compared to the execution on AMIDAR without synthesis (black). The speedup gain due to the optimizations is given in percent (magenta) whereas the synthesis times are given in milliseconds (green).

It can be seen in the last row that on average the speculative method inlining contributes the main part to the speedup (cases 100 - 111) while CSE contributes least. Thus, the speedup gain increases strictly monotonically from case 000 to 111.

It is also obvious that CSE becomes more effective when used in combination with partial loop unrolling and method inlining (gain increases by 88 percentage points) as it has similar effects like loop invariant code motion as described above.

Table I also shows the average synthesis time for all simulations. It can be seen that both inlining and unrolling increase the synthesis times significantly. In contrast to that CSE even decreases synthesis times in most cases because CSE decreases the size of the graphs to be scheduled. For large code sequences (e.g. when inlining and unrolling is switched on) this effect is outweighed by the increased effort to find common subexpressions.

On average all optimizations in combination result in an speedup gain of 1105 % while the synthesis time only increases from 373 ms to 2035 ms. It is obvious that the speedup caused by the combination of all optimizations is usually not equal to the linear combination of the speedups of the single optimizations on their own.

It has to be noted that the actual speedups and synthesis times are highly dependant on the current benchmark. CubeHashTest (row marked in blue in Table I) for example only benefits from CSE while the GrayscaleFilterTest (green) only benefits from loop unrolling. The SwizzleFilterTest (yellow) on the other hand highly benefits from the combination of all optimizations and achieves a speedup gain of 5456 %. In contrast to that DESedeTest_sbe (red) even has a negative gain which is caused by the error e in loop unrolling. MD5Test and SIMD512Test (lavender) do not have a significant speedup gain because the AMIDAR hardware profiler does not recognize the relevant bytecode sequences properly. Thus, the optimizations

TABLE II. STANDARD DEVIATION OF THE SIMULATION RESULTS FOR ALL EIGHT COMBINATIONS OF THE THREE OPTIMIZATIONS SWITCHED ON AND OFF. (INLINING = I, UNROLLING = U AND CSE = C)

Standard deviation	IUC 000	IUC 001	IUC 010	IUC 011	IUC 100	IUC 101	IUC 110	IUC 111
speedup gain	0%	12%	22%	26%	1041%	1056%	1244%	1440%
synthesis time	562	546	1404	1536	1280	1126	1933	1954

cannot unfold their potential. The standard deviation for both the speedup gain and the synthesis time are given in Table II.

Further investigations showed that the number of bytecode instructions of a synthesized kernel does not correlate strongly with synthesis time or speedup. The following numbers refer to the case 111. In Blake256Test for example two kernels are synthesized of which one consists of 1868 instructions. The synthesis time for this kernel is 1537 ms while an overall speedup of 26.34 was achieved. On the other hand in SIMD512Test 12 kernels (with 1222 instructions on average) were synthesized but this resulted in a speedup of only 2.30. The longest kernel consisted of 1522 instructions and took 1146 ms to synthesize. The GrayscaleFilterTest only has one kernel with 297 instructions but achieves a speedup of 19.74 while the synthesis time of 706 ms is relatively high.

In sum the results show that all optimizations discussed in this paper are feasible in online synthesis while some restrictions on the bytecode length have to be set for both method inlining and partial loop unrolling as discussed in Section III.

V. RELATED WORK

To the best of our knowledge, no other approaches are mapping software to a CGRA at runtime. Thus, no immediate competitors are known.

AMIDAR relies on CGRAs as the acceleration technology. CGRAs have been used to speed up applications for quite a while. They typically depend on compile time analysis and generate a single datapath configuration for an application beforehand: RaPiD [8], PipeRench [9], Kress-Arrays [10] or the PACT-XPP [11]. In most cases, specialized tool sets and special purpose design languages had to be employed to gain substantial speedups. Whenever general purpose languages could be used to program these architectures, the programmer had to restrict himself to a subset of the language and the speedup was very limited.

Static transformation from high level languages into a CGRA is also investigated by several groups. The DRESC[12] tool chain targeting the ADRES[13] architecture is one of the most advanced tools. Yet, it requires hand written annotations to the source code and in some cases even some hand crafted rewriting of the source code. Compilation times easily get into the range of days for a single instance of the CGRA.

The RISPP architecture [14] lies between static and dynamic approaches. Here, a set of candidate instructions are evaluated at compile time. The HW primitives for these candidates (atoms) have to be implemented manually. These candidates are constructed dynamically at runtime by varying sets of the predefined atoms. Thus, alternative design points are chosen depending on the actual execution characteristics.

TABLE I. SIMULATION RESULTS FOR ALL BENCHMARKS SHOWING SPEEDUP (BLACK), SPEEDUP GAIN IN PERCENT (MAGENTA) AND SYNTHESIS TIME FOR ALL KERNELS IN SUM IN MILLISECONDS (GREEN). ALL EIGHT COMBINATIONS OF THE THREE OPTIMIZATIONS SWITCHED ON OR OFF ARE SIMULATED. (INLINING = I, UNROLLING = U AND CSE = C)

Benchmark	IUC = 000			IUC = 001			IUC = 010			IUC = 011			IUC = 100			IUC = 101			IUC = 110			IUC = 111		
	sp	ga	ti	sp	ga	ti	sp	ga	ti	sp	ga	ti	sp	ga	ti	sp	ga	ti	sp	ga	ti	sp	ga	ti
AESTest_rkg	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	5.6	+456%	496	5.6	+456%	503	5.5	+455%	1183	5.6	+457%	1239
AESTest_sbe	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	5.0	+397%	591	5.0	+405%	586	4.9	+386%	746	4.9	+389%	814
BlowfishTest_rkg	1.0	+0%	504	1.0	+0%	506	1.0	+0%	1038	1.0	+0%	1026	20.6	+1930%	812	20.6	+1930%	804	19.1	+1781%	1826	19.2	+1786%	1733
BlowfishTest_sbe	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	4.1	+311%	97	4.1	+311%	92	4.0	+299%	237	4.0	+299%	235
DESedeTest_rkg	9.6	+0%	824	9.9	+3%	831	17.2	+78%	1964	19.1	+98%	1959	9.6	+0%	822	9.9	+3%	831	17.2	+78%	1959	19.1	+98%	1953
DESedeTest_sbe	5.3	+0%	199	5.2	-0%	186	5.2	-1%	482	5.2	-2%	524	5.3	+0%	198	5.2	-0%	182	5.2	-1%	479	5.2	-2%	524
IDEATest_rkg	2.8	+0%	540	2.8	+0%	551	2.8	+0%	1452	2.8	+0%	1527	5.9	+114%	587	5.9	+114%	594	6.2	+124%	1582	6.2	+124%	1679
IDEATest_sbe	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	5.4	+444%	307	5.4	+444%	311	5.1	+414%	1006	5.1	+414%	1068
RC6Test_rkg	1.1	+0%	378	1.1	+0%	376	1.1	+0%	669	1.1	+0%	659	19.0	+1625%	507	19.0	+1625%	505	23.9	+2065%	1102	23.9	+2065%	1108
RC6Test_sbe	1.1	+0%	81	1.1	+0%	81	1.1	+0%	229	1.1	+0%	234	11.4	+912%	247	11.4	+912%	235	11.9	+955%	601	11.9	+955%	622
SerpentTest_rkg	1.0	+0%	280	1.0	+0%	280	1.0	+0%	389	1.0	+0%	391	12.7	+1159%	1465	13.1	+1197%	1451	13.5	+1243%	1839	14.0	+1286%	1835
SerpentTest_sbe	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	10.9	+989%	1205	11.8	+1075%	1189	10.9	+989%	1009	11.8	+1075%	1027
SkipjackTest_rkg	13.6	+0%	373	13.6	+0%	374	19.2	+42%	709	19.6	+45%	711	13.6	+0%	373	13.6	+0%	373	19.2	+42%	712	19.6	+45%	710
SkipjackTest_sbe	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	10.4	+938%	605	10.4	+938%	588	10.9	+987%	3842	10.9	+995%	3455
TwofishTest_rkg	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	39.4	+3835%	2050	40.3	+3932%	2127	44.0	+4301%	4848	51.2	+5015%	4928
TwofishTest_sbe	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	6.3	+529%	149	6.1	+508%	134	6.1	+507%	542	5.8	+484%	575
XTEATest_rkg	11.3	+0%	433	11.3	+0%	432	14.9	+33%	841	15.6	+39%	838	11.3	+0%	433	11.3	+0%	434	14.9	+33%	838	15.6	+39%	837
XTEATest_sbe	5.6	+0%	59	5.6	+0%	60	5.6	-1%	179	5.6	-1%	178	5.6	+0%	59	5.6	+0%	60	5.6	-1%	178	5.6	-1%	178
BLAKE256Test	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	25.9	+2494%	1273	26.1	+2511%	1316	26.3	+2526%	2005	26.3	+2534%	2163
CubeHash512Test	10.9	+0%	1860	17.8	+64%	1550	10.9	+0%	1873	17.8	+64%	1557	10.9	+0%	1855	17.8	+64%	1544	10.9	+0%	1872	17.8	+64%	1559
ECOH256Test	7.3	+0%	966	7.5	+2%	941	7.2	-1%	1789	7.4	+1%	1767	30.2	+312%	7043	34.0	+365%	6006	31.2	+326%	7937	35.3	+383%	7270
MD5Test	1.0	+0%	276	1.0	+0%	276	1.0	+0%	362	1.0	+0%	362	1.0	+0%	276	1.0	+0%	275	1.0	+0%	361	1.0	+0%	362
RadioGatun32Test	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	19.0	+1797%	1229	18.9	+1794%	1268	19.0	+1797%	1958	19.0	+1797%	2103
SHA1Test	1.4	+0%	379	1.4	+0%	381	1.5	+1%	655	1.5	+1%	666	15.0	+940%	1376	15.0	+940%	1400	7.2	+397%	3852	7.2	+397%	4175
SHA256Test	1.0	+0%	265	1.0	+0%	266	1.0	+0%	351	1.0	+0%	350	32.0	+3082%	1468	32.0	+3082%	1556	33.1	+3191%	1736	33.1	+3191%	1820
SIMD512Test	2.2	+0%	1987	2.2	+0%	1988	2.2	+1%	6685	2.2	+1%	7851	2.3	+4%	2174	2.3	+4%	2231	2.3	+5%	6928	2.3	+5%	8140
ContrastFilterTest	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	10.4	+938%	526	10.4	+938%	525	27.4	+2644%	1191	32.4	+3136%	1210
GrayscaleFilterTest	10.6	+0%	383	10.6	+0%	385	19.7	+87%	688	19.7	+87%	707	10.6	+0%	383	10.6	+0%	385	19.7	+87%	689	19.7	+87%	706
SobelFilterTest	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	10.1	+914%	1367	11.1	+1005%	1371	15.4	+1438%	2924	15.4	+1441%	2737
SwizzleFilterTest	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	1.0	+0%	0	30.6	+2956%	630	31.6	+3056%	621	43.5	+4253%	1613	55.6	+5456%	1601
JpegEncode	2.5	+0%	1788	2.5	+0%	1897	2.6	+7%	4080	2.7	+9%	3616	6.6	+169%	2226	6.8	+175%	2283	7.8	+218%	5221	8.6	+251%	4716
Averages		+0%	373		+2%	366		+8%	788		+11%	804		+879%	1059		+896%	1025		+1017%	2026		+1105%	2035

Dynamic transformation from software to hardware has been investigated already by other researchers. Warp processors dynamically transform assembly instruction sequences into fine grain reconfigurable logic[15]. Yet, only very short basic blocks are taken into consideration, delivering only very limited application speedups. No optimizations are applied before mapping the instruction sequences to logic. In AMIDAR processors, full loop bodies, even including conditional execution paths, can be transformed into configurations of reconfigurable hardware in the form of a CGRA[16][17]. However, few bytecode instructions can not be mapped to the CGRA and also, in some cases, no acceleration can be obtained by the mapping.

A CGRA architecture that has been designed to simplify scheduling and placement of operations has been investigated in [18]. The resulting synthesis algorithm runs very fast (well below one second even for moderate problems), but does not employ the high level optimizations discussed in this paper.

Other approaches also try to map compute intense parts of the application to reconfigurable datapaths[19][20][21]. These approaches both use configurable datapaths with very limited options for the routing of intermediate results. Longer sequences of code or even nested loops can not be mapped to them. The published material describes no high-level optimizations.

In [22][23] the authors apply loop pipelining. The authors consider executing this task during runtime but up until now the computation of the configuration is done offline.

VI. CONCLUSION AND OUTLOOK

Several high level compiler optimization are indeed feasible during online synthesis. Our experiments have shown that with reasonable restrictions for unroll factors and the length of inlined methods, synthesis times still remain small enough for execution at runtime. On the other hand, the gain that can be achieved through these optimizations is substantial. Yet, the effects of individual optimizations are application dependent.

Currently, we work on a HW implementation of the AMIDAR processor. A first prototype has already greeted the world with "Hello World!".

In the future we want to further improve the quality of the software mapping and also we want to simplify the base CGRA design. To this end we are investigating simplified modulo scheduling approaches. Also, currently a real HW implementation of the CGRA on 28nm technology is investigated.

REFERENCES

[1] S. Gatzka and C. Hochberger, “The AMIDAR class of reconfigurable processors,” *The Journal of Supercomputing*, vol. 32, no. 2, pp. 163–181, 2005.

[2] ——, “Hardware based online profiling in amidar processors,” in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, April 2005, pp. 144b–144b.

[3] C. Hochberger, L. J. Jung, A. Engel, and A. Koch, “Synthilation: JIT-Compilation of Microinstruction Sequences in AMIDAR Processors,” in *DASIP*, 2014, pp. 193–198.

[4] S. Gatzka and C. Hochberger, “On the scope of hardware acceleration of reconfigurable processors in mobile devices,” in *HICSS*, 2005, p. 299.

[5] "JH Labs," http://www.jhlabs.com/ip/filters/, accessed: 2015-10-12.

[6] "Dustin Sallings on Github," https://github.com/dustin/photo/tree/master/src/java/net/spy/photo, accessed: 2015-10-12.

[7] "Bouncy Castle," http://www.bouncycastle.org/, accessed: 2015-10-12.

[8] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD - reconfigurable pipelined datapath," in *FPL*, 1996, pp. 126–135.

[9] Y. Chou, P. Pillai, H. Schmit, and H. P. Shen, "Piperench Implementation of the Instruction Path Coprocessor," in *MICRO*, Monterey, 2000, pp. 147–158.

[10] R. W. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "Mapping applications onto reconfigurable Kress Arrays," in *FPL*, 1999, pp. 385–390.

[11] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP — a self-reconfigurable data processing architecture," *The Journal of Supercomputing*, vol. 26, no. 2, pp. 167–184, Sep. 2003.

[12] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," in *DATE*, 2003, pp. 10 296–10 301.

[13] ——, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *FPL*, 2003, pp. 61–70.

[14] L. Bauer, M. Shafique, S. Kramer, and J. Henkel, "RISPP: Rotating instruction set processing platform," in *DAC*, 2007, pp. 791–796.

[15] R. L. Lysecky and F. Vahid, "Design and implementation of a microblaze-based WARP processor," *ACM Trans. Embedded Comput. Syst.*, vol. 8, no. 3, pp. 1–22, 2009.

[16] S. Döbrich and C. Hochberger, "Exploring online synthesis for cgras with specialized operator sets," *International Journal of Reconfigurable Computing*, vol. 2011, p. 10, 2011.

[17] S. Döbrich and C. Hochberger, "Practical resource constraints for online synthesis," in *Proceedings of the 5th International Workshop on Reconfigurable Communication-centric Systems on Chip (ReCoSoC 2010)*, O. S. Michael Hübner, Loïc Lagadec and J. Becker, Eds. KIT Scientific Publishing, 2010, pp. 51–58.

[18] R. Ferreira, J. Vendramini, L. Mucida, M. Pereira, and L. Carro, "An FPGA-based heterogeneous coarse-grained dynamically reconfigurable architecture," in *Compilers, Architectures and Synthesis for Embedded Systems (CASES), 2011 Proceedings of the 14th International Conference on*, Oct 2011, pp. 195–204.

[19] J. Bispo, N. Paulino, J. Cardoso, and J. Ferreira, "From instruction traces to specialized reconfigurable arrays," in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, Nov 2011, pp. 386–391.

[20] J. Bispo and J. Cardoso, "Techniques for dynamically mapping computations to coprocessors," in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, Nov 2011, pp. 505–508.

[21] L. Chen, J. Tarango, T. Mitra, and P. Brisk, "A just-in-time customizable processor," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 524–531. [Online]. Available: http://dl.acm.org/citation.cfm?id=2561828.2561930

[22] N. Paulino, J. a. C. Ferreira, J. a. Bispo, and J. a. M. P. Cardoso, "Transparent acceleration of program execution using reconfigurable hardware," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015, pp. 1066–1071. [Online]. Available: http://dl.acm.org/citation.cfm?id=2757012.2757061

[23] R. Ferreira, V. Duarte, W. Meireles, M. Pereira, L. Carro, and S. Wong, "A just-in-time modulo scheduling for virtual coarse-grained reconfigurable architectures," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, July 2013, pp. 188–195.