The Molen Media Processor: Design and Evaluation

Georgi Kuzmanov Georgi Gaydadjiev Stamatis Vassiliadis

Computer Engineering, EEMCS, Delft University of Technology,
P.O. Box 5031, 2600 GA Delft, The Netherlands
http://ce.et.tudelft.nl
{G.Kuzmanov, G.N.Caydadjiev, S.Vassiliadis}@EWI.TUDelft.NL

ABSTRACT

We present a fully operational prototype of the Molen reconfigurable processor based on the tightly coupled co-processor architectural paradigm. Within the Molen concept, a general purpose core processor controls the execution and reconfiguration of a reconfigurable co-processor, tuning the latter to various application specific algorithms. An ISA extension of only 4 instructions supports an arbitrary number of application specific functionalities running on the reconfigurable processor. The design is implemented on the Xilinx Virtex II Pro™ technology and is modular. For the experiments, we consider three media applications: MJPEG, MPEG-2, and MPEG-4. Experimental data suggest two orders of magnitude kernel speedups, approaching 98% of the theoretical maximum speedups at the application level. The Molen infrastructure consumes trivial hardware resources. Its hardware-efficient FPGA implementation leaves 98% of the considered xc2vp20 chip area available for reconfigurable implementations of user designs.

1. INTRODUCTION

Reconfigurable hardware coexisting with a core general-purpose processor (GPP) has been considered by a number of researchers as a good candidate for speeding up computationally intensive and performance critical applications. For the description of most of the existing reconfigurable proposals, the interested reader is referred to two review/classification articles [1, 2]. However, current reconfigurable computing proposals, where the possibility exists to combine general-purpose computing with reconfigurable fabric, fall short of expectation because of the following shortcomings:

- **Opcode space explosion**: For reconfigurable fabric, a common approach (e.g., [3], [4], [5]) is to introduce a new instruction for each portion of application mapped on the field-programmable gate array (FPGA). The consequence is the limitation of the number of operations implemented on the FPGA, due to the limitation of the opcode space. Furthermore, this results in ad hoc instruction set architecture (ISA) extensions which **excludes compatibility**.
- **No modularity**: Each approach has a specific definition and implementation bounded for a specific reconfigurable technology and design. Consequently, the applications cannot be (easily) ported to a new reconfigurable platform. Further, there are no mechanisms allowing reconfigurable implementation to be developed separately and ported transparently, as indicated in [6].
- **Limitation of the number of parameters**: In a number of approaches, the operations mapped on an FPGA can only have a small number of input and output parameters. (e.g., [7], [8]).

The Molen processor paradigm [9, 10] addresses and solves the shortcomings of the current reconfigurable proposals discussed above. The Molen processor is established on the basis of the tightly coupled co-processor architectural paradigm. The prototype design, presented hereafter, is implemented on the Xilinx Virtex II Pro™ technology. Its experimental evaluation with MJPEG, MPEG-2, and MPEG4 suggests the following advantages of the proposal:

- Improved performance of various media applications:
  - two orders of magnitude media kernel speedups;
  - 2X-3X speedup at the application level;
  - 98% of the maximum attainable speedup achieved;
  - no performance penalties for pure software executions.
- Efficient hardware utilization:
  - trivial hardware costs for the Molen infrastructure (below 1% of the considered xc2vp20 FPGA chip);
  - 98% of the FPGA chip (xc2vp20) available for implementations of specific functionalities by users.
- Compact PowerPC ISA extension: only 4 new instructions support an arbitrary number of user-defined functionalities.
- Modularity - a reconfigurable user design can be easily ported to support per application functionalities. A clearly defined and simple user design interface has been implemented.
- Arbitrary number of input-output parameters can be utilized for the application specific reconfigurable functions

The remainder of this paper is organized as follows. In Section 2, we briefly present the background on the Molen processor. Section 3 describes the design considerations employed in the particular Molen Virtex II Pro prototype presented. Our performance evaluation methodology is explained, followed by experimental results in Section 4. Finally, we conclude the presentation in Section 5.

2. THE MOLEN BACKGROUND

In this section, we present the general concept of transforming an existing program to one that can be executed on the Molen reconfigurable computing platform and some brief background on the Molen processor itself.

General approach: The conceptual view of how program P (intended to execute only on the GPP) is transformed into program P' (executing on both the GPP core and the reconfigurable hardware) is depicted in Figure 1. The steps involved in this transformation are the following:

1. **Identify** pieces of software code "α" in program P to be mapped in reconfigurable hardware
2. **Design** a hardware unit performing the functionality of program kernel "α" and map it onto reconfigurable hardware.

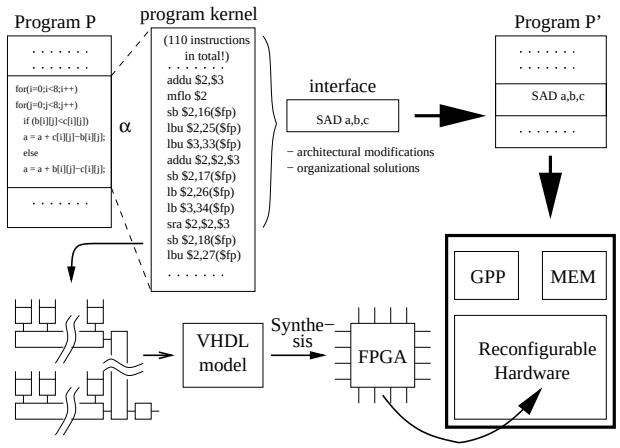

Figure 1: The general Molen approach: a program transformation example.

3. **Eliminate** the identified code "α" from program P. Insert an equivalent code A (e.g., SAD a,b,c), which calls the hardware through a preestablished SW/HW calling interface. This interface comprises:
   - Parameters and results for communication between the GPP and the reconfigurable processor.
   - Configuration code, inserted to configure the hardware.
   - Emulation code, used to perform the functionality of the hardware accelerated kernel "α".
4. **Compile and execute** program P' with original code plus code having functionality A (equivalent to "α", i.e., SAD a,b,c) on the GPP/reconfigurable processor.

It is noted that the only constraint on "α" is its implementability, which possibly implies complex hardware. Due to the complexity of this hardware, the microarchitecture may have to support emulation [11], which in turn requires the utilization of microcode. This reconfigurable microcode is termed as $\rho\mu$-code and it is different from the traditional microcode. The difference is that such microcode does not execute on fixed hardware facilities. It operates on facilities that the $\rho\mu$-code itself "designs" to operate upon.

Processor organization: The two main components in the Molen machine organization (depicted in Figure 2) are the *Core Processor*, which is a general-purpose processor (GPP), and the *Reconfigurable Processor* (RP). The ARBITER performs a partial decoding on the instructions in order to determine where they should be issued. Instructions implemented in fixed hardware are issued to the GPP. Instructions for custom execution are redirected to the RP. Data transfers from(to) the main memory are handled by the *Data Load/Store* unit. The *Data Memory MUX/DEMUX* unit is responsible for distributing data between either the reconfigurable or the core processor. The reconfigurable processor consists of the *reconfigurable microcode* ($\rho\mu$-code) unit and the *custom computing unit* (CCU). The CCU consists of reconfigurable hardware and memory, intended to support additional and future functionalities that are not implemented in the core processor. Pieces of application code can be implemented on the CCU in order to speed up the overall execution of the application. A clear distinction exists between code that is executed on the RP and code that is executed on the GPP. The parameter and result passing between the RP targeted code and the remainder application code is performed utilizing the *exchange registers* (XREGs), depicted in Figure 2.

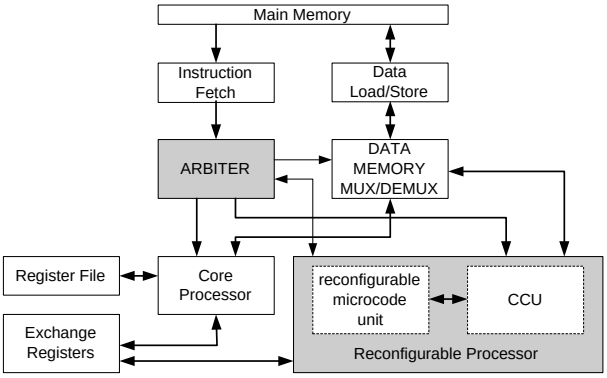

Figure 2: The Molen machine organization

Polymorphic operations: An operation, executed by the RP, is divided into two distinct phases: *set* and *execute*. The set phase is responsible for reconfiguring the CCU for the operation. In the execute phase, the actual execution of the operations is performed. No specific instructions are associated with specific operations to configure and execute on the CCU. Instead, pointers to *reconfigurable microcode* ($\rho\mu$-code) are utilized. The $\rho\mu$-code emulates both the configuration and the execution of CCU implementations resulting in two types of microcode: 1) reconfiguration microcode that controls the CCU configuration; and 2) execution microcode that controls the execution of the configured CCU implementation.

The complete polymorphic instruction set: The complete list of the eight instructions, supporting the Molen paradigm (for details see [12]) is denoted as polymorphic instruction set architecture (πISA). These instructions are:
1) partial set (p-**set** $<address>$) performs common and frequently used configurations;
2) complete set (c-**set** $<address>$) *completes* the CCU's configuration to perform less frequent functions;
3) **execute** $<address>$: controls the execution of the operations on the CCU configured by the set instructions;
4) **set prefetch** $<address>$, and 5) **execute prefetch**:
prefetch the needed microcodes responsible for CCU reconfigurations and executions into a local on-chip storage (the $\rho\mu$-code unit);
6) **break**: synchronizes parallel executions on the RP and the GPP;
7) **movtx** $\text{XREG}_a \leftarrow \text{R}_b$, and 8) **movfx** $\text{R}_a \leftarrow \text{XREG}_b$:
move the content of general-purpose register R_b to/from XREG_a.

The $<address>$ field denotes the location of the reconfigurable microcode responsible for the configuration and the execution. In the prototype processor, described hereafter, we have implemented a minimal πISA, comprising c-**set**, **execute**, **movtx**, and **movfx**.

3. DESIGN DESCRIPTION

In this section, we describe the particular design considerations regarding the separate parts of the Molen Virtex II Pro prototype. Our experiments have been carried out on an Alpha-Data development board ADM-XPL (`http://www.alpha-data.com/`) equipped with the Xilinx Virtex II Pro chip xc2vp20-5 [13]. The mounted FPGA chip is an engineering silicon with vendor recommended PowerPC clock frequency of 250 MHz. We have described the Molen organization in VHDL. The Xilinx standard development tools embedded in the ISE 5.2. SP3 have been utilized both for the synthesis (with the Xilinx XST tool) and for the FPGA mapping. Behavioral hardware simulations have been performed with ModelSim SE 5.7c. The programs have been compiled with GCC.

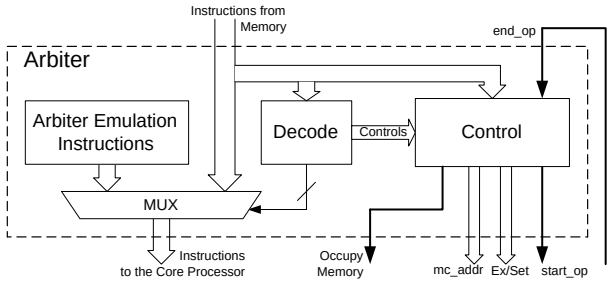

Figure 3: General organization of the arbiter.

3.1 The arbiter

The arbiter performs partial decoding on the instructions in order to determine where they should be issued. In Figure 3, a general view of the implemented arbiter organization is presented.

Software considerations: In the prototype design, the core processor is driven into a wait state, while a reconfigurable operation is executing. This wait state is initiated, maintained, and eventually discontinued by arbiter emulation instructions (see Figure 3). We decided to emulate the wait state of the GPP during reconfigurable operations by unconditional branch instructions from the PowerPC ISA. This approach is very performance efficient, as it does not require interrupts. We employed the *'branch to link register'* (**blr**) and *'branch to link register and link'* (**blrl**) PowerPC instructions to emulate a wait state and to get out of this state, respectively. More details on the Molen-specific utilization of the PowerPC **blr** and **blrl** instructions can be found in [14].

Instruction encoding: To perform the Molen processor operations on the prototype, we extend the PowerPC ISA. The additional instructions considered are **execute** and c-**set**. To encode these instructions efficiently, we have considered an encoding scheme, consistent with the PowerPC instruction format. The instructions **movtx**, and **movfx** are directly mapped to the existing PowerPC instructions **mtdcr** and **mfdcr** respectively. We note that the instructions *c*-**set**, **execute**, **movtx**, and **movfx**, determine the minimal functionally complete ISA extension (for details see [10]).

Hardware requirements: To implement the arbitration of the PowerPC instruction bus, the following design considerations are taken into account:

- The entire input information, related to instruction decoding, arbitration and timing is obtained through the instruction bus.
- The PowerPC instruction bus is 64-bit wide and instructions are fetched in couples (doublewords).
- Speculative prefetches are performed without disturbing the correct timing of a reconfigurable instruction execution.
- The arbiter uses the same clock as the instruction memory.

The arbiter has been described in synthesizeable VHDL and mapped on the Virtex II Pro FPGA.

Arbiter functional testing: To test the operation of the arbiter, we have implemented an assembly program, strictly aligned into memory, which tests all possible sequences and alignments of the ISA-extending instructions on the instruction bus. For more details on the arbiter testing see [14].

3.2 The $\rho\mu$-code unit

The $\rho\mu$-code (reconfigurable microcode) unit is a part of the RP, which controls the operation of the configurable computing units (CCU) in a way similar to the traditional microcoded designs. A general view of the $\rho\mu$-code unit is depicted in Figure 4. The $\rho\mu$-code loading unit, loads microprograms from the external memory. It also initiates the operation of the sequencer, once the desired microprogram is transferred to or has been available in the ρ-control store. More specifically, the $\rho\mu$-code unit operates as follows:

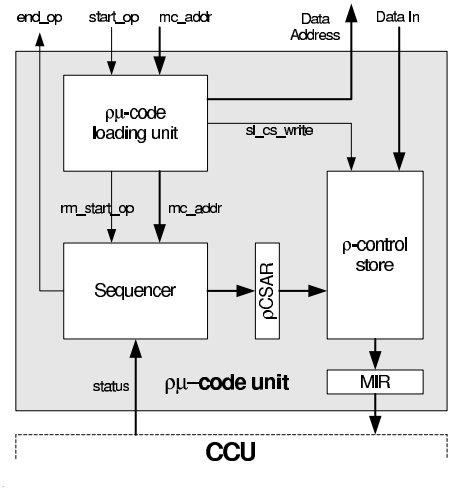

Figure 4: General view of the $\rho\mu$-code unit.

1. When a reconfigurable instruction is decoded, the arbiter generates signal *start_op* initiating a reconfigurable operation.
2. The $\rho\mu$-code loading unit sequentially generates the addresses of the microprogram (through the 'Data Address' bus in Figure 4) to the main memory starting with address *mc_addr*. During the address generation, the desired microprogram is stored into the ρ-control store via the write-only port *Data_In*.
3. Once the desired microprogram is available in the ρ-control store, signal *rm_start_op* activates the sequencer.
4. The sequencer starts to generate $\rho\mu$-code addresses towards the ρCSAR (reconfigurable Control Store Address Register). The microinstruction to be executed by the CCU is loaded into the microinstruction register (MIR).
5. *Status* signals from the CCU are directed to the sequencer to determine the next microcode address. Once the status signals indicate that the CCU has completed the operation, the sequencer generates signal *end_op* to the arbiter.

The *end_op* signal indicates that the reconfigurable operation has completed and the arbiter initiates the execution of the next instruction from the application program.

3.3 XREGs, memory organization, and clocks

The exchange registers (XREGs) provide the interface for the communication between the GPP and the reconfigurable processor (RP). Typical exchanged values are function parameters and results. It is not advised to exchange large pieces of data via the XREGs, because this would decrease the performance of the machine. The XREGs are implemented in a dual-port register file. One port is connected to the GPP, the other - to the RP. We utilize the Device Control Registers (DCR) interface of the embedded PowerPC for the GPP interface of the XREG file. The DCR ports are easily connected to the GPP-side ports of the XREG. Moreover, the DCR transfers are supported by two dedicated PowerPC instructions: **mtdcr** and **mfdcr**. Thus, the Molen instructions **movtx** and **movfx** are mapped to the PowerPC **mtdcr** and **mfdcr**, respectively. The RP port of the XREG file is connected to the CCUs via

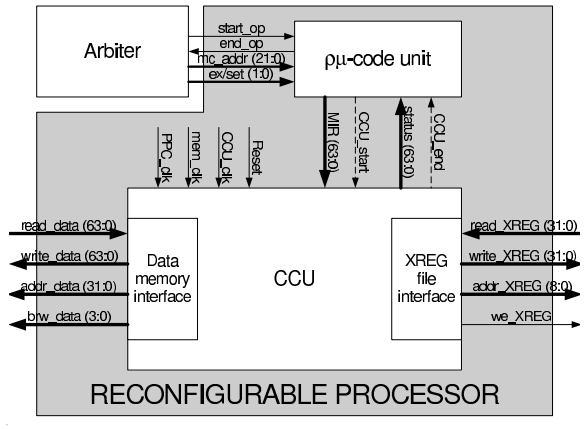

Figure 5: The CCU polymorphic interface.

an interface, described in Subsection 3.4. In the particular prototype, a single BRAM block (2KBytes) organized as 512×32-bit storage is utilized for the XREG file.

Shared memory organization: For transferring large amounts of data, e.g., image data arrays, the XREG mechanism is not efficient. A huge performance penalty has to be paid if every piece of data is moved from the PowerPC allocated memory, via the XREGs, to the CCU allocated memory. To avoid such performance draw-backs, a shared memory is implemented. The arbiter determines the access to the memory by the dedicated signal 'occupy memory'(see Figure 3) based on the performed instruction. In case the instruction is from the standard ISA, the memory control is directed to the PowerPC, if a reconfigurable instruction is executed, the memory control is transferred to the RP, more precisely - to the active CCU.

For the memory design of the prototype, we considered the available BRAM blocks in xc2vp20 and implemented 64 KB for instructions and other 64 KB for application data. Both the instruction and the data memory are organized as 64-bit word memories.

Clock domains: For performance efficiency, three clock domains have been implemented in the prototype:

- **PPC_clk**- PowerPC clock signal set to 250 MHz (vendor recommended frequency);
- **mem_clk**- clock signal to the main memory. This signal has been set to be 3 times lower than the PPC_clk, i.e., 83 MHz;
- **CCU_clk**- clock signal to the CCU driven by an external pin. It may be utilized by any CCU, which requires frequencies different from PPC_clk and mem_clk.

3.4 The CCU interface

An important advantage of the Molen processor is that a new application specific functionality can be embedded without changing the architecture of the processor. At architectural level, the interface between the core processor and the CCUs is determined by the πISA, the exchange registers, and the shared data memory space. Figure 5 depicts the hardware interface signals between a single CCU and the rest of the Molen organization. In accordance with the architectural definition, each CCU can interface with three parts of the Molen organization: the $\rho\mu$-code unit, the XREGs, and the data memory.

Microprogrammable and self-controlled CCUs: The interface with the $\rho\mu$-code unit is based on microprogrammable control. A microinstruction is directed to the CCU via the MIR bus, and status signals are generated to the sequencer of the $\rho\mu$-code unit. The proposed control interface is general and supports not only microprogrammable CCUs, but also CCUs with embedded hardwired control units. For such self-controlled CCUs the two synchronizing signals *CCU_start* and *CCU_end* are explicitly depicted in Figure 5. The *CCU_start* signal initiates the operation of the CCU and the *CCU_end* indicates that the CCU has completed its task. In order a hardwired control CCU to be embedded in the Molen processor, only the requirement to support such a start-stop synchronization should be met. In essence, the *CCU_start* signal is a part of the MIR bus and the *CCU_end* signal- a part of the status bus.

Data memory and XREG file interfaces: For exchanging parameters and results, which do not exceed a few register-wide words, the XREG interface is utilized. The implemented CCU interface to the XREGs includes 32-bit data busses and a 9-bit address bus (512×32-bit XREGs). The implemented shared memory allows the PowerPC and the CCUs to allocate and process data in the same locations. Regarding the data memory interface of the CCUs, we implemented 64-bit data busses and 32-bit addresses. In addition, a byte-enabled writing into the data memory is possible utilizing the *brw_data* bus.

Synchronizing signals: An input *reset* signal and the three clock signals described earlier are available for any CCU design.

3.5 Hardware specifications

In this section, we consider only the reconfigurable hardware overhead due to the Molen "backbone" hardware described above, i.e., the arbiter, the $\rho\mu$-code unit and the associated infrastructure. No CCU implementations are considered as they are application specific and vary per configuration. Data memory has not been considered either, because arbitrary memory volumes can be implemented, without influencing the overall utilization of the other hardware categories.

Design features: The presented Molen prototype design, has the following implementation features:

- Program memory: 64KB in BRAM.
- Data memory: 64KB in BRAM.
- XREGs: 512×32-bit in BRAM.
- Microcode word length: 64 bits.
- Logical data memory segment for microprograms: 4Mx64-bits (22-bit address).
- ρ-control store size: 8KB organized in 64-bit words in BRAM.
- PowerPC clock: 250 MHz.
- Memory clock: 83MHz.
- Available reconfigurable hardware for users: 98% of xc2vp20.

The ρ-control store BRAMs are configured as a monolithic dual port memory. Each of the two ports is unidirectional - one read-only and one write only. The read-only port is used to feed the MIR, while the write-only one loads microcodes from the external memory into the pageable section of the ρ-control store.

Synthesis results: Hardware costs reported by the synthesis tools are presented in Table 1. The first column displays the FPGA resources considered. Column two reports their utilization by the RP, without considering any CCU implementation. This includes the $\rho\mu$-code loading unit, the sequencer and the ρ-control store. Column three presents resource utilization by the arbiter. In column four, the overall resource consumption by the RP infrastructure, the arbiter and the XREGs is presented. Finally, columns five and six respectively present the available FPGA resources in the xc2vp20 chip and the utilized part of these resources by the Molen organization (in %). Synthesis results strongly suggest that the Molen infrastructure consumes trivial hardware, leaving virtually all FPGA resources available for user CCU implementations.

Table 1: Molen organization synthesis results (* RP infrastructure only, CCU implementations excluded).

Device xc2vp20 Speed Grade -5	RP*	Arbiter	Total + XREGs	FPGA Resources	usage %
Slices	71	84	156	10304	1
Flip Flops	78	69	147	20608	1
4 input LUTs	171	150	322	20608	1
BRAMs:	4	N.A.	5	112	4
F_{max} [MHz]	130	143	130	N.A.	N.A.

4. EXPERIMENTAL RESULTS

We experiment with three popular media applications, namely MJPEG, MPEG-2, and MPEG-4.

Evaluation methodology: *A real experimental evaluation approach is employed.* In this approach, the original application code is compiled and run on the core processor as a pure software. The duration of the entire program execution is measured in number of processor clock cycles obtained from the built-in PowerPC timers. In the following step, the benchmark program is annotated to support the reconfigurable implementations of some application specific kernels. Those kernel implementations are loaded into the Molen prototype and the annotated program is then executed on the prototype. The ratio between the cycle numbers for the pure software execution and the program execution in the Molen scenario represents the actual (real) speedup of the application. Based on our real measurements, we also project the overall speedup for applications with compiled code exceeding the amount of program memory implemented on the prototype (MPEG-2 and MPEG-4). We measure the speedup for each interesting kernel using the real experimental approach. Regarding the overall speed-up of the entire application (S_i) with respect to the local speedup of a reconfigurably implemented kernel *i*, the Amdahl's law [15] holds:

$$S_i = \frac{1}{(1 - a_i) + \frac{a_i}{s_i}} \quad (1)$$

Where a_i is the fraction taken by kernel i from the total time of the sequential software execution, and s_i is the (local) speedup of kernel i in the Molen execution scenario. We note that both a_i and s_i are experimentally obtained from the prototype. Furthermore, multiple kernels will speedup an application altogether by:

$$S = \frac{1}{(1 - a) + \sum_i \frac{a_i}{s_i}} \quad (2)$$

Considering (2), we establish the theoretical maximum of the overall achievable application speedup to be:

$$S_{max} = \lim_{\forall i, s_i \to \infty} S = \frac{1}{1 - a} \quad (3)$$

Reconfigurable units considered: Regarding the MJPEG testbench, we utilized an automatically generated reconfigurable design, which supports four computationally demanding operations of the encoding algorithm, namely *block input*, *pre-shift*, *2D-DCT* and *block output* (see Figure 6). They are embedded in a single hardware design generated with the Compaan [16] and the Laura [17] tools. In addition, we manually designed a Molen consistent wrapping interface to embed the automatically generated hardware unit as a CCU. Synthesis results for the DCT* CCU are reported in Table 2. For the MPEG-2 experiments, we considered the Sum-of-Absolute-Differences (SAD) operation, the Discrete-Cosine Transform (DCT), and its inverse transform (IDCT). Three SAD implementations of the design proposed in [18] have been considered:

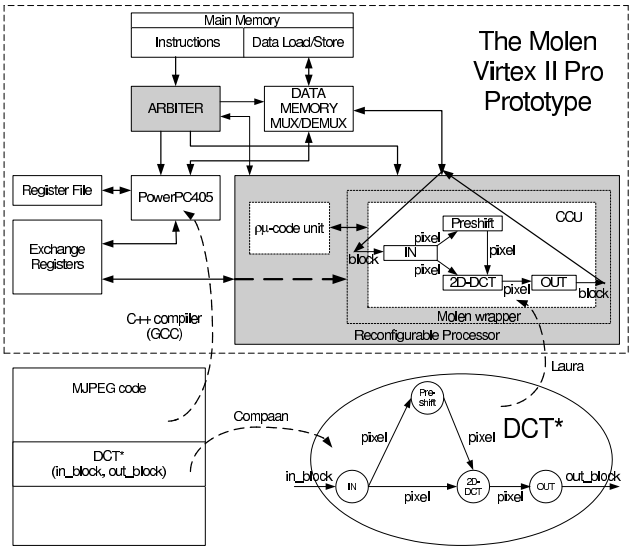

Figure 6: Mapping MJPEG onto the Molen prototype.

Table 2: Synthesis results for the generated DCT* CCU.

Device xc2vp20 Speed Grade -5	CCU	CCU+ wrapper	FPGA Device Resources
Slices	1804	1975	10304
Flip Flops	2271	2388	20608
4 input LUTs	2014	2228	20608
BRAMs	4	4	112
Multipliers18x18	8	8	112
F_{max} [MHz]	100	100	N.A.

SAD16, SAD128, and SAD256, all running at 250 MHz. To support the DCT and IDCT kernels, we synthesized the 2-D DCT and 2D-IDCT v.2.0 cores with the Xilinx Core Generator Tool. We have implemented two MPEG-4 reconfigurable units supporting the repetitive padding and the ACQ function. We clock the DCT, IDCT, Padding, and ACQ CCUs at mem_clk frequency (83MHz).

MJPEG experiments: We considered an MJPEG source code written in C++ and the real experimental approach to evaluate the performance gains. For the experiments, we considered an image size of 48×48 and 4:2:2 YUV macroblock format. We also considered three picture sequences: *tennis*, *barbara*, and *artemis*. In Table 3, column 5 ("impl."), the real experimental speedups are presented. The DCT* software kernel constitutes roughly 61% of the total execution time of the pure software MJPEG encoding algorithm. The theoretical maximum speedups, according to Amdahl's law are reported in column 6. Finally, in column 7, we estimate how close the experimental measurements are to the theoretical maximum speedups (speedup efficiency in %).

MPEG-2 experiments: We target the Berkeley implementation of the MPEG-2 encoder and decoder from libmpeg2. The considered reconfigurable designs of SAD, DCT and IDCT are embedded in the Molen prototype. After obtaining the execution cycle numbers for each kernel both for the execution on the PowerPC alone (pure software) and on the Molen prototype, the real experimental approach is employed for each of these kernels. Table 4 presents the measured real kernel speedups and the projected overall MPEG-2 speedups. Columns labelled "theory" present the theoretically attainable maximum speedup calculated with respect to Amdahl's law. Columns labelled with "impl." contain data for the projected

Table 3: Real MJPEG speedup by the DCT* Molen CCU implementation.

sequence	frame No	Total MJPEG execution [cycles]		Speedup		Speedup efficiency
		Software	DCT* CCU	impl.	theory	%
tennis	1	84556800	40307208	**2.10**	2.57	81.69
	2	84615272	40393200	**2.09**	2.56	81.67
	3	84689544	40462000	**2.09**	2.56	81.69
	4	84629288	40439904	**2.09**	2.56	81.59
	5	84615808	40436592	**2.09**	2.57	81.57
	6	84594184	40409512	**2.09**	2.57	81.58
	7	84471640	40308680	**2.10**	2.57	81.50
	8	84434216	40263576	**2.10**	2.57	81.49
barbara	1	85371112	41131512	**2.08**	2.53	81.94
artemis	1	85577112	41354208	**2.07**	2.52	82.01

Table 4: MPEG-2 kernel speedups and overall speedups

	Kernel speedup			Overall speedup					
	SAD128	DCT	IDCT	MPEG2 encoder			MPEG2 decoder		
				theory	impl.	efficiency%	theory	impl.	efficiency%
carphone	18.9	302.3	24.4	2.85	**2.64**	93	2.02	**1.94**	96
claire	23.9	302.2	24.4	2.99	**2.80**	94	1.60	**1.56**	98
container	35.2	302.1	24.4	3.12	**2.96**	95	1.68	**1.63**	97
tennis	35.0	302.1	32.3	3.37	**3.18**	94	1.68	**1.65**	98

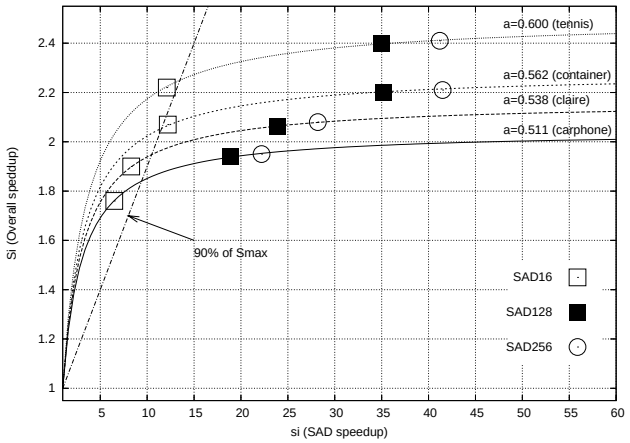

Figure 7: Overall MPEG-2 encoder speedup with three SAD configurations.

speedups with respect to the considered Molen prototype. For the MPEG-2 encoder, the simultaneous configuration of the SAD128, DCT, and IDCT operations has been considered. For the MPEG-2 decoder, only the IDCT reconfigurable implementation has been employed. The results suggest that the real experimental MPEG-2 speedups closely approach the theoretically estimated maximum attainable speedups (see columns "efficiency").

The *projected application speedup* is obtained, employing (1). Figure 7 depicts the overall MPEG-2 encoder speedup depending on the different local speedups of the three SAD kernel implementations. The illustration suggests that the much better kernel performance of SAD128 and SAD256, compared to SAD16, does not contribute as much to the overall application performance (in accordance with Amdahls' law).

MPEG-4 experiments: We consider the publicly available source code of the MoMuSys project, which strictly implements the MPEG-4 video verification model.

The implemented repetitive padding CCU reads the texture and shape data for each boundary block, processes these data, and returns the results into the main Molen memory. We note, that the compiler code optimization effort influences the measured numbers of cycles. Therefore, we carry out experiments both without compiler optimization (option -O0) and with maximum optimization (-O3). Experimental results are presented in table 5. We consider five Binary Alpha Block (BAB) patterns to evaluate the performance gains, denoted by PATT*N*, and calculate an average cycle number for these BAB patterns. Kernel speedups are reported in Table 5, as well.

We experiment with the ACQ CCU on the Molen prototype as we consider two compiler optimization options again. Without optimization (-O0), the function completes for 640 PowerPC cycles; considering -O3, the ACQ is computed in 415 PowerPC cycles. These computational times include all data and parameter transfers and are dominated by the BAB data loads.Table 6 reports the experimental results from running the pure software ACQ kernel and the measured kernel speedups (s_i) on the Molen prototype. We considered a single original BAB pattern and patterns with different distortions from the original. Experiments were carried out for all defined values of the α threshold (see column 1, Table 6). The pattern number in column 2 of Table 6 is equal to the value of the α threshold, for which the ACQ function result switches from 0 to 1 (see column 3). The achieved kernel speedups s_i are reported in the last two columns of Table 6. The bottom line of Table 6 contains the average values for the considered items.

We consider profiling results, reported in the literature [19–23], to establish different MPEG-4 scenarios. From the experimentally measured real kernel speedups we determine an average kernel speedup ($\bar{s}_{av}$), which takes into account the individual contribution of each kernel with respect to its speedup and its part from the entire application execution time. The kernel and projected overall speedups of the same MPEG-4 high profile applications in the considered scenarios are presented in Table 7. Scenarios 5.1, 5.2, and 5.3 (extracted from [23]) correspond to bitstreams L6, L5, and Children, respectively.

Table 5: PowerPC cycles for the MPEG-4 repetitive padding algorithm per block.

	Software -O0		Software -O3		CCU -O0		CCU -O3		si -O0		si -O3	
Block size	8 x 8	16x16	8 x 8	16x16	8 x 8	16x16	8 x 8	16x16	8 x 8	16x16	8 x 8	16x16
PATT0	42581	132117	11637	36277					71	116	32	40
PATT1	70329	259449	17865	65097					117	227	49	71
PATT2	68233	232065	16889	54873	599	1143	368	912	114	203	46	60
PATT3	46088	232161	17445	52929					77	203	47	58
PATT4	70820	256068	17093	60661					118	224	46	67
Average	59610	222372	16186	53967	599	1143	368	912	100	195	44	59

Table 7: Considered MPEG-4 scenarios - kernel (s_i) and projected overall (S) speedups.

Scenario	a	Padding		ACQ		SAD		DCT		IDCT		$\bar{s}_{av}$	S	S_{MAX}
		a_i	s_i	a_i	s_i	a_i	s_i	a_i	s_i	a_i	s_i			theory
MPEG-4 Encoder														
Sc.1 [19]	0.84	0.038	100	0.104	46	0.660	28	0.006	300	0.010	26	31.7	**5.48**	6.41
Sc.2 [20]	0.82	0.039	100	0.104	46	0.660	28	0.005	300	0.008	26	30.7	**4.75**	5.43
Sc.3 [21]	0.97	0.001	100	0.064	46	0.900	28	0.004	300	0.005	26	28.9	**16.74**	38.46
Sc.4 [22]	0.96	0.007	100	0.056	46	0.880	28	0.008	300	0.007	26	29.0	**13.28**	23.64
MPEG-4 Decoder														
Sc.1 [19]	0.24	0,155	100	N.A.	46	N.A.	28	N.A.	300	0.089	26	49.1	**1.31**	1.32
Sc.2 [20]	0.24	0,155	100	N.A.	46	N.A.	28	N.A.	300	0.088	26	49.2	**1.31**	1.32
Sc.3 [21]	0.27	0,042	100	N.A.	46	N.A.	28	N.A.	300	0.226	26	29.4	**1.35**	1.37
Sc.5.1 [23]	0.16	0,160	100	N.A.	46	N.A.	28	N.A.	300	0.001	26	98.3	**1.19**	1.19
Sc.5.2 [23]	0.28	0,270	100	N.A.	46	N.A.	28	N.A.	300	0.010	26	90.8	**1.38**	1.39
Sc.5.3 [23]	0.23	0,140	100	N.A.	46	N.A.	28	N.A.	300	0.090	26	47.3	**1.29**	1.30

Table 6: PowerPC software cycles and kernel speedups for the MPEG-4 ACQ function per 16×16 BAB.

(The Molen CCU cycles measured are 640 with -O0, and 415 with -O3.)

α th	Pattern	ACQ	SW -O0	SW -O3	s_i -O0	s_i -O3
0	PAT00	1	51751	23391	81	56
	PAT16	0	42087	19071	66	46
	PAT32	0	13183	6087	21	15
	PAT64	0	13167	6087	21	15
	PAT128	0	13151	6087	21	15
	ALL0	0	13119	6087	21	15
	ALL255	0	3495	1791	5	4
16	PAT00	1	51751	23383	81	56
	PAT16	1	51743	23383	81	56
	PAT32	0	13183	6075	21	15
	PAT64	0	13167	6075	21	15
	PAT128	0	13151	6075	21	15
	ALL0	0	13119	6075	21	15
	ALL255	0	3495	1779	5	4
32	PAT00	1	51751	23383	81	56
	PAT16	1	51743	23383	81	56
	PAT32	1	51687	23383	81	56
	PAT64	0	13167	6075	21	15
	PAT128	0	13151	6075	21	15
	ALL0	0	13119	6075	21	15
	ALL255	0	3495	1779	5	4
64	PAT00	1	51751	23383	81	56
	PAT16	1	51743	23383	81	56
	PAT32	1	51687	23383	81	56
	PAT64	1	51623	23383	81	56
	PAT128	0	13151	6075	21	15
	ALL0	0	13119	6075	21	15
	ALL255	0	3495	1779	5	4
128	PAT00	1	51751	23383	81	56
	PAT16	1	51743	23383	81	56
	PAT32	1	51687	23383	81	56
	PAT64	1	51623	23383	81	56
	PAT128	1	51559	23383	81	56
	ALL0	0	13119	6075	21	15
	ALL255	0	3495	1779	5	4
Average cycles			29121	13252	46	32

5. CONCLUSIONS

We presented a fully operational prototype design of the Molen reconfigurable processor. Our design effort was motivated by the increasing computational demands of the contemporary media applications and by the failure of current reconfigurable proposals to meet these demands efficiently. We presented a number of experiments on our real Xilinx Virtex II Pro™Molen prototype. These experiments suggested that an arbitrary number of user defined functionalities can be practically and efficiently implemented by a compact PowerPC ISA extension of only four instructions. Moreover, these four additional instructions and the prototype organization allow an arbitrary number of input-output parameters to be transferred between the user hardware and the GPP by means of dedicated exchange registers (XREG) and shared data memory. We demonstrated that it was easy and straight-forward to embed a new application-specific user-defined function in reconfigurable hardware through a clearly defined and comprehensive interface. Regarding performance, our experiments on the proposed prototype indicated remarkable accelerations of the considered media applications, namely MJPEG, MPEG-2, and MPEG-4. In some scenarios, the experimentally measured speedups approach up to 98% of the theoretically attainable maximum speedups. Another important advantage is that the design does not impose performance penalties on the execution of pure software application code on the GPP. Based on the presented experiments, we can safely conclude that the design can be utilized for a larger number of computationally intensive applications, not limited to the media domain only. For example, good candidates for acceleration by the Molen processor can also be many algorithms in cryptography. We believe that the proposed prototype design has potentials for more powerful implementations (e.g., larger memory implementations, complex I/O mechanisms, etc.). Such implementations will clearly enable even larger and more complex applications (compared to the considered in this work) to be implemented performance and cost efficiently within extremely shortened design cycles.

6. REFERENCES

[1] M. Sima, S. Vassiliadis, S.Cotofana, J. van Eijndhoven, and K. Vissers, "Field-Programmable Custom Computing Machines - A Taxonomy," in *12th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 79–88, LNCS 2438, September 2002.

[2] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Comput. Surv.*, vol. 34, no. 2, pp. 171–210, 2002.

[3] S. Hauck, T. Fry, M. Hosler, and J. Kao, "The Chimaera Reconfigurable Functional Unit," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 87–96, 1997.

[4] A. L. Rosa, L. Lavagno, and C. Passerone, "Hardware/ Software Design Space Exploration for a Reconfigurable Processor," in *Proc. of the DATE 2003*, pp. 570–575, 2003.

[5] M. Gokhale and J. Stone, "Napa C: Compiling for a Hybrid RISC/FPGA Architecture," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 126–135, April 1998.

[6] J. Becker and R. Hartenstein, "Configware and morphware going mainstream," *J. Syst. Archit.*, vol. 49, no. 4-6, pp. 127–142, 2003.

[7] F. Campi, M. Toma, A. Lodi, A. Cappelli, R. Canegallo, and R. Guerrieri, "A VLIW Processor with Reconfigurable Instruction Set for Embedded Applications," in *In ISSCC Digest of Technical Papers*, pp. 250–251, Feb 2003.

[8] A. Ye, N. Shenoy, and P. Banerjee, "A C Compiler for a Processor with a Reconfigurable Functional Unit," in *ACM/SIGDA Symposium on FPGAs*, pp. 95–100, 2000.

[9] S. Vassiliadis, S. Wong, and S. Cotofana, "The MOLEN $\rho\mu$-coded processor," in *11th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 275–285, August 2001.

[10] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The Molen Polymorphic Processor," *IEEE Transactions on Computers*, vol. 53, pp. 1363–1375, November 2004.

[11] S. Vassiliadis, S. Wong, and S. Cotofana, "Microcode processing: Positions and directions," *IEEE Micro*, vol. 23, pp. 21–30, July/August 2003.

[12] S. Vassiliadis, G. Gaydadjiev, K. Bertels, and E. Moscu Panainte, "The Molen Programming Paradigm," in *Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation*, pp. 1–7, July 2003.

[13] Xilinx Corporation, *Virtex-II Pro Platform FPGA Handbook v1.0*, 2002.

[14] G. Kuzmanov and S. Vassiliadis, "Arbitrating Instructions in an $\rho\mu$-coded CCM," in *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL 2003)*, pp. 81–90, LNCS 2778, September 2003.

[15] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the AFIPS 1967 Spring Joint Computer Conference*, pp. 483–485, 1967.

[16] A. Turjan, T. Stefanov, B. Kienhuis, and E. Deprettere, "The Compaan Tool Chain: Converting Matlab into Process Networks," in *Designer's Forum of DATE 2002*, pp. 258–264, 2003.

[17] C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere, "Laura:Leiden Architecture Research and Exploration Tool," in *13th International Conference on Field Programmable Logic and Applications (FPL 2003)*, pp. 911–920, LNCS 2778, September 2003.

[18] S. Vassiliadis, E. Hakkennes, J. Wong, and G. Pechaneck, "The Sum Absolute Difference Motion Estimation Accelerator," in *24th EUROMICRO conference (EUROMICRO 98)*, vol. 2, pp. 559–566, August 25-27 1998.

[19] P. Kuhn and W. Stechele, "Complexity analysis of the emerging MPEG-4 standard as a basis for VLSI implementation," in *SPIE Visual Comunications and Image Processing (VCIP)*, vol. 3309, pp. 498–509, Jan. 1998.

[20] J. Kneip, S. Bauer, J. Vollmer, B. Schmale, P. Kuhn, and M. Reissmann, "The MPEG-4 video coding standard - a VLSI point of view," in *IEEE Workshop on Signal Processing Systems,(SIPS98)*, pp. 43–52, 8-10 Oct. 1998.

[21] H.-C. Chang, L.-G. Chen, M.-Y. Hsu, and Y.-C. Chang, "Performance analysis and architecture evaluation of MPEG-4 video codec system," in *IEEE International Symposium on Circuits and Systems*, vol. II, pp. 449–452, 28-31 May 2000.

[22] H.-C. Chang, Y.-C. Wang, M.-Y. Hsu, and L.-G. Chen, "Efficient algorithms and architectures for MPEG-4 object-based video coding," in *IEEE Workshop on Signal Processing Systems*, pp. 13–22, 11-13 Oct 2000.

[23] M. Berekovic, H.-J. Stolberg, M. B. Kulaczewski, P. Pirsh, H. Moler, H. Runge, J. Kneip, and B. Stabernack, "Instruction set extensions for mpeg-4 video," *Journal of VLSI Signal Processing*, vol. 23, pp. 27–49, October 1999.