The Molen Programming Paradigm

Stamatis Vassiliadis, Georgi Gaydadjiev, Koen Bertels, Elena Moscu Panainte
Computer Engineering Laboratory,
Electrical Engineering Department, EEMCS
TU Delft
E-mail : {stamatis, georgi, koen, elena}@ce.et.tudelft.nl
http://ce.et.tudelft.nl

***Abstract*— In this paper we present the Molen programming paradigm, which is a sequential consistency paradigm for programming Custom Computing Machines (CCM). The programming paradigm allows for modularity and provides mechanisms for explicit parallel execution. Furthermore it requires only few instructions to be added in an architectural instruction set while allowing an almost arbitrary number of op-codes per user to be used in a CCM. A number of programming examples and discussion is provided in order to clarify the operation, sequence control and parallelism of the proposed programming paradigm.**

I. Introduction

Since the mid nineties *Reconfigurable Computing* (RC) is becoming increasingly popular computing paradigm which refers to the ability of the software to transform the hardware platform underneath on a per-application basis. Computing systems built according to the *Custom Computing Machines* paradigm usually consist of a *General Purpose Processor* (GPP) and reconfigurable unit(s) possibly implemented in an FPGA technology. The software and architectural support needed for CCM remains problematic. Programming CCMs usually implies the introduction in the software design flow of detailed knowledge about the reconfigurable hardware. The compiler plays a significant role in the software design flow as it has to integrate most of this information. Computational-intensive operations are usually implemented on the reconfigurable hardware provided by different vendors and the challenge is to integrate them - whenever possible - in new or existing applications. Such integration is only possible when application developers as well as hardware providers adopt a common programming paradigm.

In this paper we present such a programming paradigm. The main contributions of the paper are:

- The presentation of a programming model for reconfigurable computing that allows modularity, general ”function like” code execution and parallelism in a sequential consistency computational model,
- The definition of a minimal ISA extension to support the programming paradigm. Such an extension allows the mapping of an arbitrary function on the reconfigurable hardware with no additional instruction requirements,
- The introduction of a mechanism allowing multiple operations to be loaded and executed in parallel on the reconfigurable hardware,
- Support for the application portability to multiple reconfigurable platforms.

This paper is organized as follows: in Section II the shortcomings of the existing reconfigurable programming paradigms are highlighted. Section III introduces the Molen programming paradigm and shows how the indicated shortcomings are addressed. Section IV discusses the sequence control of the programming paradigm. The discussion is supported by a wide range of examples. Section V concludes the discussion.

II. Problems with existing RC programming paradigms

In the last decade, many different approaches have been proposed for programming FPGA / GPP combinations. The architecture of a computer, reconfigurable or not, is the minimal behavioral specification- *behavioral* so that the software can be written, *minimal* so that the widest possible range of excellence criteria can be chosen for implementations [1]. It is the conceptual structure and functional behavior as seen by the user (usually the programmer). The conceptual structure, e.g. instruction set, is determined mainly by the system functional requirements. In addition many other system requirements, e.g. power consumption, can have impact on the computer architecture. As in the case of traditional computer architecture, the *Reconfigurable Computing* (RC) paradigm strongly relies on the compilation process. The compiler plays an important role in code generation and needs to be extended with the knowledge about the RC extension architecture. In order to introduce the CCM to the instruction set architecture, the opcode space is extended. This is done by introducing new ”super instructions” to operate the CCM from the software. It is obvious that those new instructions will utilize the ”not-used” opcode space of the targeted ISA.

Four major shortcomings of existing RC programming approaches are indicated by the following:

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.

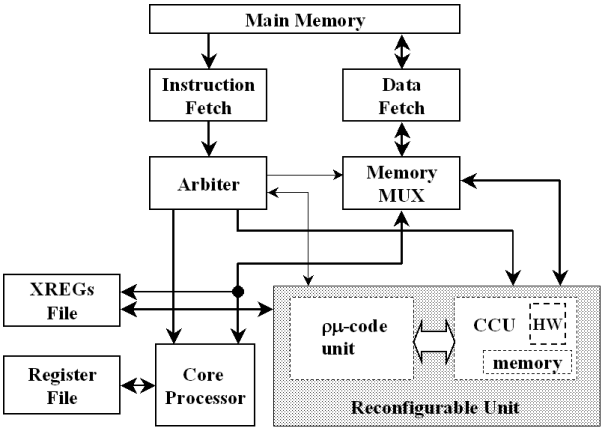


Fig. 1. The Molen machine organization

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.
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 (see for examples in [7]) 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. 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.

Due to these limitations, the implementation of the same operation on the same type of FPGA is different for each specific approach. Moreover, some reconfigurable architectures may not include a particular operation in the reconfigurable hardware due to some specific restrictions.

III. The Molen Programming Paradigm

In this section first the Molen machine organization [8] is briefly introduced. Consequently the Molen programming paradigm is described and it is shown how it addresses the major shortcomings of the RC paradigms as indicated in Section II. To clarify the discussion, a variety of examples will be supplemented.

The main Molen components as depicted in Figure 1 are: the Core Processor - which is a GPP, and the *Reconfigurable Unit* (RU) - implemented in the FPGA. The Arbiter performs

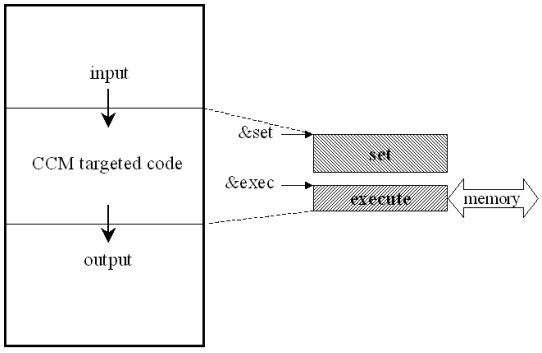


Fig. 2. Molen interface

a partial decoding of the instructions fetched from the main memory and issues them to the corresponding execution unit (GPP or RU). The division in hardware and software part is directly mappable to the two units. The hardware targeted pieces are executed by the RU which is composed usually by three part low grain reconfigurable fabric, while the software (remaining) modules are executed on the GPP. The *Custom Computing Unit* (CCU) is the RU part that performs the hardware execution. It should be noted that a CCU can incorporate complex multiple hardware "functions". The conceptual idea of how a Molen application looks like is presented in Figure 2. First, there are clear boundaries between the software execution and the RU noted by **input** and **output**. One should think of predefined parameters (or pointers to parameters) to be passed to and back from the hardware CCU engine. Second the CCU configuration file is to be loaded into the configuration memory in order to prepare the hardware (the CCU part on Figure 1). Since different CCU engines will have different content and length of their configuration files a general approach is provided for loading arbitrary configuration to the reconfigurable part of RU. This is denoted by the SET phase (initiated by a SET instruction). The SET instruction requires single parameter - the beginning address of the *configuration microcode*. When a SET instruction is detected, the Arbiter will read every sequential memory address until the termination condition is met, e.g. end_op microinstruction is detected. The $\rho\mu$ code unit will then ensure that the data fetched from memory is redirected to the reconfiguration support unit, e.g. FPGA configuration memory. For an implementation see [9]. After completion of the SET phase, the hardware is ready to be used in the context of the targeted CCU functionality. This is done in the EXECUTE phase initiated by the EXECUTE instruction. This instruction also utilizes a single parameter being the address of the *execution microcode*. The execution microcode performs the real CCU operations, this is the hardware engine initialization, reading the input parameters, performing the targeted computation and writing the results to the output registers. The majority of the CCUs will additionally access the system memory while running (depicted with horizontal arrow in Figure 2). The input/output and SET/EXECUTE parameters (and information about the memory segments used by the CCU) are made available to the compiler through a CCU *description file*. This description file and the binary images of the configuration and the execution microcode are

sufficient for the compiler / linker to create adequate Molen binaries.

The above two generic phases SET/EXECUTE are emulated as an instruction is substituted by a sequence of micro-operations using an extension of microcode (referred also as *reconfigurable microcode*). In addition, Exchange Registers (XR) are used for passing operation parameters to and from the reconfigurable hardware at the beginning and the end of the operation execution. The XRs can receive their data directly from the GPP register bank. Therefore, the corresponding move instructions have to be provided. The number of XRs is implementation dependent.

The Molen programming paradigm is a sequential consistency paradigm for programming CCMs (reconfigurable processors) 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 complete list of eight required instructions is as follows:

- Six instructions are required for controlling the reconfigurable hardware, namely:
  - Two SET $< address >$ instructions: at a particular location the hardware configuration logic is defined. This operation will actually perform the hardware configuration as stored in memory from the referred address location. The information about the configuration microcode length is embedded inside the microcode itself. This operation can be additionally split into two sub-phases (accounting for the two set instructions):
    - partial SET (P-SET) to cover common and often used functions of an application or set of applications and to diminish time expensive reconfigurations, and
    - complete SET (C-SET) to deal with the remaining blocks (not covered by the p-set sub-phase) in order to *complete* the CCU functionality by enabling it to perform the less frequent functions.
  - EXECUTE $< address >$: for controlling the executions of the operations on the reconfigurable hardware. The address sequence referred by this instruction contains the microcode to be executed on the CCU configured in the SET phase. The microcode sequence is terminated by an end_op micro operation.
  - BREAK: this instruction may be needed in implementing explicit parallel execution between GPP and CCU if it is found to gain substantial performance with simultaneous execution. This operation is used for synchronization to indicate the parallel execution and setting boundaries. In addition BREAK can be used to express parallelism between two or more concurrent CCU units.
  - SET PREFETCH $< address >$: In implementations with a reconfigurable hardware of limited size, this instruction can be used by the compiler to pre fetch the SET microcode from main memory to a local (much faster) on chip cache or the $\rho\mu$ control store in order to minimize the reconfiguration time penalty.
  - EXECUTE PREFETCH $< address >$: the same reasoning as for the SET PREFETCH holding for the EXECUTE microcode.
- Two move instructions for passing of values to and from the GPP register file and the reconfigurable hardware. More specially:
  - MOVTX $XR_a \leftarrow R_b$: (move to X-REG) used to move the content of general purpose register R_b to XR_a.
  - MOVFX $R_a \leftarrow XR_b$: (move from X-REG) used to move the content of exchange register XR_b to GPP register R_a.

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 as introduced earlier. In order to maintain the correct program semantics, the code is annotated and CCU 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. The physical number of XRs, imposed by a specific implementation, is a limitation for the number of parameters that can be passed by value. This limitation can be avoided by applying passing by reference, a method that will be demonstrated later. 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.
- the designer is given the freedom to explore among several custom computing hardware designs implementing the same function and select the best one based on design constraints, e.g. speed or size.
- the design process can be speeded up by having different design teams work in parallel of the different parts of the system under development, e.g. various software and hardware pieces can be designed in parallel.

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 a few new instructions to include an arbitrary number of

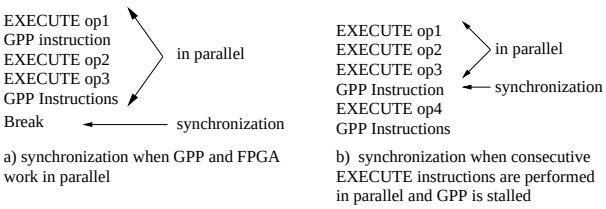
configurations.

- 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.

IV. Sequence Control

There are basically three distinctive cases with respect to the Molen instructions introduced earlier - the *minimal*, the *preferred* and the *complete case*. In more details they are as follows:

- **the minimal case:** This is essentially the smallest set of Molen instructions needed to provide a working scenario that supports execution of an arbitrary application (providing there is no shortage of hardware resources). The four basic instructions needed are SET, EXECUTE, MOVTX and MOVFX. By implementing the first two instructions (SET/EXECUTE) any suitable CCU can be loaded and executed in the reconfigurable unit. The MOVTX and MOVFX instructions are needed to provide the input/output interface between the Software (GPP code) and the Hardware (CCU engine) parts of the application. It should be noted that this case does not introduce any restrictions on parallel hardware execution. There can be more than one CCUs configured or running concurrently. The only difference is that in such cases the GPP will be stalled for the time needed for the ”slowest” CCU to complete its operation. An example is presented in Figure 3(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. The situation when a block of EXECUTE-instructions can be executed in parallel on the RU while the GPP is stalled, will most likely be the case for reconfigured ”complex” code and GPP code with numerous data dependencies. In addition, it should be noted that the above reasoning holds for the segments consiting of SET instructions or mixture of independent SET and EXECUTE instructions.
- **the preferred case:** The minimal case provides the basic support but may suffer from the time consuming reconfiguration times that can become prohibitive for some real-time applications. In order to address this issue the two additional SET sub-phases (P-SET) and (C-SET) are introduced for distinction among very often and least often used CCU functions. More specifically in the P-phase the CCU is partially configured to perform the common functions of an application, while the C-phase takes care only of the remaining (much smaller) set of less frequent functions. This allows the compiler to ”hide” the time consuming reconfiguration operations better. In addition, for the cases when the P-set and C-set improvements are not sufficient, the two prefetch instructions (SET and EXECUTE PREFETCH) are provided to allow additional freedom to the compiler instruction scheduler and further reduce the reconfiguration penalty.
- **the complete case:** In addition when it is found that there is a substantial performance to be gained by parallel execution between GPP and RU, then the GPP and the EXECUTE-instructions can be issued and executed in parallel. Since the GPP instructions (for the pertinent discussion see parallelism control discussed later) can not be used for synchronization any longer, the additional BREAK instruction is required. 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 3 (a)). The SET instructions are executed in parallel according to the same rules. It is clear that in case of an organization utilizing single GPP, the GPP instructions, present in the parallel areas, can not be executed in parallel. This is the most general Molen case that allows the highest instruction level parallelism (the fastest execution). On the other hand this approach is the most complicated in terms of covering issues such as asynchronous interrupts handling and memory management. These all are the Molen implementor responsibility to address and solve properly.



a) synchronization when GPP and FPGA work in parallel

b) synchronization when consecutive EXECUTE instructions are performed in parallel and GPP is stalled

Fig. 3. Models of synchronization

Compilation: The compiler [10] currently relies on the Stanford SUIF2 (Stanford University Intermediate Format) and the Harvard Machine SUIF back-end framework. The x86 processor has been considered as the GPP in the evaluated Molen organization in [10]. An example of the code generated by the extended compiler for the Molen programming paradigm is presented in Figure 4. In the left column, the original C program is shown. 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 CCU 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 standard function call is included. The right column 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

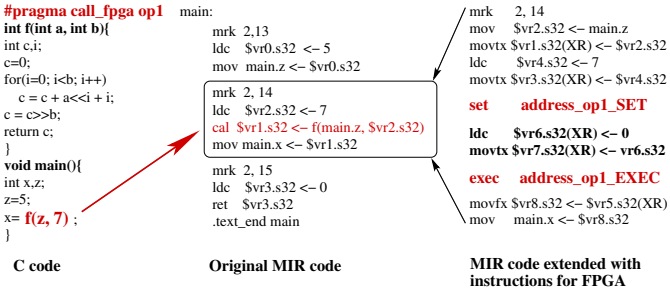


Fig. 4. Molen Code Generation

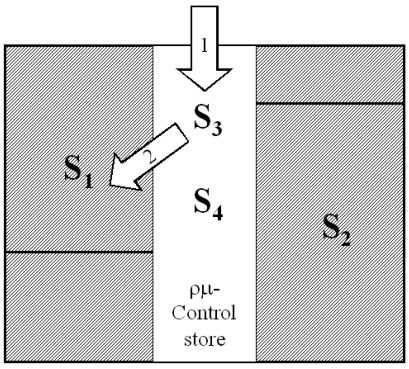


Fig. 5. SET PREFETCH instructions flow

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 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 XR, the fix XR associated with each operation, etc needed by the compiler is available in a CCU's description file.

Prefetching: The SET PREFETCH instructions behavior is illustrated in Figure 5. The arrow (1) represents the prefetching of microcode S_3 from memory to the $\rho\mu$ control store and (2) indicates the actual hardware reconfiguration. The control store area is used to move the prefetched microcode "near by" the reconfigurable hardware long before it is "needed". This instruction allows the compiler to schedule the time demanding load-from-memory operation. In Figure 6 the first $set\_prefetch$ operation will load the microcode positioned at address $\&op3\_set$ into the $\rho\mu$ control store. When the set $\&op3\_set$ instruction is processed, the S_3 microcode will be loaded from the control store (instead of the memory) into the hardware configuration memory, e.g. lookup tables, switch boxes and interconnect resources. In such a way the reconfiguration can be preformed faster than in the case when loading directly from memory. In case no more hardware resources are available, there should be a replacement strategy applied by the $\rho\mu$ control store, e.g. the least recently used (LRU) configuration will be replaced.

The prefetch instructions, as any other Molen instructions (except of the BREAK) are not limited with respect to concurrency. A Molen instruction initiates the code to be executed

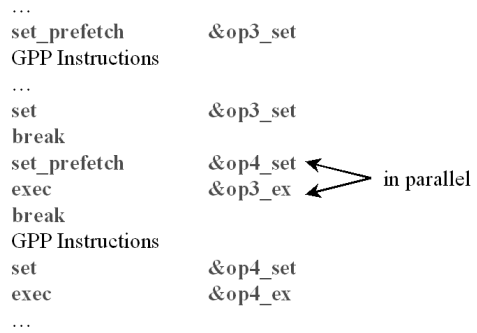


Fig. 6. Prefetching example

```
struct nod{                    struct nod{
   int no;                        int no;
   char buf[4000];                char buf[4000];
};                             };
.............                  .............
                               #pragma call_fpga op1
void f(struct nod a)           void f(struct nod * a)
{                              {
   .............                  .............
}                              }
.............                  .............
void main()                    void main()
{                              {
   struct nod root;               struct nod root;
   .............                  .............
   f(root);                       f( & root);
}                              }
```

Fig. 7. Passing parameters by reference in Molen

in parallel. As shown in Figure 6, the $set\_prefetch$ $\&op4\_set$ and the S_3 execution will be performed in parallel. The first $break$ instruction is needed to indicate that set and $exec$ instructions for S_3 cannot be executed in parallel and can be placed before or after the $set\_prefetch$ instruction for S_4. In the *preferred* Molen case the $break$ will be a GPP instruction, e.g. NOP.

Parameter exchange, Parallelism and Modularity: As shown earlier, the exchange register bank solves the limitation on the number of parameters as present in other RC approaches. The Molen XR bank can be used for passing parameters by value or by reference. When a limited number of parameters not exceeding the number of XRs is used, a straight forward passed-by-value strategy is applied. In case the number of parameters exceeds the XR bank size, the passing by reference should be used. It is obvious that passing by reference will allow an arbitrary (limited by the hardware) number of parameters to be exchanged between the calling (software) and called (hardware) functions. An example of how the system designers should modify their code in order to allow this in Molen programming paradigm is given in Figure 7.

It should be noted that the original code passes a copy of the structure to the called function. Since the size of the structure can exceed the XR bank size, e.g. 8, 16 registers, the pass by reference to the structure is used. This introduces, however, an additional feature - this is when the called function (now performed by the CCU in hardware) changes the structure contents, such changes will become "permanent" after the

```
#pragma call_fpga op1
int f( int x,  int y)
{
.............
}

#pragma call_fpga op2
int g(int x)
{
...........
}

int h(int a, int b, int c)
{
  int m,n, ...;
  m=f(a, b);
  n=g(c);
  ........
}
```

no data dependency

```
h:
    mov    a –> r1
    movtx r1 –>XR2
    mov    b –> r2
    movtx r2 –>XR3
    mov    c –> r3
    movtx r3 –> XR4
    set address_set_op1
    set address_set_op2
    ldc  2 –>r4
    movtx  r4 –>XR0
    ldc  4 –>r5
    movtx  r5 –>XR1
    execute address_ex_op1
    execute address_ex_op2
    movfx XR2 –> r6
    mov r6 –> m
    movfx XR4 –> r7
    mov r7 –> n
```

in parallel

Fig. 8. Parallel execution in Molen

control is returned back to the GPP. Such behavior will differ in functionality from the original software implementation. It is assumed that the system designers utilizing the Molen architecture will be aware about these side effects and will use passing by reference with extreme caution. The description file used as interface between the compiler and the CCU design should be extended with additional information about the structure.

The third shortcoming (parallel execution) indicated in Section II is addressed as follows. In case that two or more (generalized) functions, appointed 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 8. There is always a physical maximum of how many CCUs can be executed in parallel. This is, however, an implementation dependent issue, e.g. reconfigurable hardware size, CCU sizes, XR bank size etc. and can not be considered as a serious limitation, since it is not limited by the Molen architecture. The total execution time of the RU unit in the example of Figure 8 will be the longest execution time of both CCUs performing the functions f and g.

In addition to the above it should be emphasized that the Molen Hardware/Software (HW/SF) division ability is not limited to functions only. In case the targeted kernel is part of a function, e.g. a highly computational demanding loop, it can be isolated before its transformation to hardware implementation. The only two requirements are: a) to rewrite the kernel as a separate function, and b) to define a clear set of parameters as interface and pass 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 via input/output parameters only since both parts will execute in different contexts. An example of how such division can be done is depicted in Figure 9.

The Molen paradigm facilitates the modular system design. For instance CCM modules designed in a HDL (VHDL, Verilog or System-c) language are straight forward mappable to any FPGA technology, e.g. Xilinx or Altera. The only requirement is to satisfy the Molen SET and EXECUTE interface. In addition a wide set of functionally similar CCUs designs (from different providers), e.g. sum of absolute differences (SAD) or IDCT, can be collected in a database allowing

```
int n = 7;
void main(){
int  x, a, c, i;
a=5;
c=0;
for(i=0; i<n; i++)
   c = c + a<<i + i;
c = c>>n;
x=c;
}
```

```
int n = 7;
#pragma call_fpga op
int f(int a, int b){
int c,i;
c=0;
for(i=0; i<b; i++)
   c = c + a<<i + i;
c = c>>b;
return c;
}
void main(){
int x, a;
a=5;
x= f(a, n) ;
}
```

Fig. 9. Partial function exposing for HW implementation in Molen

```
...
for (i=0; i<N; i++) {
  c[i]=a[i]+b[i];
}
...
```

```
#pragma call_fpga op
void f(int *a, int *b, int *c, int num) {
 int i;
  for (i=0; i<num; i++) {
    c[i]= a[i] + b[i];
  }
}
...
f(a, b, c, N);
...
```

z-wide CCM

Fig. 10. Vector code CCU example

easy design space exploration.

Direct CCU Memory operations: Molen organizations are suitable for performing computation intensive operations in hardware while the data needed for the operations remains in main memory. A simple example of this is how vector addition calculation (**c** = **a** + **b**) can be moved to CCU implementation is presented in Figure 10. The three vectors **a**, **b** and **c** are contiguous vectors with the same length. In the beginning the original code needs to be restructured as introduced earlier to a separate function with usage of pointers. This process is shown by the right block arrow and prepares the input/output interfaces between the software and the CCU. More precisely in this case they are: the addresses of the first vector elements (for each vector), the number of elements to be processed and possibly the stride. The information on how the memory locations are accessed and the type of each vector element, e.g. integer, floating etc. should be taken into consideration in the CCU design. In essence, in this case, the CCU is an augmented vector architecture to include what is not performed (e.g. address generators) in the GPP. It is clear that function **f** can be parallelized in the hardware. Lets assume that the memory architecture allows access of $2 * z$ locations for read and z locations for write in parallel. This will allow z elements of the vector addition to be performed in parallel, hence speeding up the calculation by at least N/z. This is presented by the sliding window shown in Figure 11. It is clear that in this example the output interface of the CCU is just operation termination (execution of end_op). The result of the CCU is stored in the output vector **c** and can be used directly by the subsequent (hardware or software) modules.

For efficiency reasons, some operations that compute more than one value may require to return the results in the XRs (instead of memory, which is the straightforward way). These operations represent contiguous statements (in high-level pro-

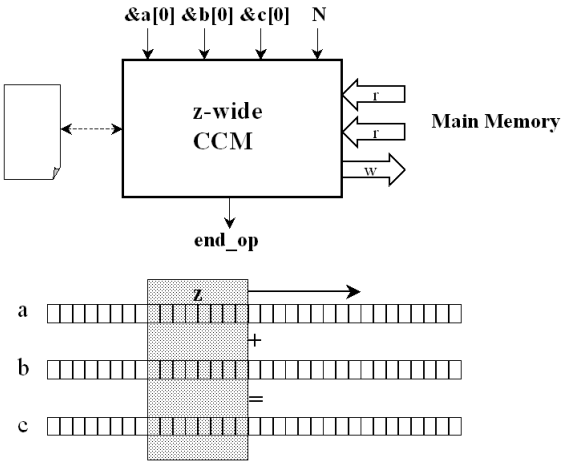


Fig. 11. Vector execution CCU example

```
#include <stdio.h>
int main(){
  int a,b,c,d,e;
  a=5;
  b=2;
  c=a+b;
  d=a–b;
  e=a*b;

  printf(" %d %d %d \n", c,d,e);
}
```

```
typedef struct {int c,d,e; } tip1;
tip1 f(int a, int b){
  tip1 x;
  x.c = a+b;
  x.d = a–b;
  x.e = a*b;
  return x;
}

int main()
  int a,b,c,d,e;
  tip1 s;
  a=5;
  b=2;
  s= f(a,b);
  c=s.c;
  d=s.d;
  e=s.e;

  printf(" %d %d %d \n", c,d,e);
}
```

Fig. 12. Multiple parameters from CCU example

gramming languages) where more variables are modified. In order to accommodate to the current approach the associated code has to be isolated in a function. The problem of returning more parameters from a function in XRs can be solved as illustrated in Figure 12. The computed values (for variables c, d, e in Figure 12(a)) are packed in a structure (in Figure 12(b)) in the new function body and unpacked after the new function call. We have to mention here that the CCU designer must be aware of the alignment and packed/unpacked conventions for structures assumed by the compiler.

Interrupts and miscellaneous considerations: In order to support GPP 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 GPP and CCU.
2) Hardware to communicate interrupts to GPP is implemented in CCU.
3) Initialization (via the GPP) of the appropriate routines for interrupt handling.

The compiler assumption is that the implementor of a reconfigurable hardware follows a co-processor type of configuration. The FPGA co-processor facility can be viewed as an extension of the GPP architecture. For examples of conventional architectural extensions that resemble the approach taken here see [11], [12], [13]. The above is a recommendation to the FPGA developer. If such an aproach is not followed and no ”clean” architecture is implemented some complications may arise. Furthermore, the unconstrained memory access of the CCUs in Molen as in any other architectural or programming paradigm can lead to memory violation problems. For example, a CCU can overwrite memory locations used by other (software or hardware) modules unintentionally.

V. Conclusions

In this paper we presented the Molen programming paradigm that addresses a number of previously unresolved issues such as parameter passing and parallel execution of operations into the reconfigurable hardware. As described the paradigm resolves the opcode space expansion, limitation of parameters passing and modularity. The proposal incorporates mechanisms for concurrent and parallel execution and it provides the user with almost arbitrary number of ”functions” to be implemented in a CCU. A number of examples have been reported in order to explain the presented programming paradigm.

References

[1] G. Blaauw and F. Brooks Jr., *Computer Architecture*. One Jacob Way: Addison-Wesley, 1997.

[2] 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.

[3] 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.

[4] 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.

[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] 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.

[8] 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.

[9] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, “Loading $\rho\mu$-code: Design Considerations,” in *International Workshop on Systems, Architecture, Modeling and Simulation (SAMOS)*, Samos, Greece, Jul 2003.

[10] 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)*, Lissabon, Portugal, Sep 2003.

[11] A. Peleg and U. Weiser, “MMX Technology Extension to the Intel Architecture,” *IEEE Micro*, vol. 16, no. 4, pp. 42–50, August 1996.

[12] A. Padegs, B. B. Moore, R. M. Smith, and W. Buchholz, “The IBM System/370 vector architecture: Design considerations,” *IEEE Transactions on Computers*, vol. 37, pp. 509–520, 1988.

[13] W. Buchholz, “The IBM System/370 vector architecture,” *IBM Systems Journal*, vol. 25, no. 1, pp. 51–62, 1986.