Polymorphic Processors: How to Expose Arbitrary Hardware Functionality to Programmers

S. Vassiliadis, S. Wong, G. Gaydadjiev, and K. Bertels
Computer Engineering Laboratory,
Delft University of Technology,
The Netherlands.
http://ce.et.tudelft.nl

Abstract – **In this paper, we describe a way to extend the flexibility of hardware and provide the programmer with an arbitrary number of processing units to use. To achieve our goals, 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 paradigm, mixes general-purpose conventional code with hardware descriptions and allows ultra complex instructions. The instruction set is designed such that it requires only a one-time extension for every family of computers. It requires only 8 instructions that are capable of invoking emulation. Emulation is combined with the micro-architecture to allow high-speed reconfiguration and execution. Finally, it is indicated that a compiler can be built to automatically transform a program to conform with the described polymorphic processing paradigm.**

1. INTRODUCTION

In processor design, two extremes exist in terms of flexibility and performance. In one extreme lie the application-specific integrated circuits that, at the expense of flexibility, achieve the highest possible performance. In the other extreme there are the general-purpose processors that are designed to operate in a wide range of opcodes, i.e., providing high flexibility. This however limits their performance since they are designed to perform reasonably well for a wide range of operations. In this paper, we discuss another approach in processor design that is quickly gathering support and acceptance, namely reconfigurable computing. Further, we describe our view on how such a paradigm can be incorporated into the general-purpose computing. In reconfigurable computing as viewed in this paper, a general-purpose processor core is augmented with reconfigurable hardware. The general-purpose processor is intended to perform non-time-critical functions while time-critical functions are implemented on the reconfigurable hardware. An example approach is depicted in the right-most block of Figure 1. Traditionally, computer programs were executed on a general-purpose processor (depicted by 'program P' and the upper long arrow in Figure 1) operating on data stored in memory. This scenario has the following advantages:

- *Op-code space insensitivity:* an arbitrary number of programs can be run without changing the interface, i.e., the instruction-set architecture (ISA), between programmers and designers;
- *Parallel execution support:* performance-related ISA support can be present enabling parallel/concurrent execution of programs;
- *Modularity:* the same program can be executed on an arbitrary number of computers without changes.

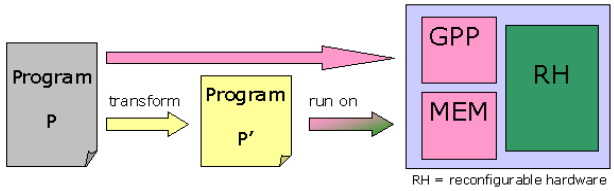

Figure 1. Reconfigurable hardware combined with general-purpose processing.

In the reconfigurable computing scenario, compute-intensive operations are implemented on the reconfigurable hardware providing the following advantages:

- inherent parallelism within the operations can be exploited and therefore the performance of the implemented operations is greatly increased;
- temporary storage of (large amounts of) data can be kept in the same reconfigurable hardware and thereby greatly reducing data access latencies.

The main benefit of utilizing reconfigurable hardware is that other (future) operations can be implemented on the reconfigurable hardware without requiring a re-design. However, each implementation on the reconfigurable hardware must be initialized and controlled by the general-purpose processor core. This can be solved by introducing new instructions to the instruction-set architecture of the general-purpose processor core for each supported and future reconfigurable hardware implementations. However, this approach leads to at least on of the following issues that are resolved in general-purpose paradigms:

- *opcode space sensitive*: in a number of reconfigurable schemes (Gokhale and Stone(1), Hauck et al(2), La Rosa(3)), anytime operations are to be executed in reconfigurable hardware, a new instruction for the operation in question has to be added. This restricts the operation to be executed in reconfigurable hardware and it restricts the program that can benefit from the reconfigurable hardware.
- *no parallel execution support*: in most approaches (Sima et al (4)), there is no support for parallel issuing of reconfigurable operations.
- *no modularity*: given that the reconfigurable operation are specific instruction set dependent, there are currently no generic methodology and tools that will allow to run the same program on multiple hardware reconfigurable platforms. Further, the compiler for a single application has to be extended on the "fly" assuming the hardware is properly set to accommodate the addition of new reconfigurable instructions.

In this paper, we describe the virtual polymorphic processing approach which addresses reconfigurable issues in a unified manner. To provide the programmer with almost arbitrary hardware functionality, we present a programming paradigm, a processor architecture, a microarchitecture, and a compilation methodology. Furthermore, our approach largely maintains the advantages associated with general-purpose processors, namely op-code space insensitivity, parallel execution support, and modularity. This presentation is organized as follows. Section 2 introduces the research questions that facilitate our discussion. Section 3 discusses how existing program can be modified to support reconfigurable computing. Section 4 describes the Molen architecture that requires at most 8 new instructions to support any functionality on the reconfigurable hardware. Furthermore, reconfigurable microcode is introduced that controls both the reconfiguration and execution processes on the reconfigurable hardware. Section 5 discusses the compiling techniques needed to support our new programming paradigm and new processor architecture. In addition, we briefly present some experimental results showing the benefits of our approach. Section 6 concludes this paper with some remarks.

2. RESEARCH QUESTIONS

In reconfigurable computing, the following scenario can be considered to be the general case. First (see Figure 1), the original program P must be transformed into program P' that incorporates support, i.e., containing instructions, to control the reconfigurable hardware. Secondly, program P' is executed on both the general-purpose processor (GPP) core and the reconfigurable hardware. As discussed in the previous section, several issues are associated with a straightforward approach in augmenting reconfigurable hardware to a general-purpose processor core. The result is that huge efforts must be made when adding support for future operations on the reconfigurable hardware. Therefore, a new approach must be investigated and established that overcomes all these issues. In order to facilitate the comprehension of the material presented here, we first clearly state the research questions we address. The first research question is:

1. *How can I change an existing program (without re-developing it) so that I can speed it up on the reconfigurable hardware?*

The answer lies in proposing a new programming paradigm. The programming paradigm must require minimal changes when transforming the original program P to program P' (see Figure 1), in which certain operations are implemented on the reconfigurable hardware. The second research question is:

2. *How can I provide flexibility without requiring a new ISA to be developed?*

The answer lies in defining a one-time instruction-set insensitive extension to existing processor architectures that is able to control the reconfigurable hardware in terms of reconfiguring it to future implementations and controlling those implementations. The third research question is:

3. *How can I implement "arbitrary" program specified operations/functionalities?*

The answer lies in the proposal of a microarchitecture that is able to implement the one-time instruction-set extension discussed in research question 2 and that operate with emulation. Finally, the fourth research question is:

4. *How can I automatically target the "transformed" program to run on the reconfigurable computing platform?*

The answer lies in proposing compilation techniques that incorporate the new programming paradigm, target the new instruction-set extension, and is aware of the micro-architecture.

3. CANDIDATES FOR RECONFIGURATION

In this section, we present the general concept of transforming an existing program to one that can be executed on a reconfigurable computing platform. Consecutively, we investigate in more detail the methodology involved in this transformation that introduces "ultra complex" instructions.

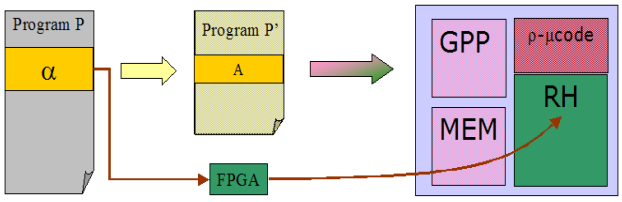

Figure 2. Program transformation example.

The conceptual view of how program P (intended to execute only on the general-purpose processor core) is transformed into program P' (executing on both the GPP core and the reconfigurable hardware) is depicted in Figure 2. The purpose is to obtain a functionally equivalent program P' from program P which (using specialized instructions) can initiate both the configuration processes of 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. eliminate the identified code and add code to have an "equivalent" code (A) assuming that A "calls" the hardware with functionality "α".
3. show hardware feasibility of "α" in a current technology (e.g., field-programmable gate array (FPGA)) and map "α" into reconfigurable hardware.
4. execute program P' with original code plus code having functionality A (equivalent to functionality "α") on the reconfigurable processor.

The mentioned steps illustrate the new programming paradigm in which both software and hardware descriptions are present in the same program. It should also be noted that because the only constraint on "α" is implementability, it is also implied that the microarchitecture has to support emulation. This implies the utilization of microcode. We have termed this as reconfigurable microcode (ρ-μcode) as it is different from that traditional microcode. The difference is that such microcode does not execute on fixed hardware facilities. It operates on facilities that itself "designs" to operate upon.

The methodology in obtaining a program for the reconfigurable computing platform is depicted in Figure 3. First, the code to be run 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 "hardware implementable". This can be determined manually or automatically (Cardoso and Neto (8)). The end result will be a new program that comprises the following elements:

- Repair code is inserted in order to communicate parameters and results to/from the reconfigurable hardware from/to the general-purpose processor core.
- "VHDL"-code and emulation code are inserted to configure the reconfigurable hardware to perform the functionality that is initialized by the "execute code".

Instead of inserting explicit code into the new program, each piece of code can be initialized by special "ultra complex" instructions. It should be noted that in the programming paradigm, software code co-exists in the program with hardware (implemented in reconfigurable fabric) descriptions.

4. ISA BEHAVIOR AND MICROARCHITECTURE

In the previous sections, we have highlighted how existing programs (to be executed on general-purpose processors) can be transformed and then executed on a reconfigurable computing platform. We have argued that in order to achieve this, a new programming paradigm is needed. Consequently, we have shown a methodology to manually or (semi)-automatically obtain the new program. In this section, we present the MOLEN architecture (Vassiliadis et al (6)) that is able to support the new paradigm. Without

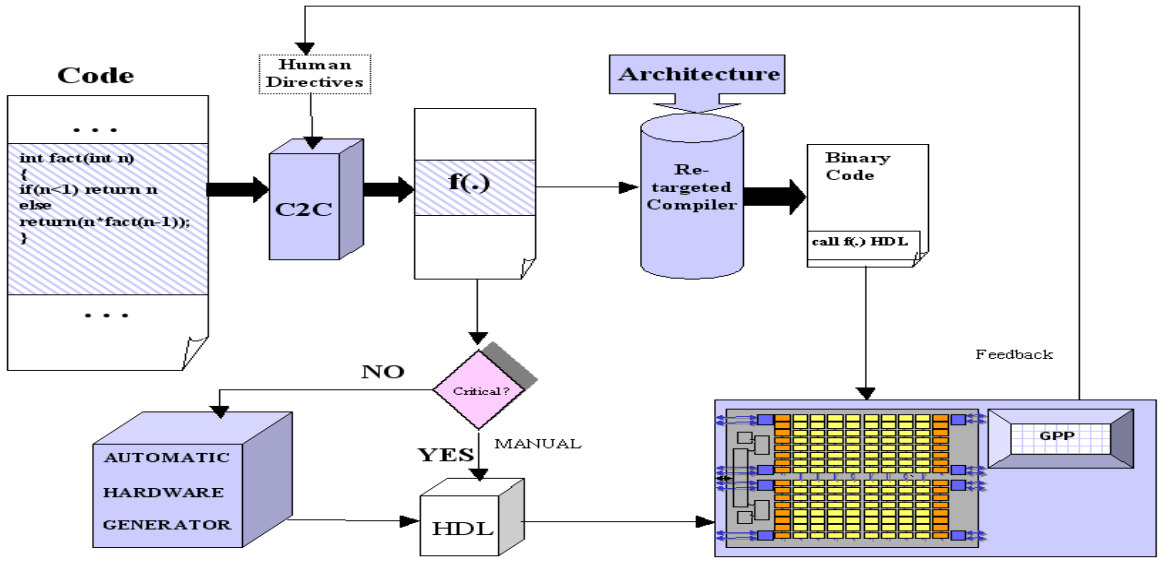

Figure 3. Program transformation methodology for reconfigurable computing.

delving into too much detail[1], we highlight two important instructions (out of a possible 8) in this presentation, namely the *set* and *execute* instructions. The *set* instruction initializes the reconfiguration process on the reconfigurable hardware while the *execute* instruction initializes the execution process on the reconfigurable hardware.

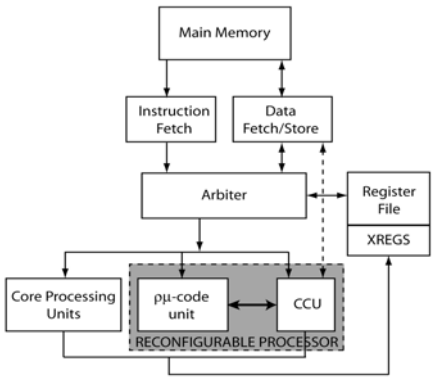

Figure 4. The MOLEN architecture.

In its most general form, the MOLEN machine organization, which is augmented with a reconfigurable hardware processor, is depicted in Figure 4. In this organization, instructions are fetched from the main memory and are temporarily stored in the 'Instruction Fetch' unit. Subsequently, these instructions are fetched by the 'Arbiter' which decodes them before issuing them to their corresponding execution units. Instructions that have been implemented in fixed hardware are issued to the 'Core Processing Units', i.e., the regular functional units such as ALUs, multipliers, and dividers. The *set* and *execute* instructions relate to the reconfigurable processor and are issued to it accordingly. More specifically in our case, they are issued to the reconfigurable microcode unit or 'ρμ-code unit'. As explained later, it provides fixed and pageable storage for reconfiguration and execution microcode that control the reconfiguration and execution processes on the 'Custom Configured Unit' (CCU), respectively. The loading of microcode to the 'ρμ-code unit' is performed via the 'Arbiter', which accesses the main memory through the 'Data Fetch/Store'-unit.
Similar to other load/store architectures, the proposed machine organization executes on data that is stored in the register file and prohibits direct memory data accesses by hardware units other than the load/store unit(s). However, there is one exception to this rule, the custom configurable unit (CCU) is also allowed direct memory data access via the 'Data Fetch/Store' unit (represented by a dashed two-ended arrow). This enables the CCU to perform much better when streaming data accesses are required, e.g., in multimedia processing. Finally, we introduce the exchange registers (XREGS) which are utilized to accommodate the mentioned input and output interface that is needed to communicate arguments and results between the implemented function(s) and the remainder of the application code. When only a small amount of data needs to be communicated, the register file suffices. However, by architecturally including the exchange registers, a more general communication framework is provided in order to communicate an arbitrary number of arguments and results.
As mentioned earlier, the *set* and *execute* instructions initialize the reconfiguration and execution processes on the CCU. It must be clear that such processes are complex in nature. Therefore, they must be emulated and thus microcoded (Vassiliadis et al (7)).

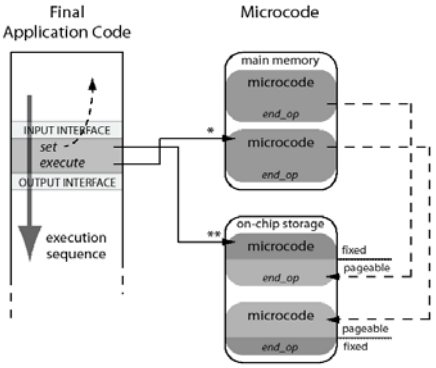

Figure 5. Microcode invocation.

Furthermore, it is impractical to specify these two instructions for each and every operation that needs to be implemented on the CCU. Therefore, these instructions point to the needed microcode that is stored in either the main memory or in an on-chip storage facility. This is illustrated in Figure 5. The architectural descriptions of both instructions are depicted in Figure 6. It should be noted that in Figure 5, there is a dashed arrow in the final application code expressing the fact that *set* and *execute* instructions can be moved with the appropriate compile interface so that reconfiguration can be overlapped with the execution of the general-purpose code.

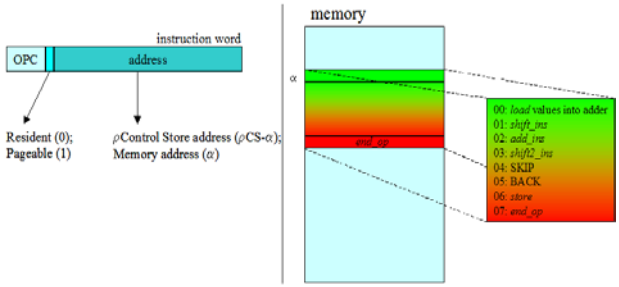

Figure 6. Architectural support.

[1] A complete description of the instructions and their sequence control can be found in Vassiliadis et al (5).

In Figure 6, the opcode specifies the instruction. The ensuing bit determines whether the microcode is located in the main memory or inside an on-chip storage. Depending on this bit, the interpretation of the remaining bit is either a main memory address (α) or a ρControl Store address (ρCS-α). It should be noted that for a single program (assuming 32-bit instructions) $2^{32-(OPC+1)}$ reconfigurable operations can be supported providing the programmer with an almost arbitrary emulation capability. The ρControl Store is a specific storage facility within the ρμ-code unit. Also depicted in Figure 6 is an example of such microcode stored in the main memory at address α. It must be noted that in order to delimit the boundaries of such microcode, an *end_op* microinstruction is utilized to denote the end of microprograms.

The on-chip storage of microcode inside the ρμ-code unit can be subdivided into two types of storage, namely fixed and pageable (depicted in Figure 7). The fixed storage is intended to store frequently used microcode in order to substantially diminish the loading time of such microcode. Finally, the pageable storage provides temporary storage for less frequently used microcode that is located in the main memory. Like regular caches, its purpose is to diminish the loading time of such microcode. It must be noted that the pageable storage is different from regular caches.

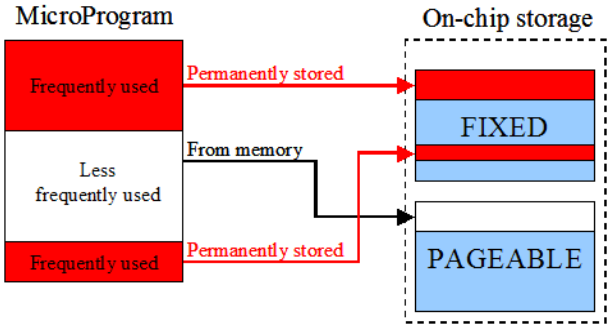

Figure 7. Fixed versus pageable microcode storage.

5. COMPILER AND EVALUATION

In this section, we present in detail the compiler mechanisms and extensions required to implement the Molen programming paradigm. The compiler needs information regarding I/O parameter passing to the reconfigured operations, the addresses indicating where the microcode resides and dependencies in terms of area or data between multiple reconfigured operations. We assume for simplicity in the rest of the paper that this information is provided in the form of a hardware description file (HDF). The current compiler system builds on the Stanford SUIF2 (Stanford University Intermediate Format) Compiler Infrastructure for the front-end, while the back-end is built over the framework offered by the Harvard Machine SUIF. The last component has been designed with retargetability in mind. It provides a set of back-ends for general-purpose processors, powerful optimizations, transformations and analysis passes. These are essential features for a compiler targeting a reconfigurable computing platform. We have currently implemented the proposed extensions for the x86 processor and are working on a PowerPC version to be used on the Xilinx Virtex II pro. Some specifics of the compiler regards it dealing with the exchange register and set-execute and are described in the following.

The Exchange Registers As explained above, exchange registers (XREGs) are used to pass parameters to the reconfigurable hardware and returning the computed values after the operation execution. These registers guarantee independence between the reconfigurable processor and the GPP as they receive their data directly from the general-purpose registers (GPRs). To this purpose, *movetx* and *movefx* instructions have to be provided to respectively put data from the GPR to these XREGs and from the XREG to the GPR. For each implemented function, the HDF specifies what exchange register is associated with what function. This register will then contain the register number where the compiler will put the parameters and where the result to be returned by the compiler can be found. All parameters of an operation will then be allocated by the compiler in consecutive XREGs forming a block of XREGs.

SET-EXECUTE Configuring the reconfigurable hardware, i.e., the CCU, is extremely expensive in terms of cycles and therefore the compiler needs to hide this latency by starting it as soon as possible in the execution flow. The reconfiguration microcode, residing in a location specified in the HDF, will be called by the appropriate *set* instruction. During the execution phase, the defined microcode is responsible for taking the parameters of its associated operation from XREGs and returning the result(s). A single *execute* instruction does not pose any specific challenge, because the whole set of exchange registers is available. However, when executing multiple *execute* instructions in parallel, a number of issues need to be resolved first. Only functions without data dependency and no area overlap can be executed in parallel. The former can be (partially) resolved by applying dependency analysis, but area overlap is information, which again needs to be stored in the HDF.

An example of the code generated by the extended compiler for the Molen programming paradigm is presented in Figure 8. In the left part, the original C program is given. 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 HDF. In the middle part of the picture,the code generated by the original compiler for the C program is depicted. The pragma annotation is ignored and a normal function call is included. The right part of the picture presents the code generated by the compiler extended for the Molen

programming paradigm; the function call is replaced with the appropriate instructions for sending parameters to the reconfigurable hardware in XREGs, hardware reconfiguration, preparing the fix XREG for the microcode of the *execute* instruction, execution of the operation and the transfer of the result back to the GPP. The presented code is at medium intermediate represenation level and the register allocation pass has not been applied.

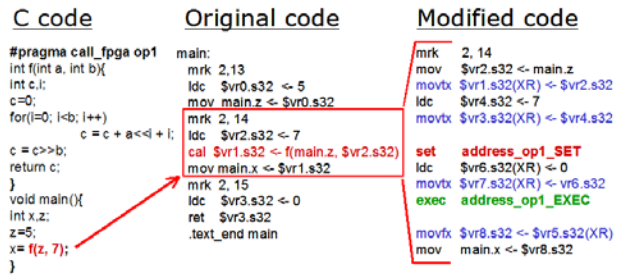

Figure 8. Code example.

In Figure 9, we present some experimental results based on running the *mpeg2enc* benchmark on a modified MIPS-based simulator by including support of the Molen architecture. In addition, we have shown the implementability of the targeted operations (DCT and SAD) and normalized their execution cycles to that of the general-purpose processor.

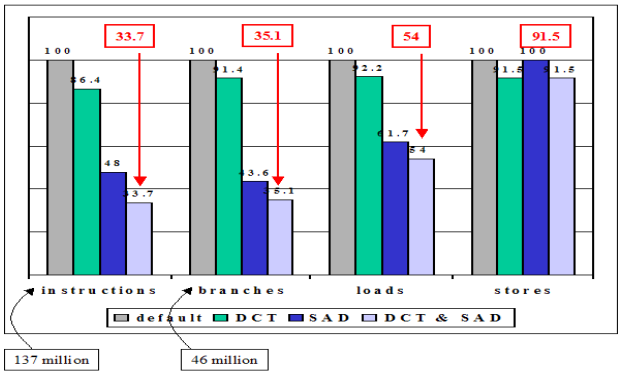

Figure 9. Experimental results.

In this figure, we can observe that the number of executed instructions and branches are greatly reduced (to about one third of the original). Furthermore, by better exploiting the memory bandwidth and temporary storage of intermediate results in the reconfigurable hardware, the number of loads can also be greatly reduced.

6. CONCLUSIONS

In this paper, we presented a unified approach for reconfigurable computing mixed with general-purpose computing. We described a programming paradigm, an architectural instruction set insensitive extension to support the programming paradigm, a generic microarchitecture based on emulation, and a compilation methodology to support the proposed polymorphic processing scheme.

REFERENCES

1. M.B. Gokhale and J.M. Stone , 1998, "Napa C: Compiling for a Hybrid RISC/FPGA Architecture", Proc. IEEE Symp. on Field-Programmable Custom Computing Machines, pp. 126-137.

2. S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, 1997, "The Chimaera Reconfigurable Functional Unit", Proc. IEEE Symp. on Field-Programmable Custom Computing Machines, pp.87-96.

3. A. La Rosa, L. Lavagno, and C. Passerone, 2003, "Hardware/Software Design Space Exploration for a Reconfigurable Processor", Proc. of the DATE 2003, 00. 570-575.

4. M. Sima, S. Vassiliadis, S.D. Cotofana, J.T. van Eijndhoven, and K. Vissers, "Field-Programmable Custom Computing Machines – A Taxonomy", 12th Int. Conf. on Field Programmable Logic and Applications (FPL2002), pp. 79-88.

5. S. Vassiliadis, G.N. Gaydadjiev, K. Bertels, and E. Moscu Panainte, 2003, "The Molen Programming Paradigm", Proc. of the 3rd Int. Conf.Workshop on Systems, Architectures, Modeling, and Simulation, pp. 1-7.

6. S. Vassiliadis, S. Wong, and S.D. Cotofana, 2001, "The Molen ρμ-coded Processor", 11th Int. Conf. on Field-Programmable Logic and Applications (FPL2001), Springer-Verlag Lecture Notes in Computer Science (LNCS), vol. 2147, pp. 275-285.

7. S. Vassiliadis, S. Wong, and S.D. Cotofana, 2003, "Microcode Processing: Positioning and Directions", IEEE Micro, vol. 23, no. 4, pp. 21-30.

8. J.M.P. Cardoso and H.C. Neto, 2003, "Compilation for FPGA-Based Reconfigurable Hardware", IEEE Design & Test of Computers, pp. 65-75.

Direct questions and comments about this paper to Stamatis Vassiliadis, Computer Engineering Laboratory, Electrical Engineering Department, Delft University of Technology, Delft, The Netherlands; S.Vassiliadis@ewi.tudelft.nl.