Optimizing coarse-grain reconfigurable hardware utilization through multiprocessing: An H.264/AVC decoder example

Andreas Kanstein[*a], Sebastian López Suárez[b], Bjorn De Sutter[c]
[a]Freescale Semiconducteurs SAS, 134 Av. du Général Eisenhower, F-31023 Toulouse Cedex 1
[b] Research Institute for Applied Microelectronics (IUMA), Department of Electronic Engineering and Control (DIEA), University of Las Palmas de Gran Canaria, E-35017 Gran Canaria
[c]IMEC vzw, Kapeldreef 75, B-3001 Leuven

ABSTRACT

Coarse-grained reconfigurable architectures offer high execution acceleration for code which has high instruction-level parallelism (ILP), typically for large kernels in DSP applications. However for applications with a larger part of control code and many smaller kernels, as present in modern video compression algorithms, the achievable acceleration through ILP is significantly reduced. We introduce a multi-processing extension to the coarse-grained reconfigurable architecture ADRES (Architecture for Dynamically Reconfigurable Embedded Systems) to deal with this kind of applications, by enabling it to exploit thread-level parallelism (TLP). This extension consists of a partitioning of an ADRES array into non-overlapping parts, where every partition can execute a processing thread independently, or a processing thread can be assigned to hierarchically combined partitions which provide a larger number of resources. Because the combining of partitions can be changed dynamically, this extension provides more flexibility than a multi-core approach. This paper discusses the architecture and an exploration into how to potentially partition a given array for executing an H.264/AVC baseline decoder.

Keywords: multi-processing, coarse-grain reconfigurable, H.264/AVC decoder, ILP, TLP

1. INTRODUCTION

Every new generation of video codecs places a greatly increased burden on the computational requirements of the multimedia sub-system of a portable device's System-on-a-Chip (SoC) [1]. This not only leads to an increase in silicon area occupied by the sub-system, but also to an increase in dynamic power consumption [2]. So far technology scaling helped to achieve the stringent silicon area and power consumption requirements. But the requirement of supporting multiple standards in a converged device, where those applications are only one of several use cases, make it attractive to look for efficiently programmable devices which offer multiple degrees of parallelism.

Here we present a multi-processing extension of the ADRES architecture. The ADRES (Architecture for Dynamically Reconfigurable Embedded Systems) [3] developed at IMEC represents a suitable option for multimedia applications, as it outperforms other optimized DSP processors [5]. Its differentiating feature is a coarse-grain reconfigurable array of processing elements and a C compiler which generates loop code with high instruction-level parallelism (ILP). MP-ADRES, the multi-processing extended ADRES architecture, provides the means to also exploit task-level parallelism (TLP) in a configurable way, by using partitions of the array as independent ADRES instances.

Because there are a large number of functional units (FUs) in an ADRES, the most appealing approach for increasing the application parallelism would be simultaneous multi-threading (SMT) [5]. ADRES however applies static scheduling due to low-power requirements, therefore such dynamic/speculative [6, 7] threading is inappropriate. For the supercomputing domain, a coarse-grained array architecture which employs static allocation and dynamic scheduling for dynamic multi-threading has been proposed and developed [8]. Our MP-ADRES threading approach instead identifies an application's thread-level parallelism based on the static analysis results. If properly reorganized and transformed at programming time, multiple kernels with weak dependency can be efficiently parallelized by the application designer.

[*] a.kanstein@freescale.com, http://www.freescale.com

Here we use the H.264/AVC decoder example to discuss the exploration phase, that is, the definition of processing threads and the mapping of threads to array partitions. We start from a mapping of an H.264/AVC baseline decoder to a single ADRES processor and analyze how well the functional blocks map to different ADRES instances which model different partitions of an MP-ADRES. We then devise different suitable groupings of functional units and assignments of those groups to partitions, as individual processing threads.
The paper is organized as follows. Section 2 introduce the MP-ADRES architecture after first discussing ADRES, and also explains ADRES programming. Section 3 presents the H.264/AVC implementation that we started from, and discusses the conceptual mapping to MP-ADRES. The concluding Section 4 reviews our results makes references to on-going work.

2. MULTIPROCESSING COARSE-GRAIN ARRAY

To introduce the multi-processing coarse-grain array architecture MP-ADRES, we first describe essential details of ADRES, then the architecture extensions for multi-processing, and finally the programming approach.

2.1. The ADRES Coarse-Grain Array

The ADRES coarse-grained array processor, as shown on the left hand in Fig. 1, consists of a VLIW processor linked to an array of functional units (FUs), enhanced with register files (RFs) and connected through routing resources like wires, multiplexors and busses. The right-hand part of Fig. 1 shows an exemplary node, a functional unit and its local register file.

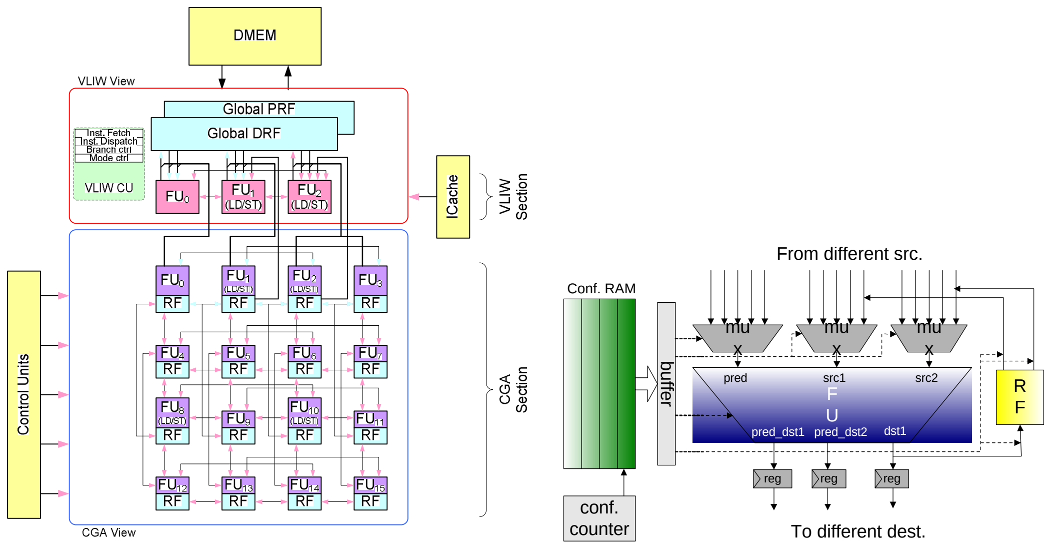

Fig. 1: Architecture of an ADRES coarse-grain reconfigurable array (left) and an array node (right). In array mode all CGA components are controlled through a very wide instruction word out of the configuration RAM, selected by a special counter. Multiple sequential words define a loop iteration, and the configuration memory typically has configurations for many loops.

An ADRES instance operates in either one of two modes, VLIW or Coarse-Grain Array (CGA). Sequential code is executed in VLIW mode, and loop code is executed in CGA mode. A compiler generates static schedules for both modes from C code, which will be explained later in this section.
ADRES is a templatized architecture that allows the construction of processors from an arbitrary number of function units, register files and interconnects. As shown in Fig. 1, an ADRES instance usually utilizes heterogeneous FUs, meaning that only some of them support multiplication operations and memory accesses. The results of the ADRES FU, shown on the right hand in Fig. 1, can be written to a local register file (RF), which is usually small and has fewer ports

than the shared RF, or routed directly to the inputs of other FUs. The multiplexors shown in Fig. 1 are used for routing data from different sources, while the configuration RAM acts as an instruction memory to control these components. It stores a number of configuration contexts locally, which are loaded on a cycle-by-cycle basis. In addition, ADRES provides predicates and rotating register files to let the compiler remove the control flow inside loops and efficiently implement prologues and epilogues of SW pipelined loops with conditional execution.
The following instruction set architecture details are necessary to understand ADRES performance data.

- The VLIW ABI and instruction set mostly follows the one given by IMPACT's L-code (see the sub-section on programming). The instructions are encoded in 32 bits with 2 source operands and one destination operand; only the store instructions have 3 source operands and no destination operand. The encoding supports up to 32 predicate registers and 64 data registers. Immediates are encoded within the instruction word.
- The VLIW pipeline has at least 5 stages, including two fetch stages. More complex instructions like multiplication take at least one extra cycle to complete. Load instructions take 3 extra cycles, and when the memory queues are enabled, 5 extra cycles. The memory queues are used in CGA mode to mitigate bank access conflicts in the data memory port, and thereby help to reduce the number of memory access stalls.
- Two branch delay slots may be used to mitigate the impact of code flow changes.
- ADRES is not designed to be programmed on the assembly-code level, therefore no HW-interlocking is implemented, and no side effects like automated saturation or instructions that change the contents of more than one register are supported.
- We have implemented a couple of special multimedia instructions on ADRES, like clipping, saturation, shift and round, INNERSUM4 and SUBABS4 for SIMD execution of, for example, calculating the sum of absolute differences, and other SIMD instructions. These instructions are supported, in our compiler, as intrinsic functions.

2.2. Multi-Processing ADRES

The multi-processing architecture is based on a partitioning of an ADRES instance into non-overlapping parts which include functional units, register files and routing elements. Each partition has its own simple controller for VLIW and CGA mode execution. Partitions can be combined hierarchically to form larger array instances, in which case their controllers will be linked to operate synchronously. There may of course be connections between components of different partitions which will only be used when those partitions are combined. The splitting of a partition in sub-partitions happens dynamically, which is adding flexibility by letting the programmer choose the partition size dependent on the kernels.
Every partition has at least one VLIW FU and global data and predicate register files, because control code always executes in VLIW mode. Since VLIW units are always defined as being in the first row of the array in our current architecture template, some partitions have a somewhat unusual shape, like two FUs in the first row combined with a 3x4 array of FUs, which we call 2p3x4. The global register file is clustered to provide each partition with its own pair of global register files. Figure 2 shows the MP-ADRES which we use in our example. Each global register file (the DRFs and PRFs) contains 32 data or 32 predicate registers, respectively.
Figure 3 shows all possible compilation targets, that is, partitions and combinations of partitions. Of course, an unused partition of combination of partitions will be clock gated, and this has to be considered when evaluating options for grouping functions into threads and mapping threads to partitions.

2.3. Programming

The C compiler for ADRES is called DRESC (Dynamically Reconfigurable Embedded Systems Compiler) [9]. It will take the architecture template file, a compilation parameters file, and the C code to generate DRE-code, which is assembly-level code for ADRES. Figure 4 shows a diagram with DRESC components. The first step in the programming flow is the code parsing and high-level optimizations for which we are using IMPACT [10], a compiler framework for VLIW and EPIC architectures. DRESC then compiles the output, called L-code, to VLIW or CGA code, thereby automatically inserting code for transferring data between the two. It has to be pointed out that modulo scheduling is an essential technique for obtaining enough independent instructions for achieving high ILP.
The DRE-code is input code for our assembler and linker, to generate binary implementation code, but it is also the input for the ADRES simulators, which are being used for generating profiling data. At design time, the profiling data is used for optimizing architecture instances, which is also called architecture exploration. At programming time, the

profiling data is used for providing information about which part of the code has to be further optimized. The source-level optimizations required to obtain good CGA implementations are discussed in the following section.

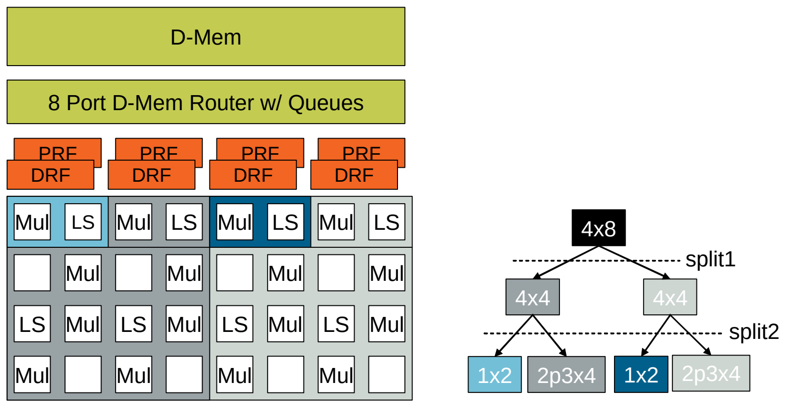

Fig.2: Multi-processing ADRES instance with a 4x8 array partitioned, at lowest level, into two 1x2 arrays and two 2p3x4 (2 plus 3x4) arrays. The right-hand graph shows how the array is hierarchically partitioned first into two 4x4 arrays and then into the lowest-level ones. The left-hand figure also shows the clustering of the global register files and how multiplication (Mul) and load-store (LS) units are distributed over the partitions.

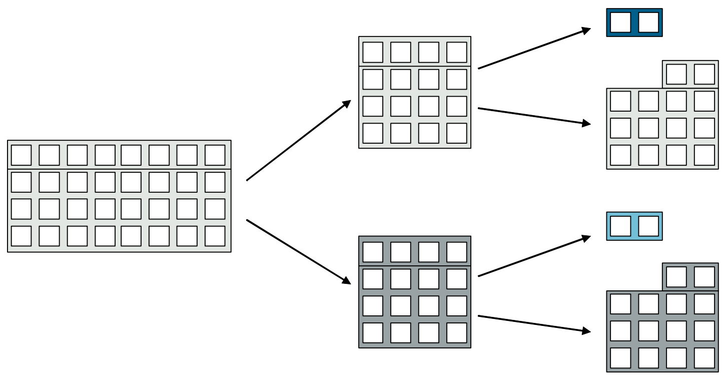

Fig.3: Hierarchically splitting the MP-ADRES of Fig. 2 into parallel processing instances. A 4x8 instance can be split into two 4x4 instances, and each can be sub-divided into a 1x2 and a 2p3x4 (2 plus 3x4).

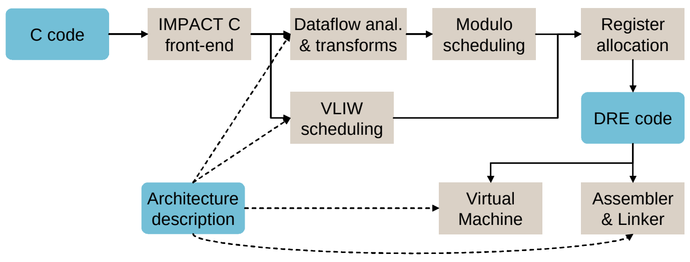

Fig.4: DRESC components. The architecture description is created at design time to specify an ADRES instance. DRE code is equivalent to assembly code. The Virtual Machine is used for cycle-accurate assembly code simulation.

For scheduling code for MP-ADRES, the template file has been enhanced to include information about which architecture resource belongs to what partition, and how partitions are grouped to form a hierarchy of partitions. Given this information, the CGA scheduler can map a loop to any partition that is part of the hierarchy. Furthermore, the VLIW scheduler now also supports clustered register files, which is a requirement for efficiently utilizing the global data register files when partitioning the ADRES. In the first compiler for MP-ADRES, the programmer has to assign functions to partitions and insert instructions for splitting and merging the execution flow.

3. EXAMPLE: H.264/AVC DECODER

The ITU-T H.264, also known as MPEG-4 (Part 10) Advanced Video Coding and therefore also called H.264/AVC in short, represents the latest evolution of video codecs [11]. Like with every video codec evolution step, encoding quality for a given bit rate is improved, at the expense of computational complexity. Baseline coding features include: in-loop signal-adaptive deblocking filter, quarter-pixel resolution motion vectors, multiple prediction block sizes from 16x16 down to 4x4, 13 intra-prediction modes, also down to a 4x4 block size, and a 4x4 integer transform. Especially the many block sizes complicate parallel processing, because multiple state machines are needed to implement the decisions related to block size and mode. Looking at the decoder from the perspective of decoding a sequence of macroblocks, the processing flow becomes mostly non-uniform.
The functional block diagram of a generic hybrid video decoder based on the H.264 standard is shown in Fig. 5. As it can be observed from this figure, the incoming video bitstream is stored in a memory buffer in order to be parsed and decoded by the entropy decoding stage. The syntax elements obtained after this process for each macroblock (16x16 luminance pixels and two blocks of 8x8 chrominance pixels) are demultiplexed and sent to the different functional kernels involved in the decoding process. In particular, the syntax elements related with the coding of the luminance and chrominance residual samples of the current macroblock (MB) are re-ordered by following a typical inverse scan procedure and passed to the inverse transform kernel. In parallel, a predictor is composed from previously decoded pixels in the same frame (intra coded macroblocks) or from pixels pointed by the received motion vectors belonging to frames previously decoded (inter coded macroblocks) depending on the information stored in the MB layer of the received bitstream. By adding the inverse transformed residual samples to the predictor selected and filtering the result in order to reduce the presence of annoying blocking artifacts, the original macroblock is recovered with minimal quality losses.

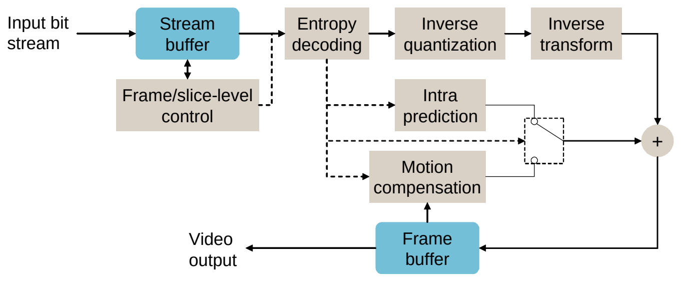

Fig.5: H.264/AVC decoder block diagram.

Important kernels for acceleration are in the motion compensation, in the deblocking filter, the inverse transform, and some data handling functions like memset. The following discusses the implementation of the decoder.

3.1. Baseline: Single-threaded FPGA Implementation

Our code is derived from the H.264/AVC decoder implementation in libavcodec [12]. We have created an FPGA implementation in which the decoder is running on a single ADRES, to create a prototype demonstrating stand-alone H.264/AVC baseline decoding at CIF resolution, 30fps, in 50MHz. At first the original code had to be prepared for working in an embedded processor, which involved reducing the code and data sizes through code pruning, removing the dependencies on standard C functions, and replacing 64 bit and unaligned data accesses.
Furthermore we had to adapt the code to the features and restrictions of our core, which means the insertion of intrinsic functions, converting all multiplications to 16x16 bit type multiplications, removing divisions and modulo operations,

and replacing floating-point data and operations. For the CGA mapping we also applied source-level code optimizations like loop unrolling, merging and coalescing.
Loop coalescing is the combination of two nested loops in one single loop. In the coalesced loop, the number of iterations is bigger than in the original inner loop, and hence switches between VLIW and CGA mode can be saved, as well as executing code of the outer loop in the less efficient VLIW mode. Loop unrolling typically increases the number of operations per iteration. Thus, the compiler has more operations available to schedule in the CGA mode, thus allowing him to exploit the offered ILP optimally.
To select the code to be mapped onto CGA mode, we used profiling information. First, we identified the hot code, and once we had identified this code, we studied it in detail to see which loops were potential candidates to be executed in CGA mode. In total, 28 out of 222 procedures were chosen to have their loops mapped onto CGA mode, for a total of 50 loops. 6 loops originate from the deblocking filter, 31 originate from the motion compensation, 5 from the inverse transformation, 3 from the decoder control part, and finally 5 from memory copy and initialization operations.

3.2. Parallelization Concept

For parallelizing the decoder we first grouped procedures into function groups according to the functional block diagram of the decoder, as shown in Fig. 5. Certain functions could not be separated, like the entropy decode, inverse scan and dequantization. However, because of data dependencies it would make little sense to assign these to different processing threads. Also, we have grouped all kernels together which implement memory management functions (memcpy, memset, and some functions for handling temporary buffers). We then compiled and profiled the code using different ADRES instances which represent partitions and grouped partitions from our exemplary MP-ADRES architecture, shown in Fig. 3. We then calculated the average number of cycles required for processing a macroblock in each of the functional groups. The results are shown in Table 1. Columns labeled 1x2 and 1x4 represent performance data for a two- and four-issue VLIW, respectively.

Table 1: Performance data for different partition sizes, in cycles per macroblock per function group, for the News CIF 30fps sequence encoded with 256kbps. Numbers are set in italics for function groups which execute only in VLIW mode. Numbers in brackets are not used but are provided for comparing VLIW with CGA mode execution.

Function Group	**1x2**	**1x4**	**2p3x4**	**4x4**
Frame/slice level control (FLC)	*677*	*410*		
Entropy decode and dequantization (ED)	*1510*	*1046*		
Inverse transform (IT)	*(762)*	*(630)*	262	223
Motion compensation kernels (MC)	*(2780)*	*(1539)*	586	470
Motion compensation control (MCC)	*220*	*140*		
Decode skipped MB (SMB)	*353*	*242*		
Memory management kernels (MM)	*(683)*	*(520)*	292	235
Deblocking filter (DBF)	*(6330)*	*(3348)*	1443	803

The reason for presenting performance data in the form of cycles/MB is that this not only simplifies the scaling of performance data, but also the definition of performance budgets per processing element, when parallelizing the code execution. For example, if the goal would be to achieve 30 frames per second SD (720x480) resolution decoding at 66MHz (half the clock rate of a typical SDRAM interface), the cycle budget would be 1629 cycles/MB.
We can now evaluate different multi-threading concepts based on the data given in Table 1. The results are given in Table 2. For each partitioning option we have a certain number of partitions: we can either use two 4x4 partitions, or one 4x4, one 1x2 and one 2p3x4 partition, or two 1x2 and two 2p3x4 partitions.
Then, functional groups are selected and assigned to partitions, based on what functional processing should be grouped together for data dependency reasons, and on the achieved loading of the partition. We then add the performance data per partition and determine the maximum processing load, which is highlighted as a boldface number.
It has to be noted that the partitioning concept so far ignores the overheads associated with functional pipelining: thread synchronization, data dependencies, and managing data buffers.
From the results in Table 2, partitioning options B and C are most interesting. From the two 4x4 partitions used in option B, one partition is executing in VLIW mode only, which means that its array units would remain in a low power mode in this application. Not surprisingly, the partitioning which uses the highest number of partitions, option C, offers

the highest expected performance. Even when taking multi-threading overhead into account, it seems possible to implement an SD decoder at 66MHz when using this partitioning of decoder functionality.

Table 2: Partitioning options and assignments of functionality, based on the profiling data provided in Table 1.

Option	**Partition**	**Assigned Function Groups**	**Cycles / MB per Partition**
A	4x4 or 1x4	FLC, DBF, MM, SMB	*410*+803+235+*242*=1690
	4x4 or 1x4	ED, MCC, MC, IT	1046+140+470+223=**1879**
B	4x4 or 1x4	DBF, MM, MCC, MC, IT	803+235+*140*+470+223=**1871**
	1x4	FLC, ED, SMB	*410+1046+242*=1698
C	1x2	FLC, SMB	*677+353*=1030
	2p3x4	DBF	1443
	1x2	ED	***1510***
	2p3x4 or 1x2	MCC, MC, MM, IT	*220*+586+292+262=1360
D	4x4 or 1x4	DBF, MCC, MC	803+*140*+470=1413
	1x2	ED	*1510*
	2p3x4 or 1x2	FLC, SMB, MM, IT	*677+353*+292+262=**1584**

Some data points in Table 1 seem implausible, like the big difference in performance of the deblocking filter when mapped onto the 2p3x4 partition instead of the 4x4 partition. One reason could be that the 2p3x4 partition has only 3 data memory ports (instead of 4), another could be that only half the number of registers are available in the global register files. The generated schedules need to be analyzed more closely to determine how the scheduling can be improved for the smaller partition.

4. CONCLUSIONS

This paper presents the multi-processing architecture extension of the ADRES coarse-grain reconfigurable array, which has been motivated by the fact that only exploiting high instruction-level parallelism for loop code will not lead to a satisfying implementation of advanced video codecs. The MP-ADRES offers a partitioning of the array which can be used in a rather flexible manner to implement a multi-threaded version of the application code. We explored different partitioning concepts and showed that the flexible grouping of partitions enable different implementation optimizations. So far the parallelization overheads due to communication and synchronization have been ignored, but we are currently working on the parallelization of the code so that we can, as a next step, update our partitioning concepts with improved performance data.

We have already shown, in an implementation study [13], how the architecture can be implemented in detail, and the compiler is currently being updated to support the creation of multi-threaded code. The architecture and compiler work is linked to IMEC's on-going work on multi-processor SoC (MPSoC) design tools [14].

Our current experiences indicate that some dedicated HW acceleration is required for the efficient implementation of advanced video codecs, especially for higher resolutions. For example, the obvious data dependencies in the serial processing of the input stream require a high performance implementation of these special operations, and the amount of control code in the deblocking filter result in a sub-optimal data-parallel implementation which uses a large amount of predicated instructions. Future definitions of video codecs should take the trend towards programmable multi-processing implementations into account when evaluating tools for enhanced compression.

REFERENCES

1. M. Horowitz, A. Joch, F. Kossentini, and A. Hallapuro, "H.264/AVC baseline profile decoder complexity analysis," IEEE Trans. Circuits and Systems for Video Technology, Volume 13, pp. 704-716, July 2003.
2. O. Silven, T. Rintaluoma, and K. Jyrkkä, "Implementing energy efficient embedded multimedia," Multimedia on Mobile Devices II. Edited by Creutzburg, Reiner; Takala, Jarmo H.; Chen, Chang Wen. Proceedings of the SPIE, Volume 6074, pp. 61-70 (2006).

3. B. Mei, S. Vernalde, D. Verkest, H. D. Man and R. Lauwereins “ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix”, Int. Conf. Field Programmable Technology, Hong Kong, Dec. 2002, pp. 166-173.
4. M. Berekovic, A. Kanstein, D. Desmet, A. Bartic, B. Mei and J.Y. Mignolet, “Mapping of Video Compression Algorithms on the ADRES Coarse-Grain Reconfigurable Array”, Workshop on Multimedia and Stream Processors’05, Barcelona, November 12, 2005.
5. Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, Dean M. Tullsen, “Simultaneous Multithreading: A Platform for Next-Generation Processors,” IEEE Micro 17, Sept.-Oct. 1997, pp. 12 - 19
6. H. Akkary, M.A. Driscoll, “A dynamic multithreading processor,” 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-31), 30 Nov.-2 Dec. 1998, pp. 226 - 236
7. E. Ozer, T.M. Conte, “High-performance and low-cost dual-thread VLIW processor using Weld architecture paradigm,” IEEE Transactions on Parallel and Distributed Systems 16, Dec. 2005, pp. 1132 - 1142
8. D. Burger, S. W. Keckler, K. S. McKinley, et al., “Scaling to the end of Silicon with EDGE architectures,” IEEE Computer 37, no. 7, 2004, pp. 44 – 55
9. B. Mei, S. Vernalde, D. Verkest, H. De Man, R. Lauwereins, “Exploiting Loop-Level Parallelism on Coarse-Grained Reconfigurable Architectures Using Modulo Scheduling”, *Proc of DATE 2003*, Munich, March, 2003
10. The IMPACT group, http://www.crhc.uiuc.edu/impact.
11. International Telecommunication Union/ITU Telecommunication Sector, "ITU-T Recommendation H.264: Advanced video coding for generic audiovisual services,” Geneva, Switzerland, March 2005.
12. libavcodec is distributed as part of Ffmpeg: http://ffmpeg.mplayerhq.hu
13. Kehuai Wu, Andreas Kanstein, Jan Madsen, and Mladen Berekovic, “MT-ADRES: Multithreading on Coarse-Grained Reconfigurable Architecture,” Proc. International Workshop on Applied Reconfigurable Computing (ARC), Mangaratiba, Rio de Janeiro, Brazil, March 27-29, 2007
14. IMEC vzw MPSoC Activity, http://www.imec.be/design/mpsoc/.