Compiling for the Molen Programming Paradigm

Elena Moscu Panainte[1], Koen Bertels[1], and Stamatis Vassiliadis[1]

Computer Engineering Lab
Electrical Engineering Department, TU Delft, The Netherlands
{E.Panainte,K.Bertels,S.Vassiliadis}@et.tudelft.nl

Abstract. In this paper we present compiler extensions for the Molen programming paradigm, which is a sequential consistency paradigm for programming custom computing machines (CCM). The compiler supports instruction set extensions and register file extensions. Based on pragma annotations in the application code, it identifies the code fragments implemented on the reconfigurable hardware and automatically maps the application on the target reconfigurable architecture. We also define and implement a mechanism that allows multiple operations to be executed in parallel on the reconfigurable hardware. In a case study, the Molen processor has been evaluated. We considered two popular multimedia benchmarks: mpeg2enc and ijpeg and some well-known time-consuming operations implemented in the reconfigurable hardware. The total number of executed instructions has been reduced with 72% for mpeg2enc and 35% for ijpeg encoder, compared to their pure software implementations on a general purpose processor (GPP).

1 Introduction and Related Work

In the last decade, several approaches have been proposed for coupling an FPGA to a GPP. For a classification of these approaches the interested reader is referred to [1]. There are four shortcomings of current approaches, namely:

1. **Opcode space explosion**: a common approach (e.g. [2], [3], [4]) is to introduce a new instruction for each portion of application mapped into the FPGA. The consequence is the limitation of the number of operations implemented into the FPGA, due to the limitation of the opcode space. More specifically stated, for a specific application domain intended to be implemented in the FPGA, the designer and compiler are restricted by the unused opcode space.
2. **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 ([5], [6]). For example, in the architecture presented in [5], due to the encoding limits, the fragments mapped into the FPGA have at most 4 inputs and 2 outputs; also, in Chimaera [6], the maximum number of input registers is 9 and it has one output register.

P.Y.K. Cheung et al. (Eds.): FPL 2003, LNCS 2778, pp. 900–910, 2003.

3. No support for **parallel execution** on the FPGA of sequential operations: an important and powerful feature of FPGA's can be the parallel execution of sequential operations when they have no data dependency. Many architectures [1] do not take into account this issue and their mechanism for FPGA integration cannot be extended to support parallelism.
4. 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. That is a reconfigurable implementation developed by a designer A can not be included without substantial effort by the compiler developed for an FPGA implementation provided by a designer B.

A general approach is required that eliminates these shortcomings. In this paper, a programming paradigm for reconfigurable architectures [7], called the Molen Programming Paradigm and a compiler are described that offer alternatives and a solution to the above presented limitations.

The paper is organized as follows: in the next section, we discuss related research and present the Molen programming paradigm. We then describe a particular implementation, called the Molen processor that uses microcoded emulation for controlling the reconfigurable hardware. Consequently, we present the two main elements of the paper, namely the Exchange Register mechanism and the compiler extension for the Molen processor. We finally discuss an experiment comparing the Molen reconfigurable processor with the equivalent non-reconfigurable processor, using two well-known multimedia benchmarks: mpeg2 and ijpeg.

2 The Programming Paradigm

The Molen programming paradigm[7] is a sequential consistency paradigm for programming CCMs possibly including a general purpose computational engine(s). 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 added instructions include:

- Two instructions[1] for controlling the reconfigurable hardware, namely:
  - SET $<$ *address* $>$: at a particular location the hardware configuration logic is defined
  - EXECUTE $<$ *address* $>$: for controlling the executions of the operations on the reconfigurable hardware

[1] Actually, five if partial reconfiguration, pre-loading of reconfiguration and executing microcode are also explicitly assumed [7].

– Two move instructions for passing values of to and from the GPP register file and the reconfigurable hardware.

Code fragments constituted 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 stored in registers are passed to special reconfigurable hardware registers denoted as Exchange Registers(XRs). The Exchange Register mechanism will be described later in the paper. In order to maintain the correct program semantics, the code is annotated and CCM description files provide the compiler with implementation specific information such as the addresses where the SET and EXECUTE code are to be stored, the number of exchange registers, etc. It should be noted that this programming paradigm allows modularity, meaning that if the interfaces to the compiler are respected and if the instruction set extension (as described above) is supported, then:

– custom computing hardware provided by multiple vendors can be incorporated by the compiler for the execution of the same application.
– the application can be ported to multiple platforms with mere recompilation.

Finally, it is noted that every user is provided with at least $2^{(n-op)}$ directly addressable functions, where n represents the instruction length and 'op' the opcode length. The number of functions can be easily augmented to an arbitrary number by reserving opcode for indirect opcode accessing. From the previous discussion, it is obvious that the programming paradigm and the architectural extensions resolve the aforementioned problems as follows:

– There is only a one time architectural extension of few new instructions to include an arbitrary number of configuration.
– The programming paradigm allows for an arbitrary (only hardware real estate design restricted) number of I/O parameter values to be passed to/from the reconfigurable hardware. It is only restricted by the implemented hardware as any given technology can (and will) allow only a limited hardware.
– Parallelism is allowed as long as the sequential memory consistency model can be guaranteed.
– Assuming that the interfaces are observed, modularity is guaranteed because the paradigm allows freedom of operation implementation.

Parallelism and Concurrency: As depicted in Figure 1, the split-join programming paradigm suggests that the SET instruction does not block the GPP because it can be executed independently from any other instruction. Moreover, a block of consecutive resource conflict free SET instructions (e.g. set op1,set op2 in our example) can be executed in parallel. However, the SET-instruction (set op3) following a GPP-instruction can only be executed after the GPP-instruction is finished. As far as the EXECUTE-instruction is concerned, we distinguish between two distinct cases, one that adds a new instruction and one that does not:

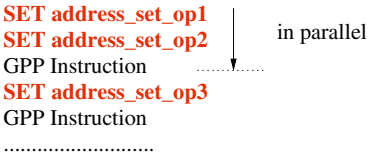

Fig. 1. SET instructions performed concurrently with GPP instructions

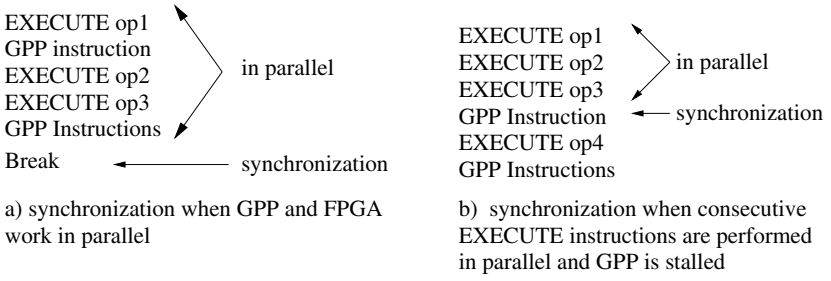

Fig. 2. Models of synchronization

1. If it is found that there is a substantial performance to be gained by parallel execution between GPP and FPGA, then the GPP and EXECUTE-instructions can be issued and executed in parallel. The sequence of instructions performed in parallel is initiated by an EXECUTE instruction. The end of the parallel execution requires an additional instruction (BREAK in the example) indicating where the parallel execution stops (see Figure2 (a)). A similar approach can be followed for the SET instructions.
2. If such performance is not to be expected (which will most likely be the case for reconfigured "complex" code and GPP code with numerous data dependencies), then a block of EXECUTE-instructions can be executed in parallel on the FPGA while the GPP is stalled. An example is presented in Figure 2(b) where the block of EXECUTE instructions which can be processed in parallel contains the first three consecutive EXECUTE instructions and it is delimited by a GPP instruction.
   We note that parallelism is guaranteed by the compiler, that checks whether there are data dependencies and whether the parallel execution is supported by the reconfigurable unit. Moreover, if the compiler detects that a block of SET/EXECUTE instructions cannot be performed in parallel, it separates them by introducing appropriate instructions. In the remaining of the paper, we assume that the separating instruction for SET/EXECUTE is a GPP instruction.

The Molen Reconfigurable Processor: The Molen $\rho\mu$-coded processor has been designed having in mind the programming paradigm previously presented. The Molen machine organization is depicted in Figure 3.

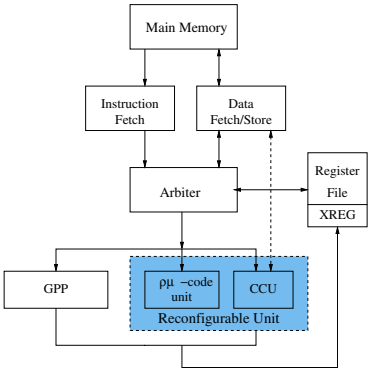

Fig. 3. The Molen machine organization

The arbiter performs a partial decoding of the instructions fetched from the main memory and issues them to the corresponding execution unit. The parameters for the FPGA reside in the Exchange Registers. In the Molen approach, an extended microcode - named reconfigurable microcode - is used for the emulation of both SET and EXECUTE instructions. The microcode is generated when the hardware implementation for a specific operation is defined and it cannot be further modified.

3 Compiler Extensions

In this section we present in detail the mechanism and compiler extensions required to implement the Molen programming paradigm.

The Exchange Registers: The Exchange Registers are used for passing operation parameters to the reconfigurable hardware and returning the computed values after the operation execution. In order to avoid dependencies between the RU and GPP, the XRs receive their data directly from the GPP registers. Therefore, move instructions have to be provided for this communication.

During the EXECUTION phase, the defined microcode is responsible for taking the parameters of its associated operation from XRs and returning the result(s). A single EXECUTE does not pose any specific challenge because the whole set of exchange registers is available. However, when executing multiple EXECUTE instructions in parallel, the following conventions are introduced:

- All parameters of an operation are allocated by the compiler in consecutive XRs and they form a block of XRs.
- The (micro)code of each EXECUTE instruction has a fixed XR, which is assigned when the microcode is developed. The compiler places in this XR a link to the block of XRs where all parameters are stored. This link is the number of the first XR in the block.

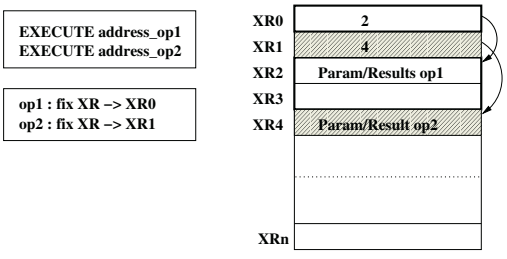

Fig. 4. Exchange Registers allocation by the compiler

Based on these conventions, the parameters for all operations can be efficiently allocated by the compiler and the (micro)code for each EXECUTE instruction is able to determine the associated block of parameters. An example is presented in Figure 4, where two operations, namely *op1* and *op2,* are executed in parallel. Their fix XRs (XR0 and XR1) are communicated to the compiler in a FPGA description file. As indicated by the number stored in XR0, the compiler allocates for operation *op1* two consecutive XRs for passing parameters and returning results, namely XR2 and XR3. The operation *op2* requires only one XR for parameters and results, which in the example is XR4, as indicated by the content of XR1.

Compiler Extensions: The compiler system relies on the Stanford SUIF2[8] (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[9]. The last component has been designed with retargetability in mind. It provides a set of back-ends for GPPs, powerful optimizations, transformations and analysis passes. These are essential features for a compiler targeting a CCM. We have currently implemented the following extensions for the x86 processor:

- 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 [2]). 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 MIR (Medium Intermediate Representation) level and LIR (Low Intermediate Representation) level.
- Register file extension: the Register File Set has been extended with the XRs. The register allocation algorithm allocates the XRs in a distinct pass applied before the GPR allocation; it is introduced in Machine SUIF, at LIR level. The conventions introduced for the XRs are implemented in this pass.
- Code generation: code generation for the reconfigurable hardware (as previously presented) is performed when translating SUIF to Machine SUIF IR, and affects the function calls marked in the front-end.

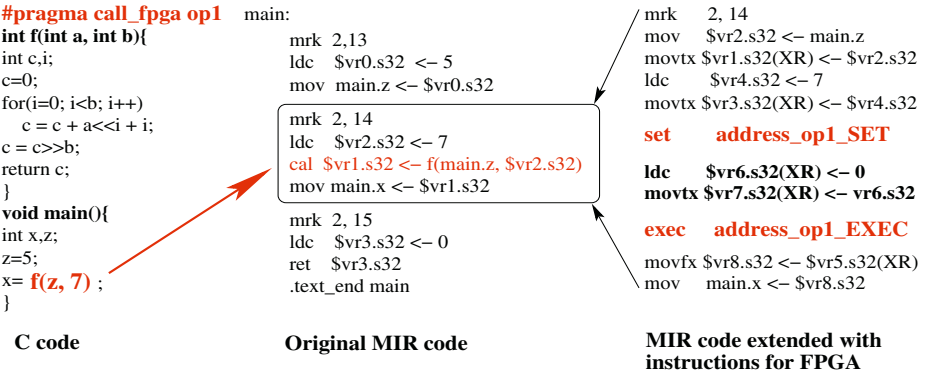

Fig. 5. Code Generation at MIR level

An example of the code generated by the extended compiler for the Molen programming paradigm is presented in Figure 5. In the first part, the 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 description file. In the central 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 last 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 XRs, hardware reconfiguration, preparing the fix XR 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 MIR level and the register allocation pass has not been applied.

The compiler extracts from a description file the information about the target architecture such as microcode address of SET and EXECUTE instructions for each operation implemented in the reconfigurable hardware, the number of XRs, the fix XR associated with each operation, etc.

4 A Case Study

In order to evaluate the performance improvements provided by the Molen processor, we used two well-known multimedia benchmarks, namely *mpeg2enc* and *ijpeg* for which we perform a pure software analysis. We made the following assumptions:

- the parts of the applications which can be implemented in the reconfigurable hardware are isolated in functions. This constitutes the base model for the comparison between the GPP and the Molen processor;
- the input data are:

- for mpeg2enc: the frames included in the benchmark
- for ijpeg: specmun, 1024 * 688

The parts of the applications that are candidates for the reconfigurable hardware implementation are the well-known time-consuming multimedia operations[7]: SAD (sum of absolute-difference), DCT (2 dimensional discrete cosine transform), IDCT (inverse DCT) and VLC (variable length coding). In order to study the performance improvements, we use the *Halt* library[10] available in Machine SUIF and which we modified to suit our purpose. This library is an instrumentation package that allows the compiler to change the code of the program being compiled in order to collect information about the program own behavior (at run-time).

For the above considered applications, the following is measured for their pure software implementation on the GPP (x86):

– The exact types and numbers of instructions - generated by the compiler- which are executed in the whole application and in each chosen function for hardware implementation plus their exact number of calls
– The number of cycles for the whole application and for each function chosen for hardware implementation

Based on these data, the following information can be computed for the Molen reconfigurable processor:

1. The code reduction as a result of implementation of parts of the application in reconfigurable hardware
2. An approximation of the maximum performance improvement of processor cycles for the whole application and for a particular implementation of one operation

However, because we lack a real implementation of the Molen processor, we cannot yet provide the second set of data for a particular implementation. We therefore restrict ourselves to indicating what functions are most likely to yield the highest performance improvement.

We introduced an additional pass in order to instrument the basic blocks of a program with the number and type of the included instructions. We also developed two sets of run-time analysis routines. The first set of routines is used to collect the type and number of instructions executed in the whole application and each specific function; it uses the instrumentation pass previously mentioned in this section. The second set of run-time analysis routines provides the number of cycles spent in the whole program or in a specific function. The measurements for the processor clock cycles have been performed on a Pentium II at 300MHz and we used the Pentium benchmarking instruction RDTSC - Read Time Stamp Counter - which returns the number of processor clock cycles since the CPU was reset. In this manner, the finest granularity is achieved (the code instrumentation does not affect the results).

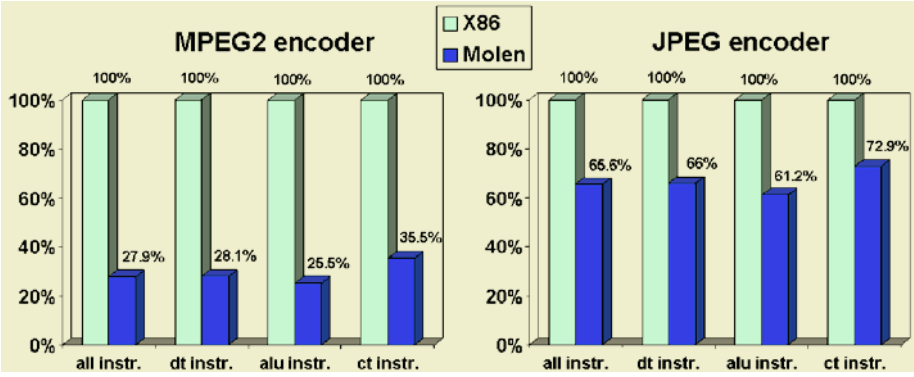

Fig. 6. mpeg2enc and ijpeg encoder instruction results

For example, we compute the total number of instructions executed in the Molen approach, when the above described functions have been implemented in hardware as follows:

$$n_{all(MOLEN)} = n_{all(GPP)} - \sum_{i=1}^{N} n_{f_i(GPP)} + \sum_{i=1}^{N} n_i(N_{call_i(MOLEN)} - N_{call_i(GPP)})$$

where n_{all} is the number of all instructions executed by the application, N is the number of functions implemented on the reconfigurable hardware, n_{f_i} represents the total number of instructions executed in the function f_i for all its calls, n_i is the number of calls of function f_i and N_{call} is the fixed number of instructions used for passing parameters, function call .

The measured data for the GPP alone and the computed data for the Molen processor are compared for mpeg2enc and ijpeg in Figure 6. The most important categories of instructions have been considered, namely data transfer (dt) instructions, arithmetic and logical (alu) instructions and control transfer(ct) instructions. From these pictures, a substantial reduction of the number of instructions is achieved by the Molen reconfigurable processor compared to the GPP: 72.1% for mpeg2enc and 34.4 % for ijpeg encoder. Also it is obvious that in both cases the alu instructions are the most reduced category of instructions, while the ct instructions are the least reduced instructions. This conclusion is confirmed by the inspection of the function code since it contains a large number of arithmetical computation and only a small number of branches. In table 1, the cycle measurements are reported. From these results, we can identify those functions that potentially give the highest performance improvement, given an efficient hardware implementation. The numbers suggest that the SAD function is the most promising candidate for hardware implementation, while the rest of the functions can provide at best a moderate performance improvement.

Table 1. mpeg2enc (left) and ijpeg encoder (right) cycle result

Fct	Cycles	% Total
SAD	149.947.461	55.2 %
DCT	42.529.647	15.7 %
VLC	3.946.954	1.4 %
IDCT	3.693.986	1.36 %
mpeg2enc Application	271.616.655	100 %

Fct	Cycles	%Total
DCT	40.206.773	12.5 %
VLC	36.571.622	10.5 %
ijpeg enc Application	341.316.466	100 %

5 Conclusions

In this paper, we presented the Molen Set-Execute paradigm that addresses a number of previously unresolved issues such as parameter passing and parallel execution of operations into the reconfigurable hardware. The paradigm involves the instruction set extension and requires on behalf of the FPGA developers only the address where the configuration(SET) and execution(EXECUTE) code is stored. A particular architectural implementation was presented, where the microcoded emulation of the SET and EXECUTE instructions are included.

The compiler extensions allow to generate code where the functions mapped on the reconfigurable hardware are automatically (rather then manually) substituted by the appropriate SET-EXECUTE instructions. It has been shown through experimentation that the compiler can be used as an important tool to support the design process focusing on the identification of good candidates for the reconfigurable hardware implementation. The presented results show a substantial reduction of the executed number of instructions and potential reduction of processor cycles for two multimedia benchmarks for their execution on the Molen reconfigurable processor compared to their pure software implementation on the GPP.

References

1. M. Sima, S. Vassiliadis, S.Cotofana, J. van Eijndhoven, and K. Vissers, "Field-Programmable Custom Computing Machines – A Taxonomy," in *12th International Conference on Field Programmable Logic and Applications (FPL)*, Montpellier, France, Sep 2002, pp. 79–88.
2. 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. 126–137.
3. S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, "The Chimaera Reconfigurable Functional Unit," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, Napa, California, 1997, pp. 87–96.
4. A. L. Rosa, L. Lavagno, and C. Passerone, "Hardware/Software Design Space Exploration for a Reconfigurable Processor," in *Proc. of the DATE 2003*, 2003, pp. 570–575.

5. F. Campi, R. Canegallo, and R. Guerrieri, "IP-Reusable 32-Bit VLIW Risc Core," in *Proc. of the 27th European Solid-State Circuits Conference*, Villah, Austria, Sep 2001, pp. 456–459.
6. Z. 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.
7. S. Vassiliadis, S. Wong, and S. Cotofana, "The MOLEN $\rho\mu$-Coded Processor," in *11th International Conference on Field Programmable Logic and Applications (FPL)*, Springer-Verlag LNCS, vol. 2147, Belfast, UK, Aug 2001, pp. 275–285.
8. http://suif.stanford.edu/suif/suif2.
9. http://www.eecs.hardward.edu/hube/research/machsuif.html.
10. M.Mercaldi, M. D. Smith, and G. Holloway, "The Halt Library," in *The Machine-SUIF Documentation Set*, Hardvard University, 2002.