The MOLEN Processor Prototype

Georgi Kuzmanov Georgi Gaydadjiev Stamatis Vassiliadis

Computer Engineering Lab, EEMCS, TU Delft, The Netherlands,
E-mail:{G.Kuzmanov, G.N.Gaydadjiev, S.Vassiliadis}@EWI.TUDelft.NL
http://ce.et.tudelft.nl/

Abstract

We present a prototype design of the MOLEN polymorphic processor, a CCM based on the co-processor architectural paradigm. The Xilinx Virtex II Pro technology is used as a prototyping platform. Experimental results prove the viability of the MOLEN concept. More precisely, the MPEG-2 application is accelerated very closely to its theoretical limits by implementing SAD, DCT and IDCT in reconfigurable hardware. The MPEG-2 encoder overall speedup is in the range between 2.80 and 2.96. The speedup of the MPEG-2 decoder varies between 1.56 and 1.63.

1 Introduction

The MOLEN CCM concept (proposed in [8]) resolves opcode space explosion, modularity, and compatibility problems (e.g., identified in [2, 3, 5]). Unlike [1, 10], this concept allows implementations with virtually unlimited number of input and output parameters for the reconfigurable functions. In this paper, we present a prototype design of the MOLEN CCM, utilizing the Virtex II Pro FPGA of Xilinx [9]. Our prototype implements a minimal ISA augmentation of only four instructions. Experimental results suggest that the current prototype realization can speedup the MPEG-2 encoder between 2.80 and 2.96 when implementing SAD, DCT and IDCT as reconfigurable functions. When implementing IDCT alone, the projected speedup of the MPEG-2 decoder is between 1.56 and 1.63.

The remainder of the paper is organized as follows. Section 2 describes the MOLEN concept. In Section 3, the microarchitectural and implementation aspects of the prototype are introduced. Considering MPEG-2, Section 4 evaluates the prototype based on experimental results. Finally, concluding remarks are presented in Section 5.

2 The MOLEN polymorphic processor

The two main components in the MOLEN machine organization (depicted in Figure 1) 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 $\rho\mu$-code unit is discussed in more detail in Section 3. 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 1.

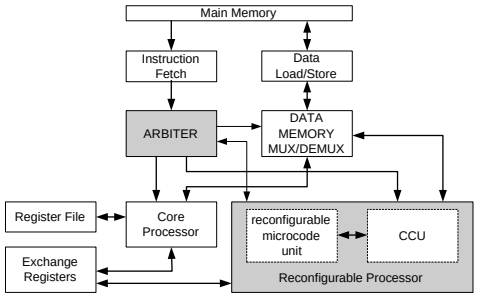

Figure 1. The MOLEN machine organization

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 opera-

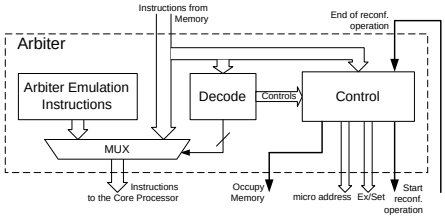

Figure 2. General arbiter organization.

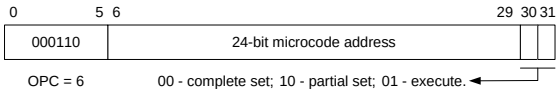

Figure 3. The ρ-form: p-set, c-set, and execute.

tions 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 CCU configured implementation.

The MOLEN programming paradigm is a sequential consistency paradigm targeting the previously described organization (for details see [6]). The complete list of the eight required instructions, denoted as polymorphic instruction set architecture (πISA), is as follows: 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 the parallel execution of 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 execution processes.

3 The prototype design

In our prototype, we consider a **minimal πISA** of four basic instructions: c-**set**, **execute**, **movtx** and **movfx**. This is the smallest set of MOLEN instructions needed to provide a working scenario. As a platform FPGA, we utilized a Xilinx xc2vp20-5 device from the Virtex II Pro family with two embedded PowerPC cores. The **movtx** and **movfx** instructions have been mapped to the existing PowerPC instructions **mtdcr** and **mfdcr**, therefore the arbiter is designed to decode only the **set** and the **execute** instructions.

The arbiter. The operation of the protoype arbiter is based on decoding the incoming instruction flow (see Figure 2). The original GPP ISA instructions are directed (via MUX) to the core processor. Upon decoding of either a **set** or an **execute**, "arbiter emulation instructions" are multiplexed through the core processor instruction bus to drive the GPP into a wait state. In the same time, control signals are issued (via the control block in Figure 2) to the RP (in essence the $\rho\mu$-code unit) to initiate a reconfigurable operation. The microcode location address is redirected to the $\rho\mu$-code unit and the data memory control is transferred to the RP. After an RP operation is completed, data memory control is released back to the GPP, an instruction sequence is generated to ensure that the GPP exits the wait state and the program execution continues with the instruction immediately following the last executed RP instruction. Details regarding arbiter operation and implementation can be found in [4].

Software considerations. Due to performance reasons, we do not use PowerPC special operating modes instructions as arbiter emulation instructions (e.g., exiting power-down modes requires an interrupt). We employed the *'branch to link register'* instruction (**blr**) to emulate a wait state and *'branch to link register and link'* (**blrl**) to move the processor out of this state. Thus the arbiter emulation instructions (Figure 2) are reduced to only one instruction for wait and one for 'wake-up'. More details and additional performance enhancing software considerations are discussed in [4].

πISA Instruction encoding. As a guideline in this implementation, we decided to follow the PowerPC instruction format. We have chosen an opcode from the set of unused opcodes to represent the **set** and **execute** instructions. Figure 3 depicts the reconfigurable instructions format, referred to as ρ-form. A **set** and an **execute** instruction (using the same opcode) are distinguished via instruction modifiers. The encoding allows to utilize a 24-bits address (embedded in the instruction word) to specify the microcode location. Within this address, a modifier bit R/P (resident/pageable) specifies where the microcode is located and how to interpret the address field. That is, either a location in the main memory (R/P=1) or in the $\rho\mu$-code unit (R/P=0).

The $\rho\mu$-code unit. The internal organization of the $\rho\mu$-code unit is depicted in Figure 4. The $\rho\mu$-code unit comprises three main parts: the sequencer, the reconfigurable control (ρ-control) store, and the reconfigurable microcode ($\rho\mu$-code) loading unit. The *start_op* signal is generated by the arbiter and initiates a CCU operation. The $\rho\mu$-code loading unit, loads ($\rho\mu$-code) into the ρ-control store from main memory address *mc_addr*. Once the desired microprogram is available in the ρ-control store, the sequencer starts generating the microcode addresses towards the rC-SAR (reconfigurable Control Store Address Register). The

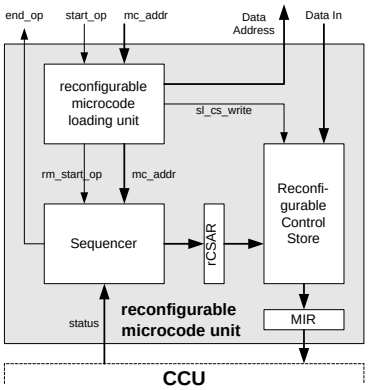

Figure 4. $\rho\mu$-code unit internal organization.

microinstruction to be executed is transferred to the CCU via the Microinsruction Register (MIR). *Status* signals from the CCU are used to determine next microcode address. Once the CCU operation is complete, the sequencer generates signal *end_op* to the arbiter, which initiates the execution of the next instruction from the application program.

The ρ-control store is used to store microcodes and comprises two sections - a *set* and an *execute* section. Both sections can be identical and are further divided into a *fixed* and a *pageable* part. The fixed part stores the resident reconfiguration and execution microcode of the *set* and *execute* phases, respectively. Other microcode is stored in memory and the pageable part of the ρ-control store acts like a cache to provide temporal storage. For more details regarding the considered cache mechanisms and the ρ-control store organization, the interested reader is referred to [8].

Memory organization. We considered the on-chip memory blocks of the utilized FPGA. The available RAM blocks (BRAM) in xc2vp20 allow the implementation of 128 KBytes memory for both data and instructions. For performance improvement, we separated the main memory into two equal segments - 64 KBytes for instructions and 64 KBytes for data. It is possible to extend the memory volume up to the entire space addressable by PowerPC (32-bit addresses) utilizing external memories. The later option has not been considered in our prototype thus far, however.

Clock domains. For performance efficiency, three clock domains have been implemented in our prototype.

1. **PPC_clk**- clock signal to the core processor (PowerPC). The frequency of this signal has been set to 250 MHz, the maximal recommended value for the processors in the utilized xc2vp20-5 engineering silicon; 2. **mem_clk**- clock signal to the main memory. This signal has been bounded to the PPC_clk and has been set to be three times lower, i.e., 83 MHz; 3. **CCU_clk**- a custom clock to the CCU driven by an external pin. It may be utilized by a CCU, which requires frequencies, different from the PPC_clk and the mem_clk.

4 Experimental evaluation

We experimented with the Alpha Data XPL Pro lite development board (ADM-XPL). The MOLEN organization and the considered CCU designs have been described in VHDL and have been synthesized with Project Navigator ISE 5.2 SP3 of Xilinx. The MPEG-2 application has been targeted as a benchmark. We used profiling information to identify and design performance critical kernels as CCU implementations. Due to memory limitations of the current prototype, we run only extracted kernels on the MOLEN processor and directly measure the performance gains. Using these measurements, the profiling data, and Amdahl's law, we estimate the projected overall speedup, rather than directly run the entire MPEG-2 application on MOLEN.

Software profiling results. In the first step of the experimentation we identify the functions that are most suitable for hardware implementations, i.e., the most time-consuming kernels from the application. To this purpose, we performed measurements on a PowerPC 970 processor. The considered application is the Berkeley implementation of the MPEG-2 encoder and decoder from libmpeg2. As input data, we used two popular video sequences, namely *claire* and *container*. Profiling results (obtained with the GNU profiler **gprof**) for the considered benchmarks are presented in Table 1. The execution time spent in SAD, DCT and IDCT operations by MPEG2 encoder (column 6) emphasizes that these functions require around 2/3 of the total execution time. Therefore these functions can be considered for CCU implementations.

Experimental results. We have embedded the considered CCU implementations (SAD, DCT and IDCT) within the prototype and carried out experiments in two stages:

Stage 1. Compile the software kernels for the original PowerPC ISA and run them on one of the PowerPC405 processors, embedded in the xc2vp20 device. The kernels have been extracted from the original application source code without any further code modifications. For our experiments, we considered the same data sequences as used in the profiling phase. After deriving the PowerPC cycle counts for each of the software kernels, we initiate the next stage.

Stage 2. The software kernel code is substituted with a new piece of code to support πISA. The corresponding kernel CCU configuration is present in the reconfigurable processor. Identically to the preceding experimentation stage, we obtain the exact number of PowerPC cycles required to complete the entire kernel operation on MOLEN.

The first two chart groups in Figure 5 present cycle counts for the original PowerPC ISA (in logarithmic scale). The last chart group presents the cycle numbers, consumed by MOLEN while processing the same data. After obtaining the execution cycle numbers both for PowerPC and MOLEN, the speedup of each kernel is estimated. Table 2 presents the calculated kernel speedups with respect to each

Table 1. MPEG2 profiling results for each of the considered functions and its descendants

		MPEG2 encoder				MPEG2 decoder
sequence	# frames@Resolution	SAD (16 x 16)	DCT (8 x 8)	IDCT (8 x 8)	Total	IDCT (8 x 8)
claire	168@360x288	53.8 %	11.8 %	1.0 %	66.6 %	37.6 %
container	300@352x288	56.2 %	10.7 %	1.0 %	67.9 %	40.4 %

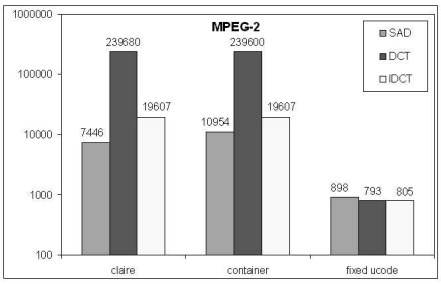

Figure 5. Cycle numbers (*lg* scale) for kernels execution in original PowerPC ISA (claire and container) and in πISA (fixed μcode)

Table 2. MPEG-2 Kernels Speedups

	SAD16	SAD128	SAD256	DCT	IDCT
claire	8.3	23.9	28.2	302.2	24.4
container	12.2	35.2	41.5	302.1	24.4

CCU implementation. Three SAD implementations have been considered, namely SAD16, SAD128 and SAD256, where the number indicates the number of simultaneously processed pixels [7].

Projected application speedup. To calculate the projected speedup of the entire application with respect to the CCU implementations and the πISA, we employed the well known Amdahl's law. Obtained figures for the entire MPEG-2 encoder and MPEG-2 decoder applications are reported in Table 3,where 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. Columns, indicated by label "theory" contain the theoretically achievable maximum speedup. Columns labelled with "impl." contain data for the projected speedups with respect to the considered MOLEN implementation. Results in Table 3 strongly suggest that the MPEG-2 encoder and decoder speedups obtained during our experimentation very closely approach the theoretically estimated maximum possible speedups.

5 Conclusions

We presented a prototype design of the MOLEN polymorphic processor implemented on a Xilinx Virtex II Pro FPGA. The paper discussed various related implementation issues. The MPEG-2 application was accelerated very closely to its theoretical limits by implementing SAD, DCT and IDCT in reconfigurable technology. The MPEG-2 encoder overall speedup was in the range between 2.80 and 2.96 while the speedup of the MPEG-2 decoder varies between 1.56 and 1.63. These results proved the viability of the MOLEN concept and showed its potentials for accelerating complex real-life applications.

Table 3. MPEG2 Overall Speedup

	MPEG2 encoder*		MPEG2 decoder	
	theory	impl.	theory	impl.
claire	2.99	2.80	1.60	1.56
container	3.12	2.96	1.68	1.63

* fixed μ-code SAD128 + DCT + IDCT

References

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

[2] M. Gokhale and J. Stone. Napa C: Compiling for a Hybrid RISC/FPGA Architecture. In *Proc. IEEE Symp. on FCCM*, pp. 126–135, 1998.

[3] S. Hauck, T. Fry, M. Hosler, and J. Kao. The Chimaera Reconfigurable Functional Unit. In *Proc. IEEE Symp. on FCCM*, pp. 87–96, 1997.

[4] G. Kuzmanov and S. Vassiliadis. Arbitrating Instructions in an $\rho\mu$-coded CCM. In *Proc. 13th Intl. Conf. FPL'03*, Springer-Verlag LNCS, vol. 2778, pp. 81–90, 2003.

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

[6] S. Vassiliadis, G. N. Gaydadjiev, K. Bertels, and E. M. Panainte. The molen programming paradigm. In *Proc. Third Intl. Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS'03)*, pp. 1–7,2003.

[7] S. Vassiliadis, E. Hakkennes, S. Wong, and G. Pechanek. The Sum-of-Absolute-Difference Motion Estimation Accelerator. In *Proc. 24th Euromicro Conf.*, pp. 559–566, 1998.

[8] S. Vassiliadis, S. Wong, and S. Cotofana. The MOLEN $\rho\mu$-Coded Processor. In *11th Intl. Conf. FPL'01*, Springer-Verlag LNCS, vol. 2147, pp. 275–285, 2001.

[9] Xilinx Corporation. *Virtex-II Pro Platform FPGA Handbook*, v.1.0, 2002.

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