

On the Scope of Hardware Acceleration of Reconfigurable Processors in Mobile Devices

Stephan Gatzka and Christian Hochberger

Chair for Embedded Systems, TU Dresden

Email: [stephan.gatzka|christian.hochberger]@inf.tu-dresden.de

Abstract—Reconfigurable devices like Configurable Systems on a Chip (CSoCs) have the ability to exchange parts of hardware during runtime. Multimedia applications often require computing power and often can be accelerated by specific hardware circuits. In this contribution we present the effect of hardware acceleration in reconfigurable devices. We introduce a new model of adaptive processors and describe its basic principle of operation. The model allows runtime variations of the type and number of functional units (including application specific FUs) as well as variations of the communication structure. Then, we discuss our multimedia kernel Inverse Discrete Cosine Transformation (IDCT) and the ability of shifting different parts of the IDCT into hardware. Simulation results show that hardware acceleration of multimedia operations in an adaptive processor leads to significant performance gains and therefore reduces energy consumption.

I. INTRODUCTION

Configurable Systems on a Chip (CSoC) are becoming more and more important in the embedded systems and mobile devices market. The main reasons for their growing popularity can be found in their cost effectiveness and flexibility. Mask costs will be very high in the future due to the required high resolution[12]. CSoCs will be cost effective under these circumstances since they offer the possibility to implement multi protocol/multi standard systems with a single chip and thus can be produced in much larger quantities. They will be flexible since the reconfigurable part can be used to adapt the hardware to future requirements that are unknown at the time of the initial development.

With CSoCs it will be possible to implement new IO functionality and peripherals in an existing system. It will also be possible to have peripherals to increase the performance (e.g. crypto accelerators), but the processor core itself cannot be changed or enhanced. To overcome this problem, we have introduced the AMIDAR class of processors[5]. It is a novel model and architecture of processors that can be adapted during runtime to the requirements of an application.

The presentation of multimedia content becomes one of the major challenges in mobile devices. Presentation typically requires a transformation from an encoding in the frequency domain to the spatial domain. This transformation requires a high amount of computing power and thus is a major source of energy consumption in mobile devices. For these reasons we have chosen typical transformations as a benchmark application for our measurements.

Although C as a programming language still dominates the development of embedded software, the growing tendency to

use Java as a programming language for embedded systems makes this language an attractive object of study. Due to the code shipping abilities of Java[7], it is most likely, that especially systems programmed in Java will experience a shift in the requirements during their lifetime. Thus, we use the example of a Java bytecode processor based on this model to show the properties of the model. Additional advantages of the Java bytecode are discussed in section II-B.

Running Java bytecode on an AMIDAR processor offers the opportunity to introduce specialized hardware components that replace parts of the software and thereby enhance the performance of the system.

In this contribution we show the different granularities that are available for mapping software parts into hardware. In contrast to previous publications[10] we show that hardware mapping is not restricted to method level, but can also be applied to bytecode sequences or full classes.

A. Related Work

Hardware implementations of Java bytecode processors are available in a large number. Yet, to our knowledge only the JEM-II processor can be customized for the application requirements[1]. But in this case only new bytecodes can be introduced as microcode sequences.

Also, recently some work has been conducted to build customized accelerators to speed up the execution of Java bytecode[8]. In this case only a small part of the bytecode execution is implemented in hardware and the main execution is done on a conventional processor.

Other researchers have addressed (re)configurable processors in general. Some are just parameterized RISC cores[2], while others are truly reconfigurable. They typically depend on compile time analysis and generate a single datapath configuration for an application beforehand[4][3][9]. Very few processors are really reconfigured at runtime[11]. But even in this case, the configurations and the time of reconfiguration are defined at compile time.

To the best of our knowledge, there is no general model for an adaptive processor.

B. Paper Outline

In the following section we will give a short overview on the general model of an adaptive processor. In section III we will present the IDCT and variable bitlength decoding in different variants and their applicability for a mapping into hardware.

Experimental results for an adaptive Java bytecode processor using special functional units for IDCT and variable bitlength decoding are shown in section IV. Finally, a conclusion and an outlook onto future work are given.

II. MODEL

In this section we give a short overview how our model of an adaptive processor works in general. A much more detailed description can be found in [5] and [6]. Additional information is also available on www.amidar.de.

Figure 1 shows the basic structure of an adaptive processor.

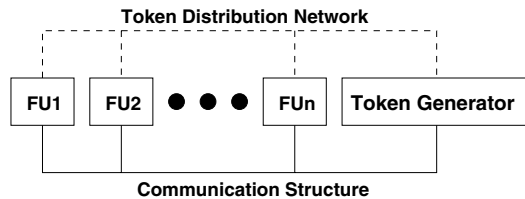


Fig. 1. General model

It consists of four main parts: a token generator, functional units (FU), a token distribution network and a communication structure. The token generator is a specialized functional unit, which is always required. Data is passed between the different FUs. Operation of the FUs is controlled by means of the tokens which are sent to the FUs from the token generator. Functional units can have a very wide range of meanings: ALUs, register files, program and/or data memory, specialized address calculation units, etc.

Data passed between FUs can have various meanings: Program information (instructions), address information, or application data.

A. Principle of Operation

Program information (i.e. the instructions) is sent to the token generator. Now, the token generator creates a set of tokens for this instruction and distributes them concurrently to the functional units. A functional unit begins the execution of a specific token as soon as the data ports have the data with the corresponding tag. Upon completion of an operation the result is sent to the destination port. An instruction is completed, when all the corresponding tokens are executed. To keep the processor executing instructions, one of the tokens must be responsible for sending a new instruction to the token generator.

This data driven approach has a number of advantages:

- It implies a maximum of parallelism, which is only limited by data dependencies.
- It does not rely on a particular timing of the FUs. Execution of an instruction will work, no matter how long a single FU needs to complete its token. Also, it is not necessary to know the structure of the communication network beforehand. Thus, it can be changed during runtime without the need to reconfigure the token generator.

- It allows overlapping execution of instructions, since the token generator can start distributing tokens for more than one instruction. It only has to increment the tag field of the token for each instruction to separate data belonging to different instructions.
- It allows the introduction of new FUs and instructions using these FUs. For this purpose a small part of the token generator must be reconfigurable to store the token set for new instructions and to attach new FUs to the token generator (which are not addressed by normal instructions).

Although, this model looks like a dataflow machine, it has to be noted, that it actually processes instructions by this scheme and not application data.

B. Adaptive Operations

Adaptivity in this model can be seen on two hierarchical levels. On the top level the available chipsize is partitioned into an area for communication infrastructure and an area for functional units. Most of the currently available reconfigurable devices will not fully support this type of adaptivity, since resources for communication may not be suitable for functional units and vice versa. Yet, the model should be general enough to capture these possibilities. On the lower hierarchical level we have adaptive operations that reconfigure each of the two main areas. Figure 2 illustrates the different adaptive operations of our model.

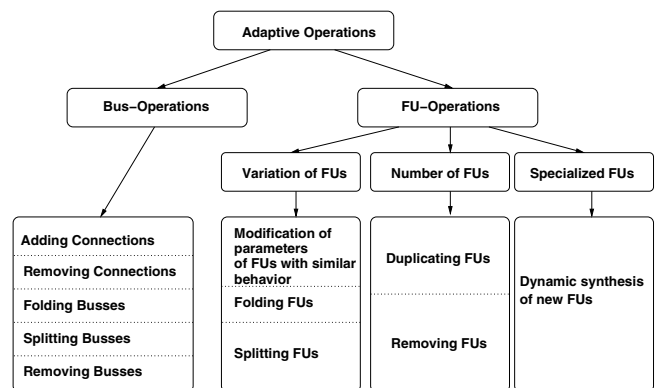


Fig. 2. Adaptive operations

Within the communication area several adaptive operations are possible:

- **Adding and removing connections.** If a functional unit has to send a data packet to another FU but is not connected to it, it is necessary to create a new connection.
- **Folding busses.** Two busses may be merged to a new bus, if there are only few collisions on both busses.
- **Splitting busses.** Buses with a high utilization and many delays can be split into two busses.
- **Removing busses.** Although this operation is already implied by folding of busses, it is useful as a separate operation, because folding of busses has a higher complexity than a simple remove.

The adaptivity of the communication structure is comparable with the dynamic reconfiguration of the forwarding unit of a superscalar microprocessor.

Within the functional unit area three different categories of adaptive operations can be applied:

- **Variation of FUs.** In this case variations of a certain FU may be available. From these variations the heuristics choose the most appropriate. This operation is fairly simple because it does not affect the token generator. Moreover, it is possible to split an FU into more specialized ones. For example, an ALU may be used for address calculations. This ALU may be split into an address calculation unit and a normal ALU. Folding of FUs into one FU is the complementary operation to FU splitting.
- **Increase and decrease the number of instances of an FU.** If the interval I of an FU can not be decreased by a more specialized version, it is possible to duplicate this FU (the interval is the time that has to be waited, before a new operation can be started). Token distribution must be adapted to this new situation which must also care about an equal utilization of the new FUs. Therefore, this adaptive operation is not as easy to implement as a simple FU exchange.
- **Addition of newly synthesized FUs.** It is also possible to identify heavily used instruction sequences and synthesize a new FU for such sequences. The instruction sequence is replaced by a new instruction and the token generator is updated with a token sequence for this new instruction. It may be applicable to synthesize complete methods or functions. The calling function can easily replace the calling code to access the new hardware. This is the most complicated adaptive operation, but promises the highest performance gain.

The addition of newly synthesized FUs requires a runtime analysis of the executed code. For this purpose Java bytecode is well suited, since it is specified on a rather high abstraction level and thus data and control flow can easily be extracted.

III. APPLICATIONS

In this section we show two applications which demonstrate that varying scopes of hardware mapping are required. Classically, only methods are mapped to hardware. In this section we discuss three scopes of hardware mapping for software components: complete classes, full methods and bytecode traces. The applications are taken from the multimedia domain. Typically, these applications would not be programmed in Java. Yet, the analysis given here can also be applied to the hardware mapping of other object oriented languages like C++ or C#. Also, the same choices for the granularity of a hardware mapping would apply to different application domains like control applications or interactive applications.

Multimedia content is often encoded in the frequency domain and must be transformed to the spatial domain. For this task the Inverse Discrete Cosine Transformation (IDCT) is often used. Prominent examples are JPEG pictures and

MPEG streams. Coefficients in the frequency domain are often encoded with variable bit length by means of a standardized huffmann tree. Most of the time of decoding an MPEG stream is spent in the IDCT and the variable bit length decoding.

A. IDCT

The next paragraph shows a highly optimized Java method which implements the IDCT of an 8x8 input matrix. It uses integer operations to simulate fixed-point arithmetics. The constant COSBITS defines the number of bits following the (virtual) point.

```
byte [] [] getTransformedInt1DFast () {
    byte [] [] res = new byte [8] [8];
    for (int k=0; k<8; k++) {
        int row [] = input [k];
        for (int i=0; i < 8; i++) {
            int x=0;
            int [] cos_row = cos_i_k [i];
            for (int l=0; l<8; l++) {
                x+= row [l] * cos_row [l];
            }
            z [k*8+i] = (x + (1<<COSBITS-1)) /
                ((1<<COSBITS+1));
        }
    }
    for (int i=0; i<8; i++) {
        for (int j=0; j < 8; j++) {
            int x=0;
            int ind=i;
            int [] cos_row = cos_i_k [j];
            for (int k=0; k<8; k++) {
                x+= z [ind] * cos_row [k];
                ind+=8;
            }
            res [j] [i] = (byte) ((x +
                (1<<COSBITS-1)) / (1<<COSBITS+1));
        }
    }
    return res;
}
```

The input matrix is stored in the `input` field, `cos_i_k` is the coefficient matrix. The array `z` is just a buffer array containing temporary results.

The IDCT consists of two main blocks of loops. The two blocks are nearly identical with one exception: the temporary buffer `z` is accessed consecutively in the first block (`z [k*8+i] = . . .`) but with a displacement of 8 in the second block. This has implications on the mapping into hardware. This concern will be discussed later in this section.

The innermost loop just accumulates the multiplication of elements of one row from the input matrix with elements of one row of the coefficient matrix. This is a simple MAC (Multiply Accumulate) instruction which is very common in DSP processors. It should be noted, that the result of one MAC instruction is not used in the multiplication of the following

	bytecodes	cycles
inner loop	122	488
middle loop	$122 \times 8 + 186 = 1162$	4648
outer loop	$1162 \times 8 + 2 = 9298$	37192

TABLE I
CALCULATION OF EXECUTION CYCLES

MAC instruction. This makes it very easy to pipeline the instructions of this loop. The middle loop just selects a new coefficient row, executes and stores the result of the innermost loop. The outermost loop of the first block just selects the next row of the input matrix, the outermost loop of the second block is only responsible for providing the correct index of the resulting matrix. We can see, that besides the innermost loops little or no computation is done in the IDCT.

Before we discuss the different mapping strategies, we will give a short analysis of the presented method from the bytecode view. First of all we will count the number of bytecodes for executing the IDCT transformation in Java. The innermost loop consists of 15 bytecodes. The loop will be executed eight times and there are two bytecodes overhead for the initialization of the loop. This accumulates to 122 bytecodes for this loop. The middle loop executes the inner loop 8 times and the selection of the new coefficient row and the storage of the results into the temporary buffer consumes 23 bytecodes for every loop pass. The initialization of this loop also requires two bytecodes. This sums up to 1162 bytecodes for the middle loop. The outermost loop executes the middle loop 8 times and the loop initialization again takes 2 bytecodes. This accumulates to 9298 bytecodes. Our experience with the AMIDAR bytecode processor suggests that on average the execution of one bytecode consumes 4 clock cycles¹. This value is slightly higher than the average number of clock cycles on a typical software JIT virtual machine. Table I summarizes the calculations. The numbers given here are only for the purpose of estimating the performance gain of different hardware mappings. Section IV shows actual measurements in our simulation environment.

A hardware mapping could be done on different levels: we can put only the innermost loop, the middle loop or the outermost in hardware. The next paragraphs will discuss the pros and cons of the three solutions.

The first solution is to put only the innermost loop into hardware. Therefore a new instruction would be introduced into the processor. This instruction would carry out the whole inner loop. Its parameters are the address of the coefficient row and the index into the cosine table (which is stored inside the FU). The calculation inside is a simple MAC instruction. No multiplication depends on previous results so it is very easy to pipeline this loop and the multiplier. The coefficients are read from the heap with a burst read operation. We can assume two

¹In previous publications, we have reported slightly smaller numbers. The increase is caused by rewriting token sequences for several bytecodes which result in a more relaxed timing and thus will lead to higher clock frequencies in a real hardware implementation.

cycles to fetch the arguments from the stack, eight cycles for the transfers of the element of the current coefficient matrix row² two more cycles for the pipeline setup and three cycles to store the result into the temporary buffer. This sums up to 15 cycles for the complete loop (execution time of the newly introduced instruction). The middle loop executes in 8×15 (innermost loop) cycles + 186 (8 times the loop overhead of 23 bytecodes) $\times 4 = 864$ cycles, the outer loop in 6912 cycles. This results in a performance gain of a factor of about 5.

If we map the middle loop into hardware we get the following numbers. Now we have 2 cycles to transfer the parameters (array base address and row number of the result), 8 cycles to read the start addresses of the inner arrays (rows) and 64 cycles to transfer the elements of the coefficient matrix, we assume 8×2 cycles for the pipeline setup and 8×3 cycles to transfer the results into the temporary buffer. This accumulates to 114 cycles for the middle loop and $8 \times 114 + 2 \times 4 = 920$ cycles for the outer loop. This leads to a performance gain of about 40.

We omit the calculation of the performance gain for the outer loop for two reasons. First of all the performance gain will not be very high compared to the hardware mapping of the middle loop. The main reason is that the loop overhead of the outer loop is very small (only loop initialization is done). Furthermore, the data access is much more complicated because there must be address calculation with two indices.

Mapping the complete method into hardware is even more complicated. We cannot use one MAC unit for calculation inside the two loop blocks, because the input of block two (temporary buffer z) depends on the results of block one. Buffer z must be filled completely by block one before block two is able to start its transformations. For an efficient pipelining we have to double the temporary buffer z and have to switch between them while executing the transformations. The performance benefit will not be very high as already discussed.

Comparing hardware effort and performance gain it is suggestive to map the middle loop into hardware. The hardware effort is small, only a MAC unit and some pipeline registers are required.

It is also possible to use this hardware unit for both loop blocks of the IDCT method. There is only one difference between both blocks. The first block accesses the input matrix and the result buffer consecutively, which results in simple burst transfers from or to memory. The second block accesses the temporary buffer with a displacement of 8 words, so simple burst transfers are not possible here. Two solutions are conceivable. The memory may have a special addressing mode which delivers the required data just like a simple burst. The other solution is that the new functional unit for the IDCT can generate the addresses for the required read operations. This

²In the Java bytecode processor discussed here, we assume separate memories for the method stack, operand stack, local variables, object heap and code memory. Java bytecode allows us to use several of them in parallel. Also, this architecture makes it easy to provide burst read operations for the object heap to access hole arrays of values.

allows the memory to effectively pipeline the read accesses.

Other hardware implementations are possible, of course. So it is possible to execute all calculations of the loops in parallel. For the inner loop we need 8 multipliers and an 8 input accumulator. This allows the execution of the inner loop in 1 or 2 cycles. The main disadvantages are the higher hardware effort and a much wider datapath to the hardware unit. The memory must be capable to feed the hardware unit with a sufficient rate. The performance gain is only about factor 6. The performance gain is much greater if we execute the middle loop parallel in hardware. The middle loop will be executed in 2 - 3 cycles (1-2 for the calculation and 1 cycle to store the result in the buffer). This leads to phenomenal performance gains at the expense of hardware effort. Now we need 64 parallel multipliers and eight 8-input accumulators. The datapath now has to transport 64 words at once to the new FU and 8 words from the FU to the memory. This seems not to be very realistic for a hardware realization.

B. Variable Bitlength Decoding

Multimedia data streams often contain information that has been transformed from the spatial domain to the frequency domain. The main purpose of this transformation is to achieve a more compact representation of the information. Typically, in the frequency domain a small number of different coefficients appear quite often and the remaining coefficients occur very rarely. This fact is used to construct Huffman trees that use a variable number of bits to encode a single coefficient. Coefficients that occur very often use few bits and coefficients that rarely occur use more bits. This minimizes the storage space required for the information.

Whereas in the IDCT it was neither feasible nor useful to implement the whole method as a functional unit, in the case of the variable bitlength decoding it is also not advisable to implement only the methods of the Huffman decoder class. Instead, it gives much higher performance gain to implement the whole Huffman decoder class as a functional unit.

The main method of the decoder is shown in figure 3.

```

public int[] nextWord() {
    int idx=0;
    int bit=0;
    while (idx != -1) {
        if (mask==0) {
            mask=(1<<31);
            curWord=src.nextWord();
        }
        bit=((curWord & mask)!=0) ? 1:0;
        mask>>=1;
        if (tab[idx][bit] == -1) {
            return tab[idx];
        } else {
            idx=tab[idx][bit];
        }
    }
}
    
```

Fig. 3. Main method of the Huffman decoder

This method references several fields of the instance of the decoder object. *tab* is a two dimensional array of integer values. It represents a logical tree that is used to decode the incoming bits. Each line of the array contains in its first and second element the pointer (i.e. index) to the next node. A -1 indicates a leaf element of the tree. Once the traversal of the tree reaches a leaf, this line of the array is returned to the caller. Different instances of the decoder use different trees. Also, *curWord* and *mask* are fields of the instance. In the following each traversal from one level of the Huffman tree to the next is called a *round*.

To evaluate the performance of different hardware implementations we executed the Huffman decoder on a sample input set (one frame of an MPEG video). Table II shows the different values that we gathered during the execution.

Total number of decoded words	7857
Total number of input words	1194
Total number of rounds	38208
Average number of rounds per decoded word	4.862
Bytecodes executed in nextWord()	1389571
Average bytecodes per round	36.369

TABLE II
STATISTICS OF THE HUFFMANN DECODER

Using the same assumptions as in the previous subsection, we can calculate the estimated number of clock cycles for one decoded word. Each round of the decoder takes 36.369 bytecodes per round on average. Each decoded word uses 4.862 rounds on average. Thus a decoded word requires 176.826 bytecodes on average. This results in an estimated number of 704.304 clock cycles per decoded word.

The usual approach would be to implement this method as a specialized functional unit. We would then introduce a new instruction that would trigger the execution of this new FU. The performance gain that can be achieved by this approach is

relatively small. Each invocation of the new instruction would start by fetching the object address from the operand stack. Then the instance fields `mask` and `curWord` need to be transferred from the heap memory to the FU. After this, each round would require yet another series of memory accesses to fetch the accessed lines of the array `tab` (at least 2 accesses for each round). After the final round the instance fields `mask` and `curWord` need to be written to the object heap and the result needs to be written to the operand stack. Table III shows the overall number of clock cycles for this kind of implementation.

Description	times	clocks	total
Fetch of the object address	1	1	1
Fetch of <code>mask</code> and <code>curWord</code>	2	1	2
Fetch of the following index (send address of <code>array+index</code> , wait for address of line send <code>line+bit</code> , wait for new index)	4.862	4	19.448
Write <code>mask</code> and <code>curWord</code> to heap	2	2	4
Transmission of the result	1	1	1
Overall average execution time			27.448

TABLE III
CYCLE BALANCE OF THE SIMPLE HUFFMANN DECODER

An alternative implementation could try to keep all the instance data inside of the FU. A corresponding circuit is shown in figure 4. The block in the upper left corner is a FIFO which is used to feed the input data stream to the decoder. The FIFO is filled from an array in memory. Every time a word is requested from the decoder, the decoder fetches on word of the bitstream from the array. The fetched address is only incremented, if the FIFO is not already full. In the other case, a word is fetched that is already at the end of the FIFO. Thus, the fetch is ignored by the hardware. Since a single encoded word always uses fewer bits than are fetched with one word of the array, the FIFO will be full most of the time. The remaining parts of the circuit are a straight forward implementation of the Java method.

At the beginning of each invocation the referred object address is transferred to the FU. The FU compares this address with the stored object address. Execution immediately starts if the addresses match. Otherwise, the current values of `mask` and `curWord` are transferred back to the object stored in the FU. Then the values of the new object address are fetched (`mask`, `curWord` and `tab`). Since a change of the decoder object typically only occurs two or three times during each frame of the movie, the overhead for this operation can be neglected.

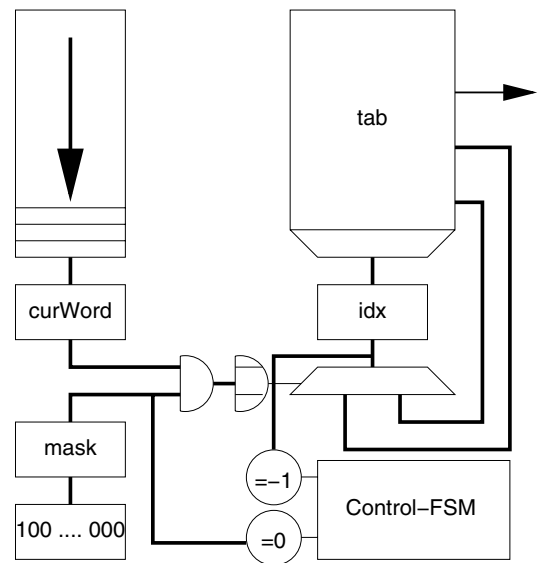


Fig. 4. A circuit implementing the huffmann decoder

Each round of the original algorithm takes one clock cycle. Thus, on average one fetch operation takes 4.86 clocks plus one clock for the transfer of the object address and plus one clock for the delivery of the result. The transmission of the address for the array fetch and the transmission of the fetched data can occur in parallel to the iterations of the decoder. Thus, this implementation requires an average total time of 6.86 clock cycles and is thus 4 times faster than the simple version and approx. 100 times faster than the software version.

IV. RESULTS

The results presented here are carried out by simulation using a special simulator that simulates the AMIDAR architecture on a cycle accurate basis. The simulator itself is written in Java and offers object models for various functional units and also offers simple to use abstract classes for functional units which make the introduction of new FUs very easy.

A. IDCT

Various scopes of hardware implementations of the IDCT method were discussed in section III-A. Two variations are of special interest for us: implementing only the most inner loop or implementing both inner loops. Both implementations could be done in parallel or as pipelined versions. The parallel versions will only show a considerable speedup, if parallel access to the memory is possible (which is not the case in the AMIDAR processor). Thus, we focus on the pipelined versions. Table IV shows the execution times of a single transformation. It should be noted, that the analysis in section III-A did only discuss one half of the whole transformation (the upper or lower block). Whereas the numbers given in table IV show the execution time of the full transformation.

For each of the variants we show the number of clock cycles for the upper and lower part of the transformation. The lower part requires significantly more clock cycles than the upper

	cycles/block	total cycles	speedup
pure software	38278		
	+ 45014	83292	1
mapping of inner loop	7438		
	+ 8922	16357	5.092
mapping of middle loop	920		
	+ 1132	2052	40.590

TABLE IV
SIMULATION RESULTS FOR THE IDCT

	total cycles	speedup
pure software	1212.864	1
mapping of method <code>nextWord()</code>	33.761	35.925
mapping of whole class	8.712	139.217

TABLE V
SIMULATION RESULTS FOR THE VARIABLE BITLENGTH DECODING

part. The main reason for this difference is the additional handling in the inner loop and the more complex addressing of the result array.

It turns out that even implementing the most inner loop already shows a speedup of 5. Although, implementing the two inner loops gives us a speedup of 40. Yet, it does not require more hardware for the computations, only a small amount of additional control hardware is required. Implementing the whole method is not an option for the IDCT method, since the resulting hardware effort would be too large.

B. Variable Bitlength Decoding

Table V shows the simulation results for three variants of the variable bitlength decoding. The numbers show the average number of clock cycles required to decode one word.

It turns out, that in this case our assumption of 4 clocks per bytecode is not fulfilled. In section III-B we estimated 706 clock cycles per word. Table V shows, that we actually need 1212 clocks per decoded word. This means that we need 6.8 clocks per bytecode on average. The main reason for this considerable deviation is the extremely high number of method invocation which take far more clock cycles than any other type of bytecode.

The simple version is already considerably faster and is more than 35 times faster than the software implementation. Yet, putting the whole class into hardware increases the efficiency by a factor of more than 4. The average execution time is decreased to 8.712 clock cycles per symbol (more than 139 times faster than the software implementation).

All hardware implementations are slightly slower than the estimated execution times in section III predicted. The additional clocks are typically introduced by resource conflicts, where different FUs would need the same resource in one clock cycle and thus their access had to be sequentialized.

IDCT and variable bitlength decoding are the largest part of the computational complexity of presenting an MPEG stream. Nevertheless, other parts still require a non negligible part of the processor performance (e.g. color space transformations,

motion vector computation, bitmap update). Thus, the overall speedup is not as high as indicated by the aforementioned FUs. Still, even an overall speedup of 2 already allows us to reduce the clock frequency by a factor of 2. As a consequence, we could also reduce the core voltage and in total we will gain an energy saving of a factor of far more than 4.

V. CONCLUSION

In this paper we have discussed different scopes of code that can be accelerated by application specific hardware. All variations can be very easily implemented in an AMIDAR type of processor. Yet, we have shown that it is typically not the best solution, to consider only methods for hardware implementation. In the case of the IDCT we can show, that implementing only a fraction of the method gives almost as good results as implementing the whole method, but needs significantly less hardware resources. In the case of the Huffman decoder, we have shown that implementing the whole class in hardware gives us a four times faster implementation, while not considerably enlarging the hardware effort.

VI. FUTURE WORK

The implementations that we presented here were hand crafted and thus could perform their tasks extremely efficient. Yet, the concept of an adaptive processor (which was the aim at developing the AMIDAR class of processors) requires that we implement such FUs automatically during runtime. Two main challenges in this context remain unresolved at the moment:

- 1) Identification of proper candidate bytecode traces, methods or classes. This requires some runtime profiling to find performance critical parts of the code. We are currently developing such techniques.
- 2) Synthesis of the corresponding FUs. This requires new synthesis algorithms, that can be used in resource constrained systems. The general idea in this case is to forfeit 10% of the quality and to gain a factor of 10 in memory consumption and runtime. We are also researching such algorithms.

REFERENCES

- [1] ajile Systems. aj-100 Datasheet, 2000. <http://www.ajile.com/>.
- [2] F. Campi, R. Canegallo, and R. Guerrieri. IP-Reusable 32-bit VLIW Risc Core. In *European Solid State Circuits Conference (ESSCIRC)*, pages 456–459, September 2001.
- [3] Y. Chou, P. Pillai, H. Schmit, and H. P. Shen. Piperech Implementation of the Instruction Path Coprocessor. In *Proceedings of the 33th Annual International Symposium on Microarchitecture*, pages 147–158, Monterey, December 2000.
- [4] C. Ebeling, D. C. Cronquist, and P. Franklin. Rapid - Reconfigurable Pipelined Datapath. In R. W. Hartenstein and M. Glesner, editors, *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pages 126–135, Berlin, 1996. Springer Verlag.
- [5] S. Gatzka and C. Hochberger. A new General Model for Adaptive Processors. In T. P. Plaks, editor, *Proceedings of the 2004 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'04)*, 2004.
- [6] S. Gatzka and C. Hochberger. The AMIDAR Class of Reconfigurable Processors. *Journal of Supercomputing*, to appear 2005.
- [7] S. Gatzka, C. Hochberger, and H. Kopp. Deployment of Middleware in Resource Constrained Embedded Systems. In *Tagungsband der GI/OCG-Jahrestagung 'Informatik 2001'*, pages 223–231, Wien (Österreich), September 2001. Österreichische Computer Gesellschaft.

- [8] Y. Ha, R. Hipik, S. Vernalde, V. Diederik, M. Engels, R. Lauwereins, and H. De Man. Adding Hardware Support to the HotSpot Virtual Machine for Domain Specific Applications. In M. Glesner, P. Zipf, and R. Michel, editors, *Field-Programmable Logic and Applications. Reconfigurable Computing Is Going Mainstream (LNCS 2438)*, pages 1135–1138, Berlin, Heidelberg, 2002. Springer.
- [9] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. Using the KressArray for Reconfigurable Computing. In J. Schewel, editor, *Configurable Computing: Technology and Applications, Proc. SPIE 3526*, pages 150–161, Bellingham, WA, 1998. SPIE – The International Society for Optical Engineering.
- [10] E. Lattanzi, A. Gayasen, M. Kandemir, V. Narayanan, L. Benini, and A. Bogliolo. Improving Java Performance Using Dynamic Method Migration on FPGAs. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, page 134b, 2004.
- [11] K. V. Palem, S. Talla, and P. W. Devaney. Adaptive Explicitly Parallel Instruction Computing. In J. Morris, editor, *Proceedings of the 4th Australasian Computer Architecture Conference*, Singapore, 1999. Springer Verlag.
- [12] SIA – Semiconductor Industry Association. The international technology roadmap for semiconductors. <http://www.itrs.net/>, 2001.