Synthilation: JIT-Compilation of Microinstruction Sequences in AMIDAR Processors

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

Andreas Engel and Andreas Koch
Department for Computer Science
Embedded Systems and Applications Group, TU Darmstadt
Email: {engel,koch}@esa.informatik.tu-darmstadt.de

***Abstract*—The large expense of current chip fabrication can generally only be amortized for large manufacturing volumes. Thus, it is desirable to build adaptable chips that can be customized to the application needs after production. In this contribution we show that this adaptation is possible even without reconfigurable HW components. We propose *synthilation*, a new method for adapting the processor to the application requirements. It combines methods of hardware synthesis and software compilation to map high-level descriptions to hardware components of the processor. Our approach is applicable to varying degrees of reconfigurability, reaching from static microarchitectures just with writable control stores (variable microcode), to the exploitation of instruction level parallelism with multiple computational units. We consider both a practical real-world example as well as theoretical bounds on the speed-ups achievable by our method.**

I. Introduction

Following Moore's law, the number of transistors on a chip doubles every 24 months. After being valid for more than 40 years, the end of Moore's law has often been forecast. Yet, technological advances have kept the progress intact.

Due to increasing design and manufacturing costs, we will need to make chips adaptable to the needs of individual applications. Reconfigurable logic in different granularities has been proposed to solve both problems[1]. It allows us to build large quantities of chips and yet use them individually. Field programmable gate arrays (FPGAs) and Coarse Grain Reconfigurable Arrays (CGRAs) are technologies that are currently in use for this purpose. Yet, programming those devices is a complicated task which has to be done by experts.

FPGAs and CGRAs can be used as hardware accelerators. In the past researchers have tried to build systems that synthesize configurations for these hardware accelerators on the fly while the application was running[2][3]. Runtime profiling of the application was used to identify suitable code sequences and online synthesis was used to create hardware configurations of the accelerating hardware. Although the potential speedups are very good, this method has the disadvantage that it is not possible with all types of applications.

In this contribution we will explore a different type of reconfiguration and a different way to accelerate applications. We will make use of the already available hardware resources in an application optimized way. To this end, just-in-time compilation of instruction sequences into optimized datapath control sequences will be carried out. We will show that this delivers good results without additional resources, but can be improved by adding a small number of ALUs and scratchpad memories.

A. Related Work

Static transformation from high level languages into fine grain reconfigurable logic has been researched by a number of academic and commercial research groups. Only very few of them support the full programming language[4].

The $(GECO)^2$-Architecture[5] provides a graphical user interface to combine basic operations of digital signal processors (DSP) to complex algorithms. These basic operations are executed by a CGRA implemented on a Flash-based FPGA, but the mapping of applications has to be done manually.

Static transformation from high level languages into coarse grain reconfigurable logic is also investigated by several groups. The DRESC[6] tool chain targeting the ADRES[7] 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. Also, the compilation times easily get into the range of days.

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[2]. Yet, only very short basic blocks are taken into consideration, delivering only very limited application speedups. In AMIDAR processors, full loop bodies even with conditional execution paths can be transformed into configurations of reconfigurable hardware in the form of a CGRA[8][9]. It should be noted that not all potential code sequences can be mapped to the CGRA and also that in some cases no acceleration will result from a mapping to the CGRA.

The processor architecture used in this contribution executes Java bytecode. Many existing bytecode processors try to avoid notorious data transfers to and from the stack memory. This approach is called it instruction folding. Different mechanisms for the folding logic have been proposed. The first published mechanism was included in the PicoJava II processor[10]. It allows folding of six well defined groups of instruction pairs which are folded to shorter sequences. It results in a folding of 42% of all stack operations. The Producer-Operator-Consumer[11] and its successor the extended POC folding mechanism[12] allow a more general folding of instructions. All instructions are categorized into producers, operators and consumers. According to these classsifications folding patterns are defined. EPOC uses a very large instruction buffer of up to 72 byte capacity. Complex rules also need to reorder instructions to achieve a folding of a total of 95% of all stack operations. The resulting hardware is considerably large when compared to the instruction issue logic.

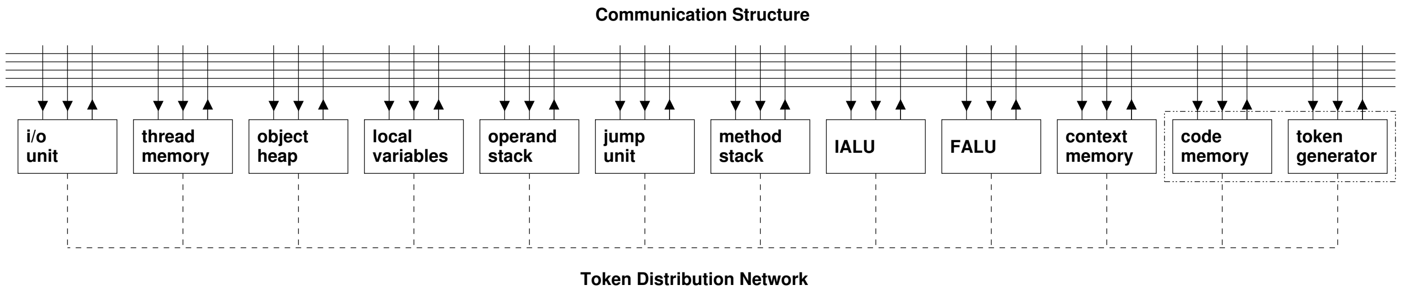

Fig. 1. Model of a Java Virtual Machine on AMIDAR Basis

B. Paper Outline

In the following section we will explain the processor model used in this work. Section 3 will explain our new method for processor customization. Section 4 shows the application domain that was used for evaluation purposes. It will be followed by a discussion of the achieved results and a conclusion of this contribution.

II. THE AMIDAR PROCESSING MODEL

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 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.

Figure 1 sketches the structure of an exemplary AMIDAR processor for the execution of Java Bytecode. The token generator fetches an instruction the code memory and retrieves a set of tokens from the token memory that composes the semantics of the instruction. This set of tokens is distributed to the FUs over a dedicated token distribution network. The token generator can distribute token sets efficiently, such that FUs have enough tokens in their input queues.

To illustrate the composition of token sets for instructions, we refer to Figure 1. Figure 2 shows a Java bytecode sequence and the corresponding token sets for each of the bytecode instructions. E.g. the `aload_1` instruction is composed of two tokens, which are sent to the *local variables* FU and the *operand stack* FU. As a different example, the `iadd` instruction consists of a total of four tokens. One token is sent to the *IALU* FU and three are sent to the *operand stack* FU. Those three have to be executed by the FU in the given order. Thus, the token generator has to send them one after each other.

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. Eventually, one of the tokens must trigger the transport of the next instruction to the token generator.

A detailed explanation of the model, its application to bytecode execution and its specific features can be found in [13].

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. In [14] we show how to identify the conflicting bus taps and also a heuristic to modify the bus structure to minimize the conflicts is shown. Also, in [13] we show that 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. In [15] we have presented mechanisms to identify code sequences as candidates for a HW implementation. This profiling is part of the code memory. 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 map the execution of the sequence to customizable FUs. The result of the mapping can be stored and following occurences of the sequence can make use of the optimization result.

We have also shown that HW implementations of relevant code sequences can lead to substantial application speedup (or power saving vice versa)[16].

Previously, we have only used CGRAs that are included in the processor to map code sequences onto HW. But in this work we are using a different method that is inspired by the idea of instruction folding.

C. Instruction Folding

Executing Java bytecode on AMIDAR processors makes heavy use of the stack memory, which is due to the nature of the stack oriented bytecode. Many of these transfers can be avoided by looking at longer sequences of the bytecode. Figure 2 shows a bytecode sequence together with its token sequence for the AMIDAR processor. It executes the Java statement `lv2 = lv1[lv2 + 10];`. On the right-hand side, the data dependencies have also been shown. In an ideal situation all the tokens targeting the stack memory (PUSH and POP) would be eliminated. It is easy to see that this can be achieved in the given situation by simply sending the data to the final consumer of the data.

Figure 3 shows the final result of instruction folding in this case. The resulting sequence is substantially shorter and will execute accordingly faster.

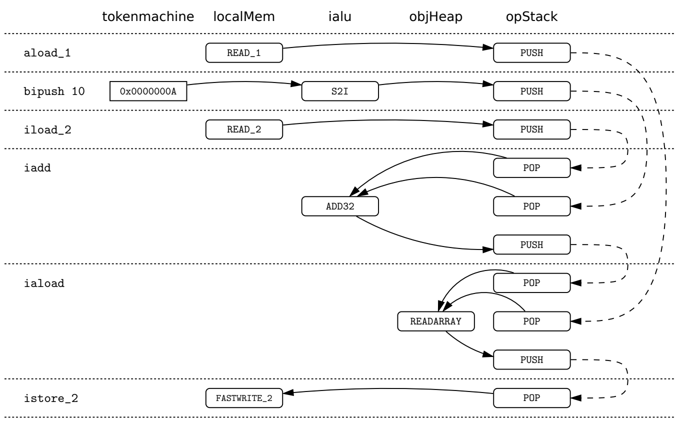

Fig. 2. Sample sequence of bytecode instructions and the corresponding set of tokens for an AMIDAR processor

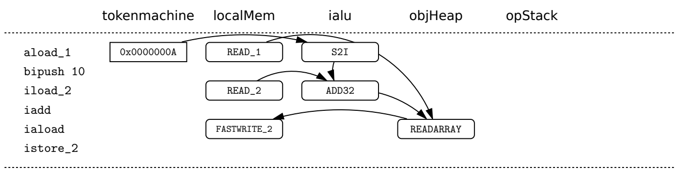

Fig. 3. The completely folded version of the already shown bytecode sequence

III. Synthilation

Although instruction folding proved useful in real bytecode processors, a thorough analysis shows that none of the mentioned mechanisms can fold all potentially foldable stack transfers. Some instruction sequences would only be foldable if the sequence of instructions would be reordered. This cannot be done concurrently with folding instructions in hardware.

This insight gives rise to a new idea how we can avoid the stack transfers. Rather than implementing the instruction folding in hardware, we make use of the existing profiling mechanism that is described in section II-B. Frequently used instruction sequences are then compiled into an optimized token set which carries out the whole bytecode sequence as one virtual instruction. This can be seen as a just-in-time compilation of tokens (or microinstructions). The methods which are employed in this JIT-compiler borrow concepts from traditional compiler engineering as well as from hardware synthesis approaches. Thus, we came up with the new term **synthilation** as a mixture between synthesis and compilation.

In order to support synthilation, the underlying machine requires only two modifications: 1) The token memory must contain an empty area that is writable by the runtime system. 2) A new software thread must be launched at system startup. This thread will be triggered whenever a suitable instruction sequence has been found by the hardware profiler. The thread will then try to compute an optimized token set for the sequence and store it in the token memory.

In contrast to the synthesis approach, no additional hardware is required to support synthilation. It only uses the already available hardware in a more efficient way (Yet, additional hardware *can* be used).

The synthilation process is very similar to the synthesis process. It starts from an instruction graph which is constructed

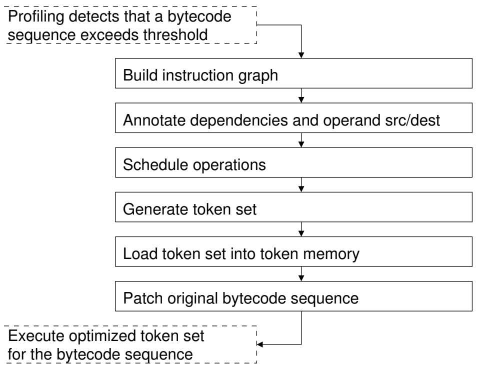
Profiling detects that a bytecode sequence exceeds threshold
Build instruction graph
Annotate dependencies and operand src/dest
Schedule operations
Generate token set
Load token set into token memory
Patch original bytecode sequence
Execute optimized token set for the bytecode sequence

Fig. 4. All process steps for the synthilation of an optimized token set

from the instructions in the candidate sequence. In a second step this graph is annotated with dependency information and information regarding source and destination of operands. The annotated graph is then subject to scheduling and binding. Eventually, the scheduled graph is traversed to evaluate the corresponding token set. Fig. 4 shows the steps for creating an optimized token sequence.

An example of such an optimized token set can be seen in Figure 3. The original bytecode sequence is shown in Figure 2. The new token set is associated with a new virtual instruction. This instruction is stored at the beginning of the bytecode sequence in the program memory and it automatically skips the remaining instructions of the sequence. When the token generator sees this new instruction, it distributes the optimized token set. For this short example sequence, an advanced version of instruction folding would have had the same effect. Yet, synthilation can deal with much longer sequences and also, it can use additional hardware to gain higher performance as we will show in section III-C.

In a mixture of synthetic benchmarks, we have seen speedups of ≈ 2.6. This is already better than the speedup of 1.8 which is achieved by the EPOC ruleset. It should be noted that these speedups were achieved for code fragments which were dominated by compute kernels. In real world applications, achievable speedups will most likely be smaller.

Synthilation is a concept that can be employed in all stack based microarchitectures which are microprogrammed.

A. *Sequence Improvement*

The initially generated token sequences can be further optimized. In several cases, it could be seen that the bytecode carried out the same sequence of operations and accesses to local variables multiple times. Thus, the executed code could benefit from *common subexpression elimination* (CSE). Our implementation of CSE has a complexity of $O(n^2)$. Thus, running it in the target system, which is our intention, could lead to noticeable delays of the synthesis process.

B. *Small Hardware Improvements*

Preliminary analysis of the generated token sequences revealed two minor weaknesses. Firstly, in several cases the token sequence contained more than one constant. Adding one additional port to send two constants concurrently slightly improved the timing of the sequences. Secondly, the generated token sequences produced suboptimal memory rows in the token memory. The rows contained empty places that could be filled with tokens from subsequent rows. Thus, we implemented a peephole optimizer that tries to compact the token sets. This compaction only reduces the amount of memory needed to store the sequence. It does not improve the timing.

C. Additional Hardware Resources

A more thorough analysis of the generated token sequences disclosed additional potential for increased speedups. Some sequences exposed a certain amount of instruction level parallelism (ILP). The ILP is dependant on the nature of the application, but for performance demanding applications it is highly likely to expose considerable ILP.

In an unaltered AMIDAR processor, it would not be possible to parallelize these computations. But when we compile a sequence of tokens for a particular candidate instruction sequence, we can make use of additional ALUs to compute in parallel. First experiments in this direction showed that we might also need extra temporary storage in the range of no more than 128 words. The current synthilation algorithm can use an arbitrary number of ALU and additional memories (so called scratchpad memories). It tries to distribute the non dependant computations of the sequence to the different ALUs. Regular code will not make use of these additional resources.

IV. ADPCM as an Application Example

An implementation of an adaptive differential pulse code modulation (ADPCM) taken from [17] was used as an application example. ADPCM is used for example in ITU audio codec G.726 for IP telephony [18] like in DECT for cordless phones. The following subsections explain the algorithm and the code structure of the implementation used in this work in order to enable a proper discussion of the achieved speedups.

A. ADPCM

Simple differential pulse code modulation just transmits the difference of two succesive samples and an initial value. Thus, for highly autocorrelated input data like audio signals, the variance and the mean of all transmitted values is decreased. With appropriate coding it is now possible to transmit the information with less effort. In ADPCM the mean value and the variance are decreased further by transmitting the difference between a prediction for the next value and the actual value. This difference is also called prediction error $d_i = x_i - \tilde{x}_i$. The prediction is calculated with a linear prediction filter of order M with the coefficients $a_i, i \in 0...M-1$ and the previous samples $x_{i-M}, ..., x_{i-1}$:

$$\tilde{x}_i = \sum_{k=0}^{M-1} a_k \cdot x_{i-M+k}$$

The decoder needs to know the first M samples and the filter coeffiecient. Thus they are transmitted directly.

The filter coefficients are calculated for each block of size N in advance. To find the optimal filter coefficients the autocorrelation r_k of the current block is calculated. These values are used to build a system of linear equations, whose solution results in filter coefficients that minimize the variance of the prediction error sequence $(d_1, ..., d_N\ )$ [19].The prediction error sequence is mapped from signed to unsigned integer via code spreading.

These unsigned integer values are coded with Golomb-Rice-Coding which is very effective for input streams in which small values are more probable than large values. This is the case for the prediction error sequence for input streams like audio signals which can be predicted effectively. Golomb-Rice-Coding has one parameter which will be called *riceCoefficient* in this work.

B. Code Structure of ADPCM Implementation

The AMIDAR simulator framework does not yet support input streams. Thus the input and output streams of the ADPCM are simulated by arrays in this work. Both the encode and the decode methods receive the parameters `blockSize`, `order`, `riceCoefficient` plus the references to two arrays for input data and output data.

1) Structure of the Encode Method: The encode method consists of a single while loop which contains all functionality. If a new block starts, the prediction filter coefficients have to be calculated. Afterwards the prediction errors are calculated, encoded (Golomb-Rice-Encoding) and sent to the output stream. The structure of the code is shown in Algorithm 1.

Right hand-side comments show which loops are contained within the instructions. Loops whose number of iterations are dependant on the parameter `order` are marked with **O** while loops whose number of iterations change dynamically with the input data are marked with **D**. Finally, loops with a static number of iterations are marked with **S**. The operand $*$ denotes the scalar product of two vectors.

Algorithm 1 Encode

```
while input stream not finished do
    read next sample from byte stream;                $\rightarrow$ one loop (S)
    if calc new coefficients then
        if start of block then
            init correlation sum;                     $\rightarrow$ one loop (O)
        update correlation history;                   $\rightarrow$ one loop (O)
        calc correlation sum;                         $\rightarrow$ one loop (O)
        if end of block then
            calc auto corr val;                       $\rightarrow$ two nested loops (D/O)
            derive linear eq system;                  $\rightarrow$ two nested loops (O)
            gauss elimination;                        $\rightarrow$ four nested loops (O/O/D/O)
            get coefficients;                         $\rightarrow$ three nested loops (O)
            transmit coefficients;                    $\rightarrow$ two nested loops (O/D)
            calc new coefficients = false;
            rewind input stream for encoding;
    else
        if sample history full then
            prediction = coeff $*$ smpleHist;         $\rightarrow$ one loop (O)
            error = sample - prediction;
            output byte = encode error;               $\rightarrow$ two loops (D)
            write output byte;
            update sample history;                    $\rightarrow$ one loop (O)
            if end of block then
                calc new coefficents = true;
```

It can be seen, that the encoding itself consists of simple loops while the coefficients are determined using nested loops.

2) Structure of the Decode Method: The decode method consists of two parts. In both parts is necessary to check if a new block is transmitted in order to extract the new prediction filter coefficients. This is implemented with two nested loops (**O/S**). The first part initializes the decoder by filling the sample history. It contains one simple loop (**S**). The second part operates in steady state and uses the filter coefficients and the sample history to calculate a prediction for the next sample. In the next step the incoming bytes are decoded with the help of two while loops (Golomb-Rice-Decoding) in order to obtain the prediction error. Afterwards the current sample is calculated and the sample history is updated. In the last step the current sample is sent to the output stream. These parts add up to five simple loops (**O/D/D/O/S**) in total. It is obvious that nested loops are only executed when a new block is started.

C. Input data

The input data used in this work are neural activities of primates solving different tasks. The data was measured by a micro-electrode inside the probands brains at the German Primate Center in Göttingen. The sensor data was sampled with 16 bit resolution at a frequency of 24.414 kHz. Applying ADPCM to this input data, results in compressed data with a size of 36% of the original size[20].

V. Evaluation

The aims of this simulation are to determine the maximal achieveable speedup for the synthilation and to find the optimal recource configuration. The speedup is measured in relation to the execution time in cycles on AMIDAR without Synthilation and without instruction folding. The simulation was executed on our AMIDAR simulation framework.

The parameters that can be changed in this experiment are `riceCoeffecent`, `blockSize`, `order` and the hardware resources in AMIDAR. The impact of the parameter `riceCoeffecent` on the whole runtime can be neglected if the order is high enough. It is assumed that this is the case for values equal or greater than two. Hence the parameter will be constant for all simulations. This also applies to the parameter `blockSize` which has no significant influence on the runtime for large values.

Thus, the optimal simulation to achieve the aims mentioned above, would be a 3D sweep over `order`, number of ALUs and number of memories. As this is too costly the simulation was split into two parts. In the first part the optimal hardware configuration is determined with a 2D sweep over number of ALUs and number of memories. Afterwards this configuration is used for a sweep over the parameter `order`.

A. Sweeping the amount of Resources

In a first simulation the speedup was measured for different amounts of resources. The parameters used were `riceCoeffecent = 4`, `blockSize = 1024` and `order = 2`.

In both cases a speedup of 1.56 was achieved without addtional resources (one ALU and one memory). Using two additional ALUs and two additional memories gives a speedup of 1.66 for decoding and a speed up of 1.70 for encoding. Figure 5 and 6 also show the speedup for different amounts

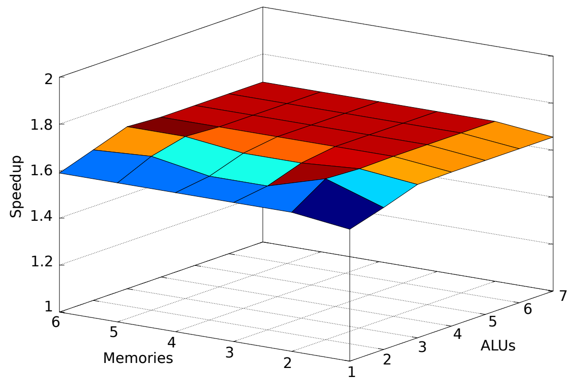

Fig. 5. Simulation Results for Decoding with different amount of resources

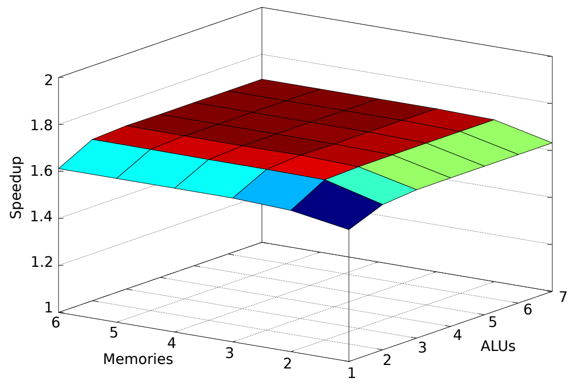

Fig. 6. Simulation Results for Encoding with different amount of resources

of resources and it is obvious that at this point a plateau is reached. Thus, adding further resources does not contribute any additional speedup. The reason for this is that optimizations like ILP or CSE are already exploited using three ALUs and three memories.

In Figure 5 some artefacts can be seen when using two ALUs. Adding memories decreases the speedup in some cases. This is due to the heuristic scheduling algorithms used in Synthilation. Hence, adding memories can lead to a suboptimal schedule which results in a slightly decreased speedup.

Apart from those artefacts using three ALUs and three memories is the optimal hardware configuration because it leads to a nearly optimal speedup with a minimum of additional hardware. The artefacts were neglected when taking this decision because for different ADPCM prediction orders those artefacts look different so this case does not give any information about the general case.

B. Sweeping the Order

In the second simulation the resources were fixed to three ALUs and three memories. The parameters `riceCoefficient = 4` and `blockSize = 1024` were fixed as well while the parameter `order` was swept from 2 to 14 to test the influence of the order on the speedup. Figure 7 show the results for decoding and encoding respectively.

It is obvious that the speedup increases monotonically with the order. The reason for the monotonous increase is, that for higher orders the number of loop iterations increases (for loops marked with **O**). As only loops are synthilated this

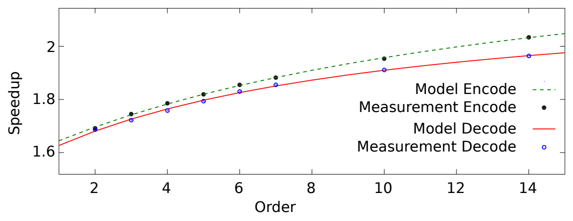

Fig. 7. Simulation Results for Encoding and Decoding with different orders

means that the percentage of executed instructions that can be synthilated increases. Thus, if the order increases, the speedup also increases. This implies that for large orders the speedup converges to a limiting value l.

This phenomenon can be mathematically modeled as follows. The execution time $T(o) = a + b + c(o)$ consists of one part a that will not be synthilated, one part b that will be synthilated but is independant of the order and the last part $c(o) = h \cdot o$ which will be synthilated and have a runtime that depends on the order linearly. This part consists of loops whose number of iterations is directly dependant on the order (denoted with **O** in Section IV). Nested loops may lead to a runtime that is quadratically or even cubically dependant on the order, but the experiment showed that the linear model is sufficient, as nested loops occur only in the beginning of blocks or during Gaussian elimination respectively. For the synthilated case the parts that will be synthilated are multiplied with speedup factors s_b and s_c: $T_{syn}(o) = a + s_b b + s_c c(o)$. So the speedup S for the order o is

$$S(o) = \frac{T(o)}{T_{syn}(o)} = \frac{a + b + ho}{a + s_b b + s_c ho} \tag{1}$$

This equation can be transformed to a rational function with three unknowns x_1, x_2 and x_3:

$$S(o) = \frac{1 + x_1 o}{x_2 + x_3 o} \tag{2}$$

We can use this equation and the measurements shown in Figure 7 to create an overdetermined system of linear equations $Ax = y$. We solved this system approximately with $x = A^+ y$ where A^+ is the pseudoinverse matrix of A. This leads to $x_{DEC} = [0.16 \quad 0.64 \quad 0.07]^T$ and $x_{ENC} = [0.11 \quad 0.63 \quad 0.04]^T$. From this we can calculate the boundary value $l = \lim_{o \to \infty} S(o) = \frac{x_1}{x_3} = \frac{1}{s_c}$. We get $l_{DEC} = 2.29$ and $l_{ENC} = 2.75$. This means that the speedup can never be higher than 2.29 for decode and it can never be higher than 2.75 for encode respectively.

This is not a global property of ADPCM but a property of the implementation we used in this work which is a generic code and was not optimized for runtime. Figure 7 shows that the model fits the measurement very well.

VI. Conclusion

In this contribution, we have shown a mechanism that accelerates the execution of Java bytecode on AMIDAR processors. We have evaluated this synthilation approach on a real world example. It is concerned with ADPCM, a lossless compression and decompression method for digital signals. We can achieve speedup factors of up to 2 using only minimal additional hardware for higher ADPCM prediction orders. Without these hardware extensions we still achieve speedups of 1.56. By sweeping the parameter space for additional resources we can identify the optimal number of resources for additional ALUs and scratchpad memories.

In future work, we will investigate the combination of synthilation with hardware synthesis. Also, we will evaluate other real life problems.

References

[1] S. Vassiliadis and D. Soudris, Eds., *Fine- and Coarse-Grain Reconfigurable Computing*. Springer, 2007.

[2] 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.

[3] S. Döbrich and C. Hochberger, “Effects of simplistic online synthesis in amidar processors,” in *ReConFig*, 2009, pp. 433–438.

[4] A. Koch and N. Kasprzyk, “High-level-language compilation for reconfigurable computers,” in *ReCoSoC*, 2005, pp. 1–8.

[5] F. Philipp and M. Glesner, “(geco)2: A graphical tool for the generation of configuration bitstreams for a smart sensor interface based on a coarse-grained dynamically reconfigurable architecture,” in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, Aug 2012, pp. 679–682.

[6] 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.

[7] ——, “ADRES: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix,” in *FPL*, 2003, pp. 61–70.

[8] S. Döbrich and C. Hochberger, “Low-complexity online synthesis for amidar processors,” *International Journal of Reconfigurable Computing - Selected Papers from ReconFig 2009 International Conference on Reconfigurable Computing and FPGAs (ReconFig 2009)*, vol. 2010, 2010.

[9] ——, “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.

[10] S. Microsystems, *picoJava-II Microarchitecture Guide*, 1999.

[11] L.-C. Chang, L.-R. Ton, M.-F. Kao, and C.-P. Chung, “Stack Operations Folding in Java Processors,” *IEEE Transactions on Computers and Digital Techniques*, vol. 145, no. 5, pp. 333 – 340, 1998.

[12] L.-R. Ton, L.-C. Chang, and C.-P. Chung, “Exploiting Java Bytecode Parallelism by Enhanced POC Folding Model (Research Note),” in *Euro-Par*. Springer, August 2000, pp. 994 – 997.

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

[14] ——, “The organic features of the AMIDAR class of processors,” in *ARCS*, 2005, pp. 154–166.

[15] ——, “Hardware based online profiling in AMIDAR processors,” in *IPDPS*, 2005, p. 144b.

[16] ——, “On the scope of hardware acceleration of reconfigurable processors in mobile devices,” in *HICSS*, 2005, p. 299.

[17] A. Engel and A. Koch, “Hardware-accelerated data compression in low-power wireless sensor networks,” in *Reconfigurable Computing: Architectures, Tools, and Applications*, ser. Lecture Notes in Computer Science, D. Goehringer, M. Santambrogio, J. Cardoso, and K. Bertels, Eds. Springer International Publishing, 2014, vol. 8405, pp. 167–178.

[18] CCITT, “Recommendation g.726,” International Telecommunication Union, Tech. Rep., 1990.

[19] K. Sayood, *Introduction to Data Compression (2Nd Ed.)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000.

[20] J. Hofmann, “Hardware accelerated data compression in wireless sensor network,” Bachelor Thesis, TU Darmstadt, Embedded Systems and Applications Group, 2012.