The MOLEN Polymorphic Processor

S. Vassiliadis, *Fellow, IEEE,* S. Wong, G. Gaydadjiev, *Member, IEEE,* K. Bertels, *Member, IEEE,* G. Kuzmanov, *Student Member, IEEE,*, and E. Moscu Panainte

***Abstract*— In this paper, we present a polymorphic processor paradigm incorporating both general purpose and custom computing processing. The proposal incorporates an arbitrary number of programmable units, exposes the hardware to the programmers/designers and it allows them to modify and extend the processor functionality at will. To achieve the previously stated attributes, we present a new programming paradigm, a new instruction set architecture, a microcode-based microarchitecture, and a compiler methodology. The programming paradigm, in contrast with the conventional programming paradigms, allows general-purpose conventional code and hardware descriptions to coexist in a program. In our proposal, for a given instruction set architecture a one-time instruction set extension of 8 instructions is sufficient to implement the reconfigurable functionality of the processor. We propose a microarchitecture based on reconfigurable hardware emulation to allow high-speed reconfiguration and execution. To prove the viability of the proposal we experimented with the MPEG-2 encoder and decoder and a Xilinx Virtex II Pro FPGA. We have implemented three operations, SAD, DCT, and IDCT. The overall attainable application speedup for the MPEG-2 encoder and decoder is between 2.64 - 3.18 and between 1.56 - 1.94, respectively, representing between 93% and 98% of the theoretically obtainable speedups.**

***Index Terms*— Custom computing machines, FPGA, firmware, reconfigurable microcode, polymorphic processors, reconfigurable processors.**

I. Introduction

General-purpose processors allow us to run the same program over a range of implementations of the same architectural family [1] in a compatible manner. Furthermore, they allow various programs to run on the same system and the same program to run over multiple processing families. One of the major continuous concerns of general-purpose processors is performance. Reconfigurable hardware coexisting with a core processor has been considered as a good candidate to address such a concern. Even though such an approach is promising and several processor paradigms have been proposed, see numerous examples in [2], [3], the organization of such a hybrid processor can be viewed mostly as an open topic. In this paper, we propose a polymorphic processor that improves substantially various aspects, including performance, of such hybrid general purpose processor paradigm. The main contributions of the proposed approach can be summarized by the following:

- For a given ISA, a one time architectural extension (based on the co-processor architectural paradigm) comprising 8 instructions suffices to provide an almost arbitrary number of reconfiguration "functions" per single programming space. This realization resolves the opcode space explosion and modularity problems and provides ISA compatibility and portability of reconfigurable programs, present in previous proposals, such as the ones described in [4]–[6].
- We propose a new processor organization and we describe a programming paradigm based on sequential consistency that allows the proposed co-processor environment to coexist with the general-purpose processor and to resolve parameter limitations and parallel execution problems, present in other proposals (see for example [7], [8]).
- We propose a back-end compiler technology that allows to target the proposed processor architecture, a microarchitecture based on reconfigurable emulation ($\rho\mu$-code), and an implementation that allows the compiled code to execute.

The paper is organized as follows. Section II discusses related work and describes the general approach of how to modify an existing program to support reconfigurable computing. Section III introduces the Molen organization, the Molen programming paradigm, and the polymorphic instruction set architecture (πISA). Section IV discusses the sequencing and compiler extensions required to implement the Molen programming paradigm. Section V describes in detail the underlying microarchitecture and the $\rho\mu$-code unit. Section VI presents an evaluation of the proposed Molen architecture. Section VII presents the overall conclusions.

II. Related work and general approach

As indicated earlier, reconfigurable hardware coexisting with a core general-purpose processor has been considered by several researchers as a good candidate for speeding up applications. For the description of most of the existing proposals, the interested reader is referred to two review/classification articles [2], [3]. 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:

S. Vassiliadis is with the Computer Engineering Laboratory, Delft University of Technology, Mekelweg 4, 2628CD Delft, The Netherlands. Email: S.Vassiliadis@ewi.tudelft.nl.

S. Wong is with the Computer Engineering Laboratory, Delft University of Technology, Mekelweg 4, 2628CD Delft, The Netherlands. Email: J.S.S.M.Wong@ewi.tudelft.nl.

G. Gaydadjiev is with the Computer Engineering Laboratory, Delft University of Technology, Mekelweg 4, 2628CD Delft, The Netherlands. Email: G.N.Gaydadjiev@ewi.tudelft.nl.

K. Bertels is with the Computer Engineering Laboratory, Delft University of Technology, Mekelweg 4, 2628CD Delft, The Netherlands. Email: K.L.M.Bertels@ewi.tudelft.nl.

G. Kuzmanov is with the Computer Engineering Laboratory, Delft University of Technology, Mekelweg 4, 2628CD Delft, The Netherlands. Email: G.Kuzmanov@ewi.tudelft.nl.

E. Moscu Panainte is with the Computer Engineering Laboratory, Delft University of Technology, Mekelweg 4, 2628CD Delft, The Netherlands. Email: elena@ce.et.tudelft.nl.

Manuscript received ..

- **Opcode space explosion**: For reconfigurable fabric, a common approach (e.g., [4], [5], [6]) 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. More specifically stated, for a specific application domain intended to be implemented on the FPGA, the designer and compiler are restricted by the unused 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 [9]. This implies that a reconfigurable implementation developed by a vendor A can not be included without substantial effort by the compiler developed for an FPGA implementation provided by a vendor B.

Additional shortcomings of current proposals regarding performance gains include the following:

- **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]). For example, in the architecture presented in [7], due to the encoding limits, the fragments mapped into the FPGA have at most 4 inputs and 2 outputs; also, in [8], the maximum number of input registers is 9 and it has one output register.
- No support for **parallel execution** on the FPGA of sequential operations: An important and powerful feature of FPGAs can be the parallel execution of sequential operations when they have no data dependency. Many architectures (see for examples in [2]) do not take into account this issue and their mechanism for FPGA integration cannot be extended to support parallelism.

In the discussion to follow, we present the general concept of transforming an existing program to one that can be executed on the reconfigurable computing platform we propose and hints to the new mechanisms, intended to improve existing approaches.

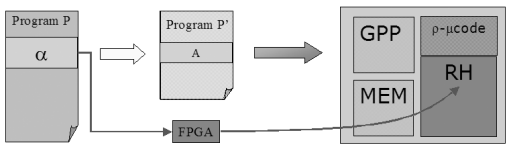


Fig. 1. Program transformation example.

The conceptual view of how program P (intended to execute only on the general-purpose processor (GPP) core) is transformed into program P' (executing on both the GPP core and the reconfigurable hardware) is depicted in Figure 1. The purpose is to obtain a functionally equivalent program P' from program P which (using specialized instructions) can initiate both the configuration and execution processes on the reconfigurable hardware. The steps involved in this transformation are the following:

1) identify code "α" in program P to be mapped in reconfigurable hardware.
2) show that "α" can be implemented in hardware in an existing technology, e.g., FPGA, and map "α" onto reconfigurable hardware (RH).
3) eliminate the identified code "α" and add "equivalent" code (A) assuming that A "calls" the hardware with functionality "α". The code A comprises the following:
   - Repair code inserted to communicate parameters and results to/from the reconfigurable hardware from/to the general-purpose processor core.
   - "HDL"-like hardware code and emulation code inserted to configure the reconfigurable hardware and to perform the functionality that is initialized by the "execute code".
4) compile and execute program P' with original code plus code having functionality A (equivalent to functionality "α") on the GPP/reconfigurable processor.

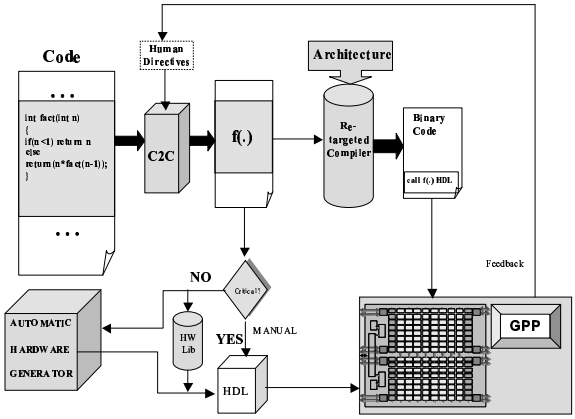


Fig. 2. Program transformation methodology for reconfigurable computing.

The mentioned steps illustrate the need for a new programming paradigm in which both software and hardware descriptions are present in the same program. It should also be noted that the only constraint on "α" is implementability, which possibly implies complex hardware. Consequently, the microarchitecture may have to support emulation [11] via microcode. We have termed this reconfigurable microcode ($\rho\mu$-code) as 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. The methodology of the transformation described previously for the reconfigurable computing platform is depicted in Figure 2. First, the code to be executed on the reconfigurable hardware must be determined. This is achieved by high-level to high-level instrumentation and benchmarking. This results in several candidate pieces of code. Second, we must determine which piece of code is suitable for implementation on the reconfigurable hardware. The suitability is solely determined by whether the piece of

code is implementable (i.e., "fits in hardware"). Those parts can then be mapped into hardware via a hardware description language (HDL). In case the HDL corresponds to "critical" hardware in terms of, for instance, area, performance, memory and power consumption, the translation will be done manually (see Figure 2). Otherwise, the translation can be done automatically, as for example described in [10], [12], [13], or be extracted from a library.

III. Organization, ISA, and Programming

The two main components in the Molen machine organization [14] (depicted in Figure 3) are the 'Core Processor', which is a general-purpose processor (GPP), and the 'Reconfigurable Processor' (RP). Instructions are issued to either processors by the 'Arbiter' and data are fetched(stored) by the 'Data Fetch' units. The 'Memory MUX' unit is responsible for distributing(collecting) data.

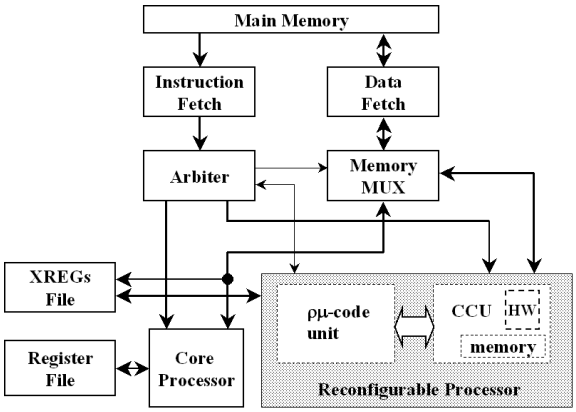


Fig. 3. The Molen machine organization.

The reconfigurable processor is further subdivided into the $\rho\mu$-code unit (discussed in Section V) and the *custom configured unit* (CCU). The CCU consists of reconfigurable hardware, e.g., a field-programmable gate array (FPGA), and memory. All code runs on the GPP except pieces of (application) code implemented on the CCU in order to speed up program execution. Exchange of data between the GPP and the RP are performed via the exchange registers (XREGs) (described in Section IV) depicted in Figure 3. The envisioned support of operations[1] by the reconfigurable processor can be initially divided into two distinct phases: set and execute. In the set phase, the CCU is configured to perform the supported operations. Subsequently, in the execute phase the actual execution of the operations is performed. This decoupling allows the set phase to be scheduled well ahead of the execute phase and thereby hiding the reconfiguration latency. As no actual execution is performed in the set phase, it can be even scheduled upwards across the code boundary in the code preceding the RP targeted code.

In order to target the $\rho\mu$-code processor, we propose a sequential consistency programming paradigm [15]. The paradigm allows for parallel and concurrent hardware execution and it is intended (currently) for single program execution. It requires only a one-time architectural extension of few instructions to provide a large user reconfigurable operation space. The complete list of the eight required instructions, denoted as polymorphic ($\boldsymbol{\pi}o\lambda\nu\mu o\rho\phi\iota\kappa\acute{o}$) **I**nstruction **S**et **A**rchitecture (πISA), is as follows:

- Six instructions are required for controlling the reconfigurable hardware, namely:
  - Two **set** instructions: these instructions initiate the configurations of the CCU. Two instructions are added for partial reconfiguration:
    - the partial set (p-**set** $<address>$) instruction performs those configurations that cover common parts of multiple functions and/or frequently used functions. In this manner, a considerable number of reconfigurable blocks in the CCU can be preconfigured.
    - the complete set (c-**set** $<address>$) instruction performs the configurations of the remaining blocks of the CCU (not covered by the p-**set**) to *complete* the CCU functionality.

    We must note that in case no partial reconfigurable hardware is present, the c-**set** instruction alone can be utilized to perform all the necessary configurations.
  - **execute** $<address>$: this instruction controls the execution of the operations implemented on the CCU. These implementations are configured onto the CCU by the **set** instructions.
  - **set prefetch** $<address>$: this instruction prefetches the needed microcode responsible for CCU reconfigurations into a local on-chip storage facility (the $\rho\mu$-code unit) in order to possibly diminish microcode loading times.
  - **execute prefetch** $<address>$: the same reasoning as for the **set prefetch** instruction holds, but now relating to microcode responsible for CCU executions.
  - **break**: this instruction is utilized to facilitate the parallel execution of both the reconfigurable processor and the core processor. More precisely, it is utilized as a synchronization mechanism to complete the parallel execution.
- Two $move$ instructions for passing values between the register file and exchange registers (XREGs):
  - **movtx** $\text{XREG}_a \leftarrow \text{R}_b$: (move to XREG) used to move the content of general-purpose register R_b to XREG_a.
  - **movfx** $\text{R}_a \leftarrow \text{XREG}_b$: (move from XREG) used to move the content of exchange register XREG_b to general-purpose register R_a.

The $<address>$ field in instructions introduced above denotes the location of the reconfigurable microcode responsible for the configuration and execution processes (see Section V). It must be noted that a single address space is provided with at least $2^{(n-op)}$ addressable functions for reconfiguration, where n represents the instruction length and op the opcode length. If $2^{(n-op)}$ is found to be insufficient, indirect

[1] An operation can be as simple as a single instruction or as complex as a piece of code.

pointing or GPP-like status word mechanisms can extend the addressing of the reconfigurable function space at will. Code fragments constituting of contiguous statements (as they are represented in high-level programming languages) can be isolated as generally implementable functions (that is code with multiple identifiable input/output values). The parameters are passed via the exchange registers (XREGs). In order to maintain correct program semantics, the code is annotated and a hardware description file provides the compiler with implementation specific information such as the addresses where the reconfigurable microcode are to be stored, the number of exchange registers, etc. It should be noted that it is not imperative to include all instructions when implementing the Molen organization. The programmer/implementor can opt for different ISA extensions depending on the performance that needs to be achieved and the available technology. There are basically three distinctive πISA possibilities with respect to the Molen instructions introduced earlier - the *minimal*, the *preferred* and the *complete* πISA extension. In more detail, they are the following:

- **The minimal πISA:** This is essentially the smallest set of Molen instructions needed to provide a working scenario. The four basic instructions needed are **set** (more specifically: c-**set**), **execute**, **movtx** and **movfx**. By implementing the first two instructions (**set**/**execute**) any suitable CCU implementation can be loaded and executed in the reconfigurable processor. Furthermore, reconfiguration latencies can be hidden by scheduling the **set** instruction considerably earlier than the **execute** instruction. The **movtx** and **movfx** instructions are needed to provide the input/output interface between the RP targeted code and the remainder application code.
- **The preferred πISA:** In order to address reconfiguration latencies both p-**set** and c-**set** instructions are utilized. In this case, as the reconfiguration latencies are substantially (or completely) hidden, the loading time of microcode will play an increasingly important role. In these cases, the two **prefetch** instructions (**set prefetch** and **execute prefetch**) provide a way to diminish the microcode loading times by scheduling them well ahead of the moment that the microcode is needed. Parallel execution, for both minimal and preferred πISA, is initiated by a **set**/**execute** instruction and ended by an general-purpose instruction as described in Figure 4(a).
- **The complete πISA:** This scenario involves all πISA instructions including the **break** instruction. In some applications, it might be beneficial performance-wise to execute instructions on the core processor and the reconfigurable processor in parallel. In order to facilitate this parallel execution, the preferred ISA is further extended with the **break** instruction. The **break** instruction provides a mechanism to synchronize the parallel execution of instructions by halting the execution of instructions following the **break** instruction. The sequence of instructions performed in parallel is initiated by an **execute** instruction. The end of the parallel execution is marked by the **break** instruction. It indicates where the parallel execution stops (see Figure 4 (b)). The **set** instructions are executed in parallel according to the same rules.

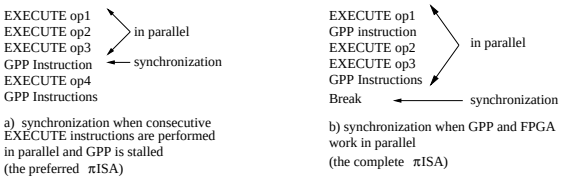


Fig. 4. Parallel execution and models of synchronization.

IV. Compiler and Program Sequence Control

We begin by discussing the exchange registers (XREGs) and the parameter and result passing mechanism between the general-purpose processor and the reconfigurable processor.

The Exchange Registers: The XREGs are used for passing operation parameters to the reconfigurable hardware and returning the computed values after operation execution. Parameters are moved from the register file to the XREGs (**movtx**) and the results stored back from the XREGs in the register file (**movfx**) and the reconfigurable microcode is responsible for managing the parameters from the XREGs and returning the result(s). The following conventions are introduced for single and parallel execution. All parameters of an operation are allocated by the compiler in consecutive XREGs forming a block of XREGs. The microcode of each **execute** instruction has a fixed XREG, which has been assigned during the microcode development. The compiler places in this XREG a link to the block of XREGs where all parameters are stored. This link is the number of the first XREG in the block. Based on these conventions, the parameters for all operations can be efficiently allocated by the compiler and the microcode of each **execute** instruction is able to locate its associated block of parameters. An example is presented in Figure 5, where two operations, namely *op1* and *op2*, are executed in parallel. Their fixed XREGs (XREG0 and XREG1) are communicated to the compiler in a hardware description file. As indicated by the number stored in XREG0, the compiler allocates for operation *op1* two consecutive XREGs for passing parameters and returning results, namely XREG2 and XREG3. The operation *op2* requires only one XREG for parameters and results passing, which in the example is XREG4, as indicated by the content of XREG1.

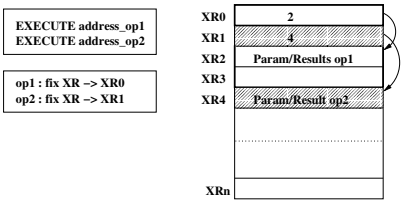


Fig. 5. Exchange Registers allocation by the compiler.

The Compiler: Currently, the compiler [16] relies on the Stanford SUIF2 [17] (Stanford University Intermediate Format) Compiler Infrastructure for the front-end and for the

back-end on the Harvard Machine SUIF [18] framework. The following essential features for a compiler targeting a custom computing machines (CCM) have currently been implemented:

- Code identification: for the identification of the code mapped on the reconfigurable hardware, we added a special pass in the SUIF front-end. This identification is based on code annotation with special pragma directives (similar to [6]). In this pass, all the calls of the recognized functions are marked for further modification.
- Instruction set extension: the instruction set has been extended with **set**/**execute** instructions at both the medium intermediate representation level and low intermediate representation (LIR) level.
- Register file extension: the register file set has been extended with the exchange registers. The register allocation algorithm allocates the XREGs in a distinct pass applied before the register allocation; it is introduced in Machine SUIF, at LIR level. The conventions introduced for the XREGs are implemented in this pass.
- Code generation: code generation for the reconfigurable hardware (as previously presented) is performed when translating SUIF to Machine SUIF intermediate representation, and affects the function calls marked in the front-end.

An example of the code generated by the extended compiler for the Molen programming paradigm is presented in Figure 6. On the left, the C code is depicted. The function implemented in reconfigurable hardware is annotated with a pragma directive named *call_fpga*. It has incorporated the operation name, *op1* as specified in the hardware description file. In the middle, the code generated by the original compiler for the C code is depicted. The pragma annotation is ignored and a normal function call is included. On the right, the code generated by the compiler extended for the Molen programming paradigm is depicted; the function call is replaced with the appropriate instructions for sending parameters to the reconfigurable hardware in XREGs, hardware reconfiguration, preparing the fixed XREG for the microcode of the **execute** instruction, execution of the operation and the transfer of the result back to the general-purpose register file. The presented code is at medium intermediate representation level in which the register allocation pass has not been applied yet.

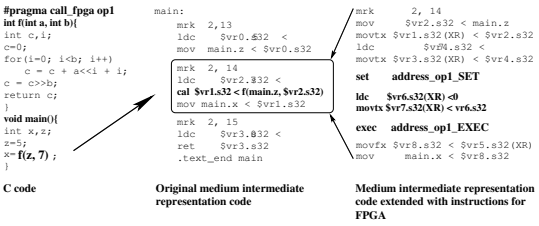


Fig. 6. Medium intermediate representation code.

The compiler extracts from a hardware description file the information about the target architecture such as the microcode address of the **set** and **execute** instructions for each operation implemented in the reconfigurable hardware, the number of XREGs, the fixed XREG associated with each operation, etc.

Parameter exchange, parallelism and modularity: As shown earlier, the exchange registers solve the limitation on the number of parameters present in other reconfigurable computing approaches. If the parameters do not exceed the number of XREGs, parameters are passed by value, otherwise - by reference. The Molen architecture also addresses an additional shortcoming of other reconfigurable computing approaches concerning parallel execution. In case two or more functions considered for CCU implementation do not have any true dependencies, they can be executed in parallel. An example of how this can be performed is depicted in Figure 7. It should be noted that kernels can, as far as such kernels can, be appropriately transformed to the Molen programming paradigm by: a) rewriting the kernel as a separate function, and b) defining a clear set of parameters as interface and passing them as values (or references) between the modified "old" and the new function code. All of the communication between the two functions should be done as much as possible via input/output parameters since both parts will execute in different contexts. The Molen paradigm facilitates modular system design. For instance, hardware implementations described in an HDL (VHDL, Verilog or System-C) are mappable to any FPGA technology in a straightforward manner. The only requirement is to satisfy the Molen set and execute interface. In addition, a wide set of functionally similar CCU designs (from different providers), e.g., sum of absolute differences (SAD) or IDCT, can be collected in a database allowing easy design space explorations.

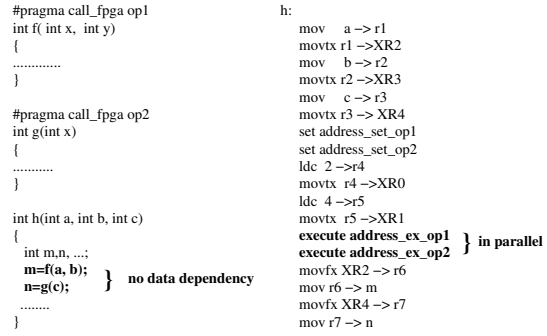


Fig. 7. Parallel execution in Molen.

Interrupts and miscellaneous considerations: Our approach is based on the GPP co-processor paradigm (see for example [19], [20]). Consequently, all known co-processor interrupt techniques [21] are applicable. In order to support the core processor interrupts properly, the following parts are essential for any Molen implementation:

1) Hardware to detect interrupts and terminate the execution before the state of the machine is changed are assumed to be implemented in both core processor (as usual) and reconfigurable processor.
2) Interrupt policies, e.g., priorities, are usually handled by the core processor. Consequently, hardware to communicate interrupts to the core processor is implemented in the reconfigurable processor.
3) Initialization (via the core processor) of the appropriate routines for interrupt handling.

The compiler assumption is that the programmer/implementor of a reconfigurable hardware follows a co-processor paradigm and that (as in the GPP paradigm) the reconfigurable co-processor facility can be viewed as an extension of the core processor architecture, the way co-processors, such as floating point, vector facilities, etc., have been viewed in conventional architectures.

V. A Microarchitecture and its Implementation

In this section, we discuss issues encountered in implementing a microarchitecture supporting the minimal Molen πISA on the Virtex II Pro with the embedded PowerPC 405 serving as the core processor. Experienced microcode designers will recognize that for performance reasons, there is the necessity of having microcode that resides permanently in the control store and microcode that is pageable. We borrow a 'bit' from the instruction to implement resident/pageable microcode. In the instruction format (see Figure 8), the location of the microcode is indicated by the resident/pageable-bit (R/P-bit) which implicitly determines the interpretation of the address field, i.e., as a memory address α (R/P=1) or as a ρ-control store address ρCS-α (R/P=0) indicating a location within the $\rho\mu$-code unit. This location contains the first instruction of the microcode which must always be terminated, e.g., by an *end_op* microinstruction.

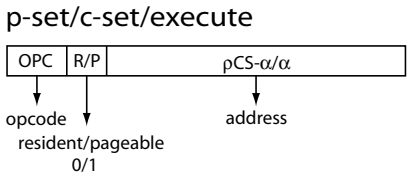


Fig. 8. The *p*-**set**, *c*-**set**, and **execute** instruction format.

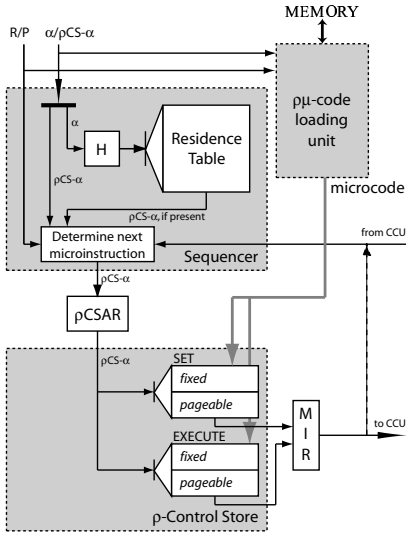


Fig. 9. $\rho\mu$-code unit internal organization.

The $\rho\mu$-code unit: The reconfigurable microcode ($\rho\mu$-code) unit can be implemented in configurable or fixed hardware. In this section, for simplicity, we assume that the $\rho\mu$-code unit is hardwired. The internal organization of the $\rho\mu$-code unit is depicted in Figure 9. The $\rho\mu$-code unit comprises three main parts: the sequencer, the ρ-control store, and the $\rho\mu$-code loading unit. The sequencer mainly determines the microcode execution sequence. The ρ-control store is used as a storage facility for microcode. The $\rho\mu$-code loading unit, as its name suggests, is responsible for the loading of reconfigurable microcode from the memory. The execution of microcode starts with the sequencer receiving an address from the arbiter (see Figure 3) and interpreting it according to the R/P-bit. When receiving a memory address, it must be determined whether the microcode is already cached in the ρ-control store or not. This is done by checking the residence table (see Figure 10) which stores the most frequently used translations of memory addresses into ρ-control store addresses and keeps track of the validity of these translations. It can also store other information: least recently used (LRU) and possibly additional information required for virtual addressing[2] support. In the case that a memory address is received and the associated microcode is not present in the ρ-control store, the $\rho\mu$-code unit initiates the loading of microcode from the memory into the ρ-control store. In the case a ρCS-α is received or a valid translation into a ρCS-α is found, the ρCS-α is transferred to the 'determine next microinstruction'-block. This block determines which (next) microinstruction needs to be executed:

- When receiving the address of the first microinstruction: Depending on the R/P-bit, the correct ρCS-α is selected, i.e., from the instruction field or from the residence table.
- When already executing microcode: Depending on previous microinstruction(s) and/or results from the CCU, the next microinstruction address is determined.

The resulting ρCS-α is stored in the ρ-control store address register (ρCSAR) before entering the ρ-control store. Using the ρCS-α, a microinstruction is fetched from the ρ-control store

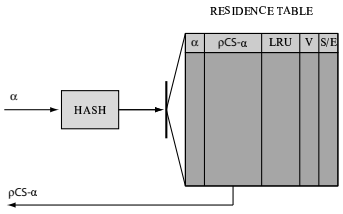


Fig. 10. The sequencer's residence table.

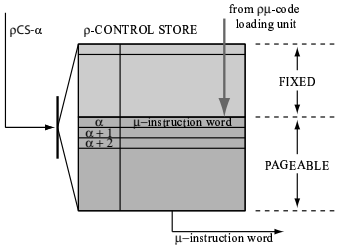


Fig. 11. Internal organization of one section of the ρ-control store.

[2]For simplicity of discussion, we assume that the system only allows real addressing.

and then stored in the microinstruction register (MIR) before it controls the CCU reconfiguration or before it is executed by the CCU. The ρ-control store comprises two sections[3], namely a *set* section and an *execute* section. Both sections are further divided into a *fixed* part and *pageable* part. The fixed part stores the resident reconfiguration and execution microcode of the set and execute phases, respectively. Resident microcode is commonly used by several invocations (including reconfigurations) and it is stored in the fixed part so that the performance of the set and execute phases is possibly enhanced. Which microcode resides in the fixed part of the ρ-control store is determined by performance analysis of various applications and by taking into consideration various software and hardware parameters. Other microcode is stored in memory and the pageable part of the ρ-control store acts like a cache to provide temporal storage. Consequently, cache mechanisms are required to ensure proper ρ-control store operation. The residence table invalidates entries when microcode has been replaced (utilizing the valid (V) bit) or substitutes the least recently used (LRU) entries with new ones. Finally, the residence table can be separate or common (requiring an additional S/E-bit to allow separation) for both the set and execute pageable ρ-control store sections. In the remainder of this section, we present some implementation issues of the minimal Molen πISA utilizing a PowerPC 405 as the core processor, as used in our experimental validation. The minimal πISA consists of the following instructions: **set**, **execute**, **movtx**, and **movfx**. The arbiter (described in detail in [22]) performs a partial decoding of instructions in order to determine where instructions should be issued. The **set** and **execute** instructions will be issued to the reconfigurable processor and in this specific implementation the **movtx** and **movfx** instructions are issued to the core processor. The latter is due to the fact that both *move* instructions are mapped to existing PowerPC instructions, namely **mtdcr** and **mfdcr**, respectively.

General requirements of the arbiter: The arbiter controls the proper co-processing of the core processor and the reconfigurable processor (see Figure 3) by directing instructions to either of these processors. It arbitrates the data memory access of the reconfigurable and core processors and it distributes control signals and the starting microcode address to the $\rho\mu$-code unit.

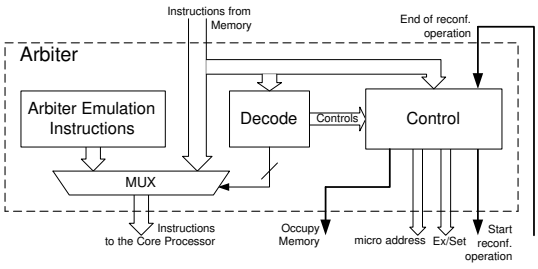


Fig. 12. General arbiter organization.

In Figure 12, a general view of an arbiter organization is depicted. The arbiter operation is based on the decoding of the incoming instructions and either directs instructions to the core processor or generates an instruction sequence to control the state of the core processor. The latter instruction sequence is referred to as "arbiter emulation instructions". Upon decoding of either a **set** or an **execute** instruction, the following actions are initiated:

1) Arbiter emulation instructions are multiplexed to the core processor instruction bus and essentially drive the processor into a wait state.
2) Control signals from the decode block are transmitted to the control block in Figure 12, which performs the following: a) Redirect the microcode location address to the $\rho\mu$-code unit. b) Generate an internal code representing either a **set** or **execute** instruction (Ex/Set) and delivering it to the $\rho\mu$-code unit. c) Initiate the reconfigurable operation by generating *'start reconf. operation'* signal to the $\rho\mu$-code unit. d) Reserve the data memory control for the $\rho\mu$-code unit by generating a *memory occupy* signal to the (data) memory controller. e) Enter a wait state until the signal *'end of reconf. operation'* arrives.

An active *'end of reconf. operation'* signal initiates the following actions: 1) Data memory control is released back to the core processor. 2) An instruction sequence is generated to ensure proper exiting of the core processor from the wait state. 3) After exiting the wait state, the program execution continues with the instruction immediately following the last executed reconfigurable processor instruction.

Software considerations: For performance reasons, PowerPC special operating modes instructions were not used – exiting special operating modes is usually performed by an interrupt. We employed the *'branch to link register'* (**blr**) to emulate a wait state and *'branch to link register and link'* (**blrl**) to move the processor out of this state. The difference between these instructions is that **blrl** modifies the link register (LR), while **blr** does not. The next instruction address is the effective address of the branch target, stored in the link register. When **blrl** is executed, the new value loaded into the link register is the address of the instruction following the branch instruction. Thus, the arbiter emulation instructions, stored into the corresponding block in Figure 12, are reduced to only one instruction for wait and one for 'wake-up' emulation. The PowerPC architecture allows out-of-order execution of memory and I/O transfers, which has to be taken into account in the implementation. To guarantee that data dependency conflicts do not occur during reconfigurable operation, the PowerPC *'synchronization'* instruction (**sync**) can be utilized before a **set** or **execute** instruction. In other out-of-order execution architectures, data dependency conflicts should be resolved by specific dedicated features of the target architectures. In in-order architecture implementations, this problem does not exist.

Instruction encoding: In the previous, we discussed that the **movtx** and **movfx** instructions are mapped to the existing PowerPC instructions **mtdcr** and **mfdcr**. This implemented solution is imposed by the fact that the Virtex II Pro PowerPC

[3]Both sections can be identical, but they are probably only differing in microinstruction word sizes.

core has a dedicated interface to the so-called Device Control Registers (DCR) [23] and two instructions that support DCR transfers (namely **mtdcr** and **mfdcr**). It should be noted that this is a PowerPC specific implementation and not applicable in the general case. This leaves only the **set** and **execute** instructions to be encoded. We follow the PowerPC I-form and choose unused opcodes for both instructions. The manner to distinguish a **set** instruction, an **execute** instruction (using the same opcode) and resident/pageable (R/P) addresses is via instruction modifiers.

Arbiter hardware requirements: To implement the arbiter, we have considered the following: 1) Information, related to instruction decoding, arbitration and timing is obtained only through the instruction bus (from memory). 2) PowerPC instruction bus is 64-bit wide and instructions are fetched in couples. 3) Speculative prefetches should not disturb the correct timing of a reconfigurable processor instruction execution. The arbiter for PowerPC has been described in synthesizable VHDL and mapped on the Virtex II Pro FPGA of Xilinx.

Microcode configuration, termination and finalization: The FPGA reconfiguration files generated after synthesis contain unpredictable bit patterns and will highly depend on the targeted FPGA technology. It is essential to note that the same high-level HDL description results in completely different configuration bitstreams when different technologies are targeted. In case of execution microcode, the end_op microinstruction at the end of the microcode segment is sufficient for the proper termination of the reconfigurable operation provided that the microcode is properly aligned into the memory. This technique, however, would not work for reconfiguration microcode, because the reconfiguration bitstreams are an arbitrary bit sequence. Therefore, it is possible, that the reconfiguration microcode loading is terminated earlier by a false end_op microinstruction. One approach to resolve early termination is the following. An additional microcode word may be aligned at the starting address of the microprogram segment. This word may contain either the length of the microprogram or its end address. Since both methods do not differ in either implementation or microcode size, we have arbitrarily selected the latter one in our current implementation. The process of preparing the microcode for its final alignment into the targeted main memory is called microcode finalization. In microcode termination, additional termination information should be explicitly added to the microprogrammable configuration code. The automated process of microcode finalization for Molen indicating the place of the finalization tool in the Molen CCU design process is depicted in Figure 13. The CCU design, described in HDL, can be targeted to different FPGA technologies. This allows descriptions that can be synthesized to any particular technology utilized by Molen.

The configuration file (indicated as *conf*) contains information about the Molen organization needed for the reconfiguration microcode finalization. The product of the finalization tool is a binary file ready to be used inside the Molen paradigm, and can be a linkable object, or a high-level data structure, incorporating the binary information, that can be included directly in a C project before compilation. It should be noted that the reconfiguration microcode endianness is transparent to the proposed approach and does not require special consideration.

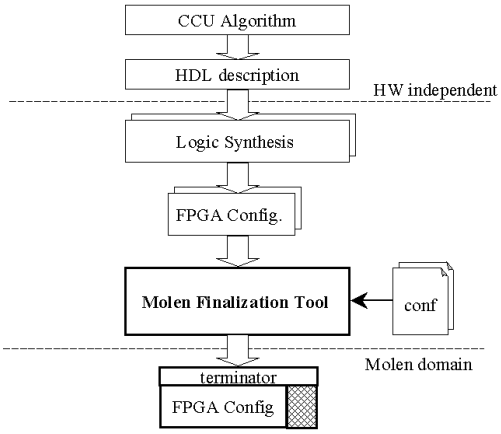


Fig. 13. Molen finalization.

$\rho\mu$-code loading unit implementation: The $\rho\mu$-code loading unit (see Figure 9) is responsible for loading microprograms from the external memory. The *start_op* signal (not depicted in Figure 9) is generated by the arbiter and initiates a reconfigurable operation. The $\rho\mu$-code loading unit sequentially generates the addresses of the microprogram in the main memory and the desired microprogram is loaded into the ρ-control store. Once the microprogram is available in the ρ-control store, i.e., the end address of the microprogram in the external memory is reached, the sequencer starts the execution of the microcode generating microcode addresses towards the $\rho CSAR$. We have to note that other parts of the $\rho\mu$-code unit are not discussed as they are essentially memory-like elements with appropriate controls.

VI. Evaluation

In order to evaluate our proposal, we experimented with the Alpha Data XPL Pro lite development board (ADM-XPL) and the Xilinx Project Navigator ISE 5.1 (Service Pack #3) design environment. As reconfigurable hardware platform, we used the latest Xilinx xc2vp20 devices (speed grade 5) from the Virtex II Pro family. For our experimentation, we target and profile the MPEG-2 application. As implemented in the platform hardware, partial reconfiguration is severely limited because it is allowed only on fixed frame boundaries (the xc2vp20 incorporates 8,214,624 bit configuration memory divided into 1,756 frames) with no possibilities for frame reduction. This limits the flexibility on CCU reconfiguration sizes. For our experiments, we reconfigure the device at system initialization stage. There is an additional platform restriction, namely the available on-chip block RAM (BRAM) memory of xc2vp20 is limited to 128kBytes for both instructions and data. Due to the space limitation we were unable to run any file, I/O, and operating system calls. As a consequence, we used the profiling information to design the kernels as CCU implementations and estimated the performance gains rather than directly run the entire MPEG-2 application on the Molen processor. Furthermore, the following has been

assumed. The parts of the applications which can be implemented on the reconfigurable hardware are isolated in functions. The core processor and the reconfigurable processor do not run concurrently. The operations performed on the reconfigurable processor are sequential (for now, we do not consider potential parallelism due to the lack of compiler support). The applications are compiled without optimizations. The PowerPC processors in VirtexII Pro do not implement floating-point instructions. Therefore, the floating-point data type of the DCT coefficients utilized in the MPEG-2 encoder benchmark has been converted to integer data types. The proper integer arithmetic has been implemented for fairness.

Software Profiling Results: The first step involves identifying the functions that are most suitable for hardware implementation. For this purpose, we performed the measurements on a PowerPC 970 running at 1600 MHz. The considered applications are a set of multimedia benchmarks consisting of the Berkeley implementation of the MPEG-2 encoder and the MPEG-2 decoder included in libmpeg2. The objective is to identify the most time-consuming operations among the following operations, namely SAD (sum of absolute-difference,), 2D-DCT (2-dimensional discrete cosine transform) and 2D-IDCT (2-dimensional inverse DCT). As input data, we used a representative series of video sequences consisting of frames with varying resolutions, presented in Table I, column two.

For our measurements, we used the GNU profiler **gprof** to determine the amount of time spent in each function and its descendants. The results for the considered benchmarks, input data and operations are presented in Table I. For the MPEG2 encoder application, we notice that the SAD function consumes more than 50% of the application time (Table I, column two) and consequently it is the best candidate for hardware implementation. The integer DCT function accounts for around 11% of the application time (Table I, column three). For the IDCT function, we notice that although in the MPEG2 encoder it takes only around 1% of the application time (Table I, column four), in the MPEG2 decoder it requires around 42% of the application time. The total execution time spent in the SAD, DCT and IDCT operations in the MPEG2 encoder (presented in Table I, column five) emphasizes that these functions require around 2/3 of the total application time. Consequently, all considered functions are good candidates for hardware implementations although their contribution to the performance improvement may differ per application.

Molen organization synthesis results: The Molen organization has been described in VHDL and simulated with Modeltech's ModelSim SE 5.7c. The synthesis has been performed with Project Navigator ISE 5.2 SP3 from Xilinx and the Virtex II Pro has been considered as a target reconfigurable technology. For the prototype implementation, we have considered a microcode word length of 64 bits. A 32 MByte memory segment has been considered for storing microprograms into a 64-bit organized main memory. The ρ-control store has been designed to handle up to 8 KBytes of 64-bit microcode words. As primary microcode storage units for the ρ-control store, we have used the BRAM blocks of the FPGA fabric, configured as a dual port memory. Each port is unidirectional - a read-only port is used to feed the MIR, while a write-only one loads microcodes from the external memory into the pageable section of the ρ-control store. The XREGs have been implemented in a single BRAM organized as 512×32-bit storage. Hardware costs reported by the synthesis tools are presented in Table II. The first column presents the FPGA resources considered. Column two reports the actual values of these resources, consumed by the reconfigurable processor, without considering any CCU implementation, i.e., the $\rho\mu$-code unit and the associated infrastructure. This includes the $\rho\mu$-code loading unit, the sequencer and the ρ-control store. Column three presents resource utilization of the arbiter. In column four, the resources consumed by the entire Molen organization are displayed, including the reconfigurable processor infrastructure, the arbiter and the XREGs. 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 %). The results strongly suggest that the Molen infrastructure consumes trivial hardware resources leaving almost the entire area for CCUs.

TABLE II
MOLEN ORGANIZATION SYNTHESIS RESULTS

Device xc2vp20 Speed Grade -5	RP*	Arbiter	Total incl. XREGs	Available Resources	%
# Slices	71	84	156	10304	1
# Flip Flops	78	69	147	20608	1
# 4 inp LUTs	171	150	322	20608	1
# BRAMs:	4	N.A.	5	112	3
f_{max} [MHz]	130	143	130	N.A.	N.A.

* Reconfigurable processor without any CCU implemented

Synthesis results for the CCUs: We implemented the functionalities of the kernels, suggested by the profiling results, into reconfigurable hardware. Synthesis results for the xc2vp20 chip are reported in Table III.

TABLE III
SYNTHESIS RESULTS PER CCU IMPLEMENTATION

Device xc2vp20-5	SAD 16	SAD 128	SAD 256*	DCT	IDCT	Available Resources
# Slices	831	6807	13613*	4314	5436	10304
# Flip Flops	1448	11862	23724*	7964	9876	20608
# 4 inp LUT	1390	11379	22757*	6832	8624	20608
# BRAMs:	N.A.	N.A.	N.A. *	2	2	112
f_{max}[MHz]	310	310	310*	96	96	N.A.

* Results for xc2vp50 FPGA

For the SAD function, we implemented the organization proposed in [24]. The super-pipelined 16-byte version of this SAD organization (SAD16) is capable of processing one 16-pixel line (1 pixel is 1 byte) of a macroblock in 17 cycles at over 300 MHz. The 128-byte version (SAD128) processes eight macroblock lines in 23 cycles, and the 256-byte version (SAD256), processes an entire 16×16-pixel macroblock in 25 cycles at 300 MHz. The latter design (SAD256) requires more resources than available in the xc2vp20 chip used for this experimentation, therefore, we consider it for future implementation when the larger xc2vp50 becomes available. To support the DCT and IDCT kernels, we synthesized the 2-D DCT and 2D-IDCT v.2.0 cores available as IPs in the Xilinx

TABLE I

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)
carphone	96@176x144	51.1 %	12.5 %	1.3 %	64.9 %	50.4 %
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 %
tennis	112@352/240	60.0 %	9.5 %	0.8 %	70.3 %	40.5 %

Core Generator Tool. The parameters for their synthesis are presented in Table IV.

TABLE IV

SYNTHESIS PARAMETERS FOR THE 2-D DCT AND 2-D IDCT IPS

Parameter	2-D DCT	2-D IDCT
Data width [bits]	16 (signed)	16 (signed)
Coeff. width [bits]	24	24
Result width [bits]	16 (rounded)	16 (rounded)
cycles/input sample	6	8
Internal latency [cyc]	94	97

Since the recommended maximum PowerPC frequency for the xc2vp20-5 FPGA is 250 MHz, the ADM-XPL prototyping board vendors recommend to obtain this frequency from a user clock of 83MHz multiplied by 3 using the on-chip FPGA Digital clock managers (DCMs). Considering these recommendations and synthesis results from Table III for our experiments, we have to run the DCT and IDCT functions at a frequency three times lower than the PowerPC clock. The SAD designs were clocked at the same frequency as the PowerPC.

MPEG-2 performance experiments: We have embedded the considered CCU implementations within the Molen organization and executed the corresponding software kernels for performance measurements. For our experiments, we first compiled the software kernels for the original PowerPC ISA and ran them on one of the PowerPC405 processors, embedded in the xc2vp20 device. The kernels have been extracted from the original application source code (the ANSI C code used for the profiling) without any further code modifications. For our experiments, we considered the same data sequences as used in the profiling phase. The PowerPC timers are initialized before a kernel is executed and are read immediately after the kernel execution has completed. Thus, the exact number of PowerPC cycles, required for the entire kernel execution can be obtained. After we derived the cycle counts for the PowerPC ISA software runs, we initiated the next stage of the experimentation. At this stage, similar to the code transformation discussed in Section II, the kernel software code is substituted with a new piece of code to support the πISA. The corresponding kernel CCU configuration is present in the reconfigurable processor considering the discussion in the beginning of this section. Identically to the preceding experimentation stage, we obtain the exact number of PowerPC cycles required to complete the entire kernel operation on Molen. The measurements include cycle numbers for transferring parameters to/from the exchange registers (implemented as DCRs), cycles for memory transfers, and data processing cycles. Figure 14 depicts the measured cycles obtained in the latter two experimentation phases. The first four chart groups present cycle counts for the original PowerPC ISA. The last chart group presents the cycle numbers, consumed by Molen while processing the same data. It should be noted that the performance of the PowerPC software implementations of the three kernels are highly dependent on the data contents. On the contrary, for all four data sequences, the cycle number for the Molen implementation depends only on the amount of data and not on the data contents due to the data independent CCU designs. Therefore, only a single group of results for all data sequences in the Molen execution is presented in Figure 14. In this figure, only fixed microcode implementations are depicted.

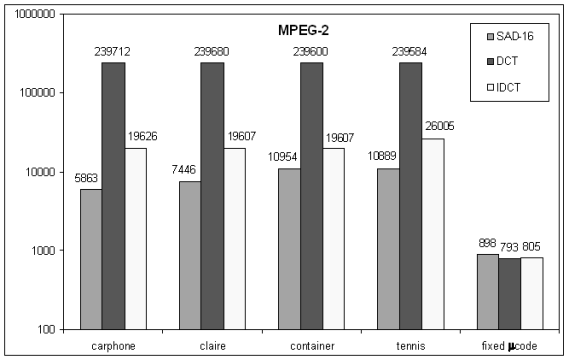


Fig. 14. Cycle numbers for kernels execution in original PowerPC ISA and fixed microcode in πISA.

In addition, we have implemented both fixed and pageable microcode implementations for SAD16 and SAD128. Table V reports measured cycle numbers for executing the SAD kernel over a single macroblock in different Molen configurations. As it has been noted, the SAD256 implementation hardware requirements exceed the capabilities of the xc2vp20 device we used. Therefore, the corresponding SAD256 cycle numbers in Table V have been extrapolated from the results of SAD16 and SAD128.

TABLE V

CYCLES PER MACROBLOCK FOR DIFFERENT SAD IMPLEMENTATIONS

	SAD16	SAD128	SAD256
fixed microcode	898	311	264
peageable microcode	914	331	284

After the cycle numbers for the execution of each kernel have been obtained, both for PowerPC and Molen, the speedup of each kernel can be estimated. Table VI presents the calculated speedups for each of the considered data sequences with respect to each CCU implementation.

Projected application speedup: Results in Table VI suggest that the considered kernels can be speeded up to 300 times

TABLE VI
MPEG-2 SPEEDUP ESTIMATIONS FOR DIFFERENT KERNELS

	SAD16		SAD128		SAD256		DCT	IDCT
	fixed	pag.	fixed	pag.	fixed	pag.	fixed	fixed
carph.	6.5	6.4	18.9	17.7	22.2	20.6	302.3	24.4
claire	8.3	8.1	23.9	22.5	28.2	26.2	302.2	24.4
cont.	12.2	12.0	35.2	33.1	41.5	38.6	302.1	24.4
tennis	12.1	11.9	35.0	32.9	41.2	38.3	302.1	32.3

TABLE VII
OVERALL SPEEDUP ESTIMATIONS FOR THE ENTIRE MPEG2

	MPEG2 encoder*			MPEG2 decoder		
	theory	impl.	impl./th.	theory	impl.	impl./th.
carphone	2.85	2.64	93%	2.02	1.94	96%
claire	2.99	2.80	94%	1.60	1.56	98%
container	3.12	2.96	95%	1.68	1.63	97%
tennis	3.37	3.18	94%	1.68	1.65	98%

* fixed $\rho\mu$-code SAD128 + DCT + IDCT

and one can incorrectly assume, that the entire application can be speeded up to the same orders of magnitude.[4] In the following, we are going to prove theoretically, combined with experiments, that in fact lower, yet considerable and impressive for the GPP domain, overall application speedups could be expected. As indicated earlier, due to space limitations, no file, I/O, or operating system calls have been implemented on the prototype FPGA, thus the application speedup can only be estimated. 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, utilizing the following notations. Let us assume T to be the execution time of the original program (say measured in cycles) and T_{SEi} - time to execute kernel *i* in software, which we would like to speed up in reconfigurable hardware. Assume $T_{\rho i}$ is the execution time (in πISA) for the reconfigurable implementation of kernel *i*. Assuming $a_i = \frac{T_{SEi}}{T}$ and $s_i = \frac{T_{SEi}}{T_{\rho i}}$, the speedup of the program with respect to the reconfigurable implementation of kernel *i* is:

$$S_i = \frac{T}{T - T_{SEi} + T_{\rho i}} = \frac{1}{1 - (a_i - \frac{a_i}{s_i})} \quad (1)$$

Identically, assuming $a = \sum_i a_i$, all the kernels considered for reconfigurable implementation would speed up the program with:

$$S = \frac{1}{1 - (a - \sum_i \frac{a_i}{s_i})},\ S_{max} = \lim_{\forall s_i \to \infty} S = \frac{1}{1 - a} \quad (2)$$

Where S_{max} is the theoretical maximum speedup. Parameters a_i are the profiling results from Table I and parameters s_i are the results from Table VI. The projected overall speedup figures for the entire MPEG-2 encoder and MPEG-2 decoder applications are reported in Table VII. Columns labeled "theory" present the theoretically achievable maximum speedup calculated with respect to Equation (2). Columns labelled with "impl." contain data for the projected speedups with respect to the considered Molen implementation. For the MPEG-2 encoder, the simultaneous configuration of the SAD128, DCT, and IDCT operations employing fixed microcode implementations has been considered. For the MPEG-2 decoder, only the IDCT reconfigurable implementation has been employed. Columns with label "imp./th." in Table VII indicate (in %) how close the real speedup is to the theoretically attainable one. Reported results strongly suggest that the actual speedup of the MPEG-2 encoder and decoder obtained during our practical experimentation very closely approach the theoretically estimated maximum possible speedups.

[4]If the considered kernels are the entire application, speedups of the same orders of magnitude can be expected. If this is not the case, as in the considered MPEG-2, the above assumption is incorrect.

VII. CONCLUSIONS

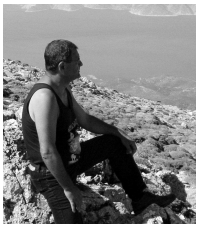
In this paper, we presented a polymorphic processor paradigm that allows the programmer/designer to modify and extend the processor functionality and hardware at will without architectural and design modifications. The proposal solves a number of limitations of existing approaches such as the opcode space explosion and it requires only a one time extension of the instruction set to incorporate an almost unlimited number of reconfiguration functions per single programming space. Finally, it introduces a modular approach allowing easy porting of applications to different reconfigurable platforms and allows compiler controlled parallelism.

Acknowledgements: This research is partially supported by PROGRESS, the embedded systems research program of the Dutch organization for Scientific Research NWO.

REFERENCES

[1] G. Blaauw and F. Brooks Jr., *Computer Architecture*. Addison-Wesley, 1997.
[2] 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)*, vol. 2438. Montpellier, France: Springer-Verlag Lecture Notes in Computer Science (LNCS), Sep 2002, pp. 79–88.
[3] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Comput. Surv.*, vol. 34, no. 2, pp. 171–210, 2002.
[4] S. Hauck, T. Fry, M. Hosler, and J. Kao, "The Chimaera Reconfigurable Functional Unit," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, Napa, California, 1997, pp. 87–96.
[5] A. L. Rosa, L. Lavagno, and C. Passerone, "Hardware/Software Design Space Exploration for a Reconfigurable Processor," in *Proc. of the DATE 2003*, Munich, Germany, 2003, pp. 570–575.
[6] M. Gokhale and J. Stone, "Napa C: Compiling for a Hybrid RISC/FPGA Architecture," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, Napa, California, April 1998, pp. 126–135.
[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*, Feb 2003, pp. 250–251.
[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*, Montery, California, USA, 2000, pp. 95–100.
[9] J. Becker and R. Hartenstein, "Configware and Morphware going Mainstream," *J. Syst. Archit.*, vol. 49, no. 4-6, pp. 127–142, 2003.
[10] 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*, Paris, France, 2003, pp. 258–264.
[11] S. Vassiliadis, S. Wong, and S. Cotofana, "Microcode Processing: Positioning and Directions," *IEEE Micro*, vol. 23, no. 4, pp. 21–30, July/August 2003.
[12] J. M. P. Cardoso and H. C. Neto, "Compilation for FPGA-Based Reconfigurable Hardware," *IEEE Design & Test of Computers*, vol. 20, no. 2, pp. 65–75, March/April 2003.

[13] C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere, "LAURA: Leiden Architecture Research and Exploration Tool," in *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL)*, September 2003, pp. 911–920.
[14] S. Vassiliadis, S. Wong, and S. Cotofana, "The MOLEN $\rho\mu$-Coded Processor," in *11th International Conference on Field Programmable Logic and Applications (FPL)*, vol. 2147. Belfast, UK: Springer-Verlag Lecture Notes in Computer Science (LNCS), Aug 2001, pp. 275–285.
[15] 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*, Samos, Greece, July 2003, pp. 1–7.
[16] E. Moscu Panainte, K. Bertels, and S. Vassiliadis, "Compiling for the Molen Programming Paradigm," in *13th International Conference on Field Programmable Logic and Applications (FPL)*, vol. 2778. Lisbon, Portugal: Springer-Verlag Lecture Notes in Computer Science (LNCS), Sep 2003, pp. 900–910.
[17] http://suif.stanford.edu/suif/suif2.
[18] http://www.eecs.hardvard.edu/hube/research/machsuif.html.
[19] A. Padegs, B. Moore, R. Smith, and W. Buchholz, "The IBM System/370 Vector Architecture: Design Considerations," *IEEE Transactions on Computers*, vol. 37, pp. 509–520, 1988.
[20] W. Buchholz, "The IBM System/370 Vector Architecture," *IBM Systems Journal*, vol. 25, no. 1, pp. 51–62, 1986.
[21] M. Moudgill and S. Vassiliadis, "Precise Interrupts," *IEEE Micro*, vol. 16, no. 1, pp. 58–67, January 1996.
[22] 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'03)*, September 2003, pp. 81–90.
[23] *Virtex-II Pro Platform FPGA Handbook v1.0*, Xilinx Corporation, 2002.
[24] S. Vassiliadis, E. Hakkennes, S. Wong, and G. Pechanek, "The Sum-of-Absolute-Difference Motion Estimation Accelerator," in *Proceedings of the 24th Euromicro Conference*, August 1998, pp. 559–566.



Stamatis Vassiliadis was born in Manolates, Samos, Greece in 1951. He is currently a chair professor in the Electrical Engineering department of Delft University of Technology (TU Delft), The Netherlands. He had also served in the EE faculties of Cornell University, Ithaca, NY and the State University of New York (S.U.N.Y.), Binghamton, NY. He worked for a decade with IBM where he had been involved in a number of advanced research and development projects. For his work he received numerous awards including 24 publication awards, 15 invention awards and an outstanding innovation award for engeneering/scientific hardware design. His 70 USA patents rank him as the top all time IBM inventor. In 1992 he received an honorable mention best paper award at the ACM/IEEE MICRO25. He received the best paper awards in the IEEE CAS (1998,2002), IEEE ICCD (2001) and PDCS (2002). Dr. Vassiliadis is an IEEE fellow.

Stephan Wong was born in Paramaribo, Suriname in 1973. He obtained his PhD in the Electrical Engineering department of the Delft University of Technology (TU Delft), The Netherlands. He is currently working as an assistant professor at the Computer Engineering Laboratory at the Delft University of Technology (TU Delft), The Netherlands. He has considerable experience in the design of embedded media processors. He has worked also on microcoded FPGA complex instruction engines and the modeling of parallel processor communication networks. His research interests include embedded systems, multimedia processors, complex instruction set architectures, reconfigurable and parallel processing, microcoded machines, and network processors.

Georgi Gaydadjiev was born in Plovdiv, Bulgaria, in 1964. He is currently an assistant professor at the Computer Engineering Laboratory, Delft University of Technology, The Netherlands. His research and development experience includes 15 years in hardware and software design at System Engineering Ltd. in Pravetz Bulgaria and Pijnenburg Microelectronics and Software B.V. in Vught, the Netherlands. His research interest include: embedded systems design, advanced computer architectures, hardware/software co-design, VLSI design, cryptographic systems and computer systems testing.

Koen Bertels was born in Antwerp, Belgium in 1961. He is currently on the faculty of Electrical Engineering at Delft University of Technology (TU Delft), The Netherlands. His research involves the development of semi-automatic platforms for the design of embedded systems. The tools are intended for SoC reconfigurable technologies. He is further involved in the simulation and analysis of interacting migrating processes and multi-agent systems from a computer engineering perspective. His research interests are in complex systems, reconfigurable computing, agent technology, back-end compilers, semi-automatic tool platforms, and distributed computing.

Georgi Kuzmanov was born in Sofia, Bulgaria in 1974. He received his M.Sc. degree in computer engineering from the Technical University of Sofia, Bulgaria, in 1998 and is currently working toward his Ph.D. degree with the computer engineering lab of Delft University of Technology (TU Delft), The Netherlands. Between 1998 and 2000, he was with "Info MicroSystems" Ltd., Sofia, where he had been involved in several reconfigurable computing and ASIC projects as a research and development engineer. Georgi Kuzmanov is an IEEE student member. His current research interests include reconfigurable computing, video and image processing, multimedia embedded systems, computer arithmetic, computer architecture and computer organization.

Elena Moscu Panainte was born in Adjud, Romania in 1977. She received her MSc degree (in computer science) from "Politehnica" University of Bucharest, Romania, in 2001. Currently, she is a PhD student in Computer Engineering Laboratory, Electrical Engineering Department, Delft University of Technology, The Netherlands. Her research interests include compiler design, reconfigurable computing and hardware-software co-design.