The MOLEN $\rho\mu$-coded processor

Stamatis Vassiliadis, Stephan Wong, and Sorin Coţofană

Computer Engineering Laboratory,
Electrical Engineering Department,
Faculty of Information Technology and Systems,
Delft University of Technology,
Delft, The Netherlands
{Stamatis, Stephan, Sorin}@CE.ET.TUDelft.NL

Abstract. In this paper, we introduce the MOLEN $\rho\mu$-coded processor which comprises hardwired and microcoded reconfigurable units. At the expense of three new instructions, the proposed mechanisms allow instructions, entire pieces of code, or their combination to execute in a reconfigurable manner. The reconfiguration of the hardware and the execution on the reconfigured hardware are performed by ρ-microcode (an extension of the classical microcode to allow reconfiguration capabilities). We include fixed and pageable microcode hardware features to extend the flexibility and improve the performance. The scheme allows partial reconfiguration and includes caching mechanisms for non-frequently used reconfiguration and execution microcode. Using simulations, we establish the performance potential of the proposed processor assuming the JPEG and MPEG-2 benchmarks, the ALTERA APEX20K boards for the implementation, and a hardwired superscalar processor. After implementation, cycle time estimations and normalization, our simulations indicate that the execution cycles of the superscalar machine can be reduced by 30% for the JPEG benchmark and by 32% for the MPEG-2 benchmark using the proposed processor organization.

1 Introduction

Reconfigurable hardware coexisting with a core processor can be considered as a good candidate for speeding up processor performance. Such an approach can be very promising. However, as indicated in [1], the organization of such a hybrid processor can be viewed mostly as an open topic. In most cases, the hybrid organization assumes the general-purpose paradigm. In such an organization, it is assumed that the processor operates in "ordinary processor environments" and is extended by reconfigurable unit(s) that speed-up the processing when possible. The execution and the reconfiguration are under the control of the "core" processor. Furthermore, due to the potential reprogrammability of the reconfigurable processor, a high flexibility is assumed in terms of programming resulting in tuning the reconfiguration for specific algorithms [2] or for the general-purpose paradigm [1].

In this paper, we introduce a machine organization where the hardware reconfiguration and the execution on the reconfigured hardware is performed by firmware via ρ-microcode (an extension of the classical microcode to include reconfiguration and execution for resident and non-resident microcode). Mechanisms within the extended microcode engine allow permanent and pageable reconfiguration and execution

ρ-microcode to coexist. Furthermore, we provide partial reconfiguration capability for "off-line" configurations and prefetching of configurations. Assuming the proposed machine organization, realistic implementations for reconfigurable operations, and cycle time computations regarding the implementation of the reconfigurable part, we show via simulations that: we are able to reduce the superscalar machine cycles by 30% for the JPEG encoding benchmark. Furthermore, the superscalar machine cycles were reduced by 32% for the MPEG-2 encoding benchmark.

The organization of the paper is as follows. Section 2 discusses in detail the new custom computing processor and the required architectural extensions. Furthermore, it also introduces a number of mechanisms to enhance the performance of the reconfigurable unit. Section 3 introduces the framework we have used to evaluate the performance and the expected performance of our proposal. Section 4 discusses some related work. Section 5 presents some concluding remarks.

2 General description

In its more general form, the proposed machine organization can be described as in Figure 1. In this organization, the I_BUFFER stores the instructions that are fetched from the memory. Subsequently, the ARBITER performs a partial decoding on these instructions in order to determine where they should be issued. Instructions that have been implemented in fixed hardware are issued to the core processing (CP) unit which further decodes them before sending them to their corresponding functional units. The needed data is fetched from the general-purpose registers (GPRs) and results are written back to the same GPRs. The control register (CR) stores other status information.

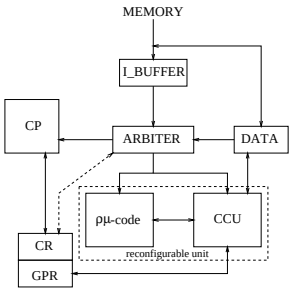

Fig. 1. *The proposed machine organization.*

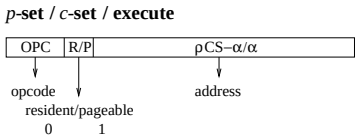

Fig. 2. *The* p-**set**, c-**set**, *and* **execute** *instruction formats.*

The reconfigurable unit consists of a custom configured unit (CCU)[1] and the $\rho\mu$-code unit. An operation[2] performed by the reconfigurable unit is divided into two distinct process phases: **set** and **execute**. The **set** phase is responsible for configuring the CCU enabling it to perform the required operation(s). Such a phase may be subdivided into two sub-phases: partial **set** (*p*-**set**) and complete **set** (*c*-**set**). The *p*-**set** sub-phase is envisioned to cover common functions of an application or set of applications. More

[1] Such a unit could be for example implemented by a Field-Programmable Gate Array (FPGA).

[2] An operation can be as simple as an instruction or as complex as a piece of code of a function.

specifically, in the *p*-**set** sub-phase the CCU is *partially* configured to perform these common functions. While the *p*-**set** sub-phase can be possibly performed during the loading of a program or even at chip fabrication time, the *c*-**set** sub-phase is performed during program execution. In the *c*-**set** sub-phase, the remaining part of the CCU (not covered in the *p*-**set** sub-phase) is configured to perform other less common functions and thus *completing* the functionality of the CCU. The configuration of the CCU is performed by executing reconfiguration microcode[3] (either loaded from memory or resident) in the $\rho\mu$-code unit. In the case that partial reconfigurability is not possible or not convenient, the *c*-**set** sub-phase can perform the entire configuration. The **execute** phase is responsible for actually performing the operation(s) on the (now) configured CCU by executing (possibly resident) execution microcode stored in the $\rho\mu$-code unit.

In relation to these three phases, we introduce three new instructions: *c*-**set**, *p*-**set**, and **execute**. Their instruction format is given in Figure 2. We must note that these instructions do *not* specifically specify an operation and then load the corresponding reconfiguration and execution microcode. Instead, the *p*-**set**, *c*-**set**, and **execute** instructions directly point to the (memory) location where the reconfiguration or execution microcode is stored. In this way, different operations are performed by loading different reconfiguration and execution microcodes. That is, instead of specifying new instructions for the operations (requiring instruction opcode space), we simply point to (memory) addresses. The location of the microcode is indicated by the resident/pageable-bit (R/P-bit) which implicitly determines the interpretation of the 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 by an *end_op* microinstruction.

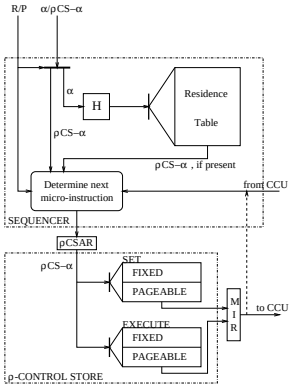

Fig. 3. *$\rho\mu$-code unit internal organization.*

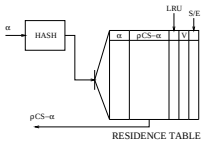

Fig. 4. *The sequencer's RESIDENCE TABLE.*

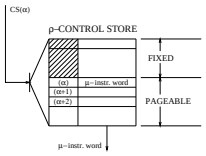

Fig. 5. *Internal organization of one section of the ρ-CONTROL STORE.*

The $\rho\mu$-code unit: The $\rho\mu$-code unit can be implemented in configurable hardware. Since this is only a performance issue and not a conceptual one, it is not considered further in detail. In this presentation, for simplicity, we assume that the $\rho\mu$-code unit is hardwired. The internal organization of the $\rho\mu$-code unit is given in Figure 3. In

[3] Reconfiguration microcode is generated by translating a reconfiguration file into microcode.

all phases, microcode is used to perform either reconfiguration of the CCU or control the execution on the CCU. Both types of microcode are conceptually the same and no distinction is made between them in the remainder of this section. The $\rho\mu$-code unit comprises two main parts: the SEQUENCER and the ρ-CONTROL STORE. The SEQUENCER mainly determines the microinstruction execution sequence and the ρ-CONTROL STORE is mainly used as a storage facility for microcodes. The execution of microcodes starts with the SEQUENCER receiving an address from the ARBITER 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 4) 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[4] support. In the cases that a ρCS-α is received or a valid translation into a ρCS-α is found, it is transferred to the 'determine next microinstruction'-block. This block determines which (next) microinstruction needs to be executed:

- When receiving address of first microinstruction: Depending on the R/P-bit, the correct ρCS-α is selected, i.e., from instruction field or from 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 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[5], 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 microcodes are stored in memory and the pageable part of the ρ-CONTROL STORE acts like a cache to provide temporal storage. Cache mechanisms are incorporated into the design to ensure the proper substitution and access of the microcode present in the ρ-CONTROL STORE. This is exactly what is provided by the RESIDENCE TABLE which 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 can be common for both the **set** and **execute** pageable ρ-CONTROL STORE sections or separate. In assuming a common table implementation, an additional bit needs to be added to determine which part of pageable ρ-CONTROL STORE is addressed (depicted as the S/E-bit in Figure 4).

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

[5] Both sections can be identical, but are probably only differing in microinstruction wordsizes.

3 Performance evaluation

To evaluate the proposed scheme, we have performed various experiments. The framework of our evaluations and assumptions are described below. We use as benchmark multimedia applications comprising: $ijpeg$ benchmark from the Independent JPEG Group (release 6b) and $mpeg2enc$ benchmark from the MPEG Software Simulation Group (v1.2). As data sets for the benchmarks we used: for $ijpeg$, we have four pictures taken from the SPEC95 $ijpeg$ benchmark: $testimg$ (277×149), $specmun$ (1024×688), $penguin$ (1024×739), and $vigo$ (1024×768). For $mpeg2enc$, we have taken the three frames set which is part of the benchmark. Furthermore, the well-known shortcut before the IDCT operation has been removed in order to abtain a fair comparison.

We have used the *sim-outorder* simulator from the SimpleScalar Toolset (v2.0)[3]. This base machine we use for comparison comprises: 4 integer ALUs, 1 integer MULT/DIV-unit, 4 FP adders, 1 FP MULT/DIV-unit, and 2 memory ports (R/W). We are interested in a realistic possibly worse case scenario, not just maximum potential performance. In order to achieve a realistic comparison: We have assumed the ALTERA APEX20K[6] chip and used the following software: FPGA Express from Synopsis (build 3.4.0.5211) and MAX+PLUS II (version 9.23 baseline). We have implemented the configurations of reconfigurable operations in VHDL and compiled it to the Altera technology directly or we implemented sub-units and mapped an operation on these sub-units. For all implementations we estimated: total area in terms of LUTs, number of cycles to perform an operation, and cycle clock frequency.

Furthermore, we assumed that the base superscalar machine can be clocked at 1 GHz. Given that cycles in a reconfigurable machine are slower than a hardwired machine, we determine the cycles required for the reconfigurable part using a normalization. The assumed normalization can be explained via an example. Assume that an reconfigurable instruction requires 2 cycles and that the estimated frequency is 200 MHz. Comparing the number of cycles alone is not enough to evaluate the scheme. We normalize to the superscalar cycles by multiplying the 2 cycles by 5 as the machine cycle of the reconfigurable units is 5 times slower than the hardwired counterpart. That is in our evaluation, the reconfigurable instruction takes 10 machine cycles rather than 2 cycles to complete. Finally, we assume that no partial reconfiguration is possible (i.e., no *p*-**set** phase). This is the worst time assumption for our scheme providing a low bound for our schemes performance gains. Further, we have assumed that there is no resident microcode which also is a worse case scenario for the proposed scheme.

By considering the application domain, we have decided to consider for reconfigurable code the following operations: SAD, 2D DCT, 2D IDCT, and VLC operations. These operation can be found in multimedia standards like JPEG and MPEG-2.

Area and speed estimates: In the first stage of estimating the overall performance of our approach, we have estimated the area requirements and possible clock speeds of the four multimedia operations within the multimedia standards JPEG and MPEG-

[6] We have to note that our FPGA technology assumption is a worse case scenario when compared to other more advanced FPGA structures with storage capabilities such as the Xilinx Virtex II family [4]. Therefore, it is expected that more improvements can be expected when our proposal is used in conjunction with such FPGA structures.

2. The results are presented in Table 1. Column one shows the operations performed. Column two presents the area that is required to implement the operation on an Altera APEX20K FPGA chip. Column three shows the number of clock cycles that the implemented operation needs to produce its results. The number in parentheses shows the maximum possible clock speed that was attainable as presented by the synthesis software. The normalized clock cycles are shown in the fourth column. By taking these normalized clock cycle numbers into our simulations, we can better estimate the performance increases when using our configurable unit inside a 1 GHz processor. We must note that the implementations used are not optimal and were only used to provide us with a good estimation of the number of clock cycles to use in the to be discussed simulations. Better implementations will yield higher performance increases. For the SAD operation, a complete VHDL model was written for the whole operation. For the DCT and IDCT cases, we did not write a complete VHDL model. Instead, we opted to estimate the area of the implementation based on the area and the performance of a 16x16 multiplier and a 32-bit adder and implemented the algorithm presented in [5].

operation	area	clock cycles (clock speed)	normalized cycles
16x16 SAD	1699 LUTs	39 (197 MHz)	234
16x16 multiply	1482 LUTs	12 (175 MHz)	69
32-bit adder	382 LUTs	5 (193 MHz)	21
DCT	using multiplier & adder		282
IDCT	using multiplier & adder		282
VLC	21408 RAM bits	variable	variable

Table 1. *Area and speed estimates.*

abbreviation	description
RLL (1000)	reconfiguration microcode load latency
RL (1000)	reconfiguration latency
MLL (100)	execution microcode load latency
ML	execution latency

Table 2. *Important latencies.*

Finally, the VLC case also requires special attention as also no VHDL model was written for it to estimate its area requirements. Instead, we obtained the higher bound on the area requirement by multiplying the total number of entries in the tables by the longest VLC code length. This resulted in an area requirement of 21408 RAM bits. Since the VLC operation greatly depends on the coarseness of the quantization step, it is not possible to use a single number of clock cycles that applies to all cases. Instead, we opted to simulate the VLC using a wide range of possible clock cycles in order to get an insight of the benefits of utilizing such an implementation in the CCU.

Simulation of the reconfigurable unit: Before we discuss the simulation results, we present the assumptions we have made prior to running the simulations. We have assumed that we extend a superscalar architecture with only two new instructions, namely the *c*-**set** and the **execute** instructions. Furthermore, we have assumed that we extend a superscalar processor with our reconfigurable unit to support the new instructions. Other assumptions are: the CCU can only contain one of the implementations as presented in the previous paragraphs. All microcode must be loaded at least once into the $\rho\mu$-code unit. Thus, we assume no microcode is present in the FIXED parts. The PAGEABLE part is only large enough to contain at most four microcode programs and we do not use the caching residence table. Furthermore, we have assumed a bandwidth of 32 bits per clockcycle between the processor and the caches. Finally, the loading of both the reconfiguration and execution microcode is handled by the existing memory units.

There are four latencies closely related to our processor that possibly affect the overall performance (see Table 2). In the discussion to follow, we fixed all the values

(using number shown in brackets) except the ML value. By varying the ML value, we tried to gain more insight into potential performance gains when faster implementations were used (i.e., lower ML values). Our implementation is indicated in bold.

Simulation results of $ijpeg$ benchmark: In Table 3, we show the results of the encoder within the $ijpeg$ benchmark which utilizes both the DCT and VLC implementations. We have varied the ML values between different values. The default case does not utilizes the new $\rho\mu$-code unit.

	$testimg$	$specmun$	$penguin$	$vigo$
default	6512947	129706215	142284352	149363055
DCT ML				
100	4525549 (-30.51%)	91279280 (-29.63%)	100581306 (-29.31%)	106461146 (-28.72%)
200	4610649 (-29.21%)	92930480 (-28.35%)	102373306 (-28.04%)	108304346 (-27.49%)
282	**4680431 (-28.14%)**	**94284464 (-27.31%)**	**103842746 (-27.02%)**	**109815770 (-26.48%)**
400	4780849 (-26.59%)	96232880 (-25.81%)	105957306 (-25.53%)	111990746 (-25.02%)
VLC ML				
100	6004054 (-7.81%)	120259786 (-7.28%)	131813270 (-7.36%)	138643470 (-7.18%)
200	6094054 (-6.43%)	121910986 (-6.01%)	133618070 (-6.09%)	140486670 (-5.94%)
400	6274054 (-3.67%)	125213386 (-3.46%)	137227670 (-3.55%)	144173070 (-3.47%)
DCT & VLC	**4505811 (-30.82%)**	**91167689 (-29.71%)**	**100294922 (-29.51%)**	**106157895 (-28.93%)**

Table 3. $ijpeg$ *encoder* cycle *results.*

Table 3 clearly shows that implementing the DCT and VLC operations in the CCU is able to reduce the total number of clock cycles to complete the $ijpeg$ encoder. For the DCT implementation, varying the ML-value between 100 cycles and 400 cycles is able to reduce the total number of clock cycles by between 30% and 25%. Assuming our implementation (ML = 282) for the DCT results in a reduction of about 27%. For the VLC implementation, the reduction is between 8% and 3%. Allowing the CCU to be reconfigured to either the DCT and VLC implementations shows a decrease of about 30%. In this case, we have assumed an ML-value of 200 for the VLC implementation. We can clearly see that the reconfiguration latencies is having an effect on the performance. This can be seen by adding the reduction of the only using the DCT and only the VLC implementations which is higher than 30%. Besides looking at the total number of clock cycles, we have also taken a look at some other metrics. In the cases, that the CCU can be configured to both the DCT and the VLC implementation, the following results were also obtained from the simulations: the total number instructions was reduced by 40.16%, the total number of branches was reduced by 28.55%, the total number of loads was reduced by 44.70%, and the total number of stores was reduced by 33.83%.

Simulation results of $mpeg2enc$ benchmark: In the simulations we have intentionally left out the results of both the IDCT and VLC implementations as they did not provide much performance increases. The main reason is that they only constitute a small part of all the operations in the MPEG-2 encoding scheme. From the result presented in the left table in Figure 6, we can see that the biggest contributor to the overall performance gain is the SAD which is able to decrease the number of cycles by about 35% (in a more ideal case). Using our implementation (ML = 234) results in a reduction of the number of clock cycles by 20%. Allowing the CCU to be reconfigured to either the DCT or the SAD implementation shows a decrease of 32%. Here we observe that the result is additive suggesting that the reconfiguration and execution of both operations do not influence each other. Besides looking at the metric the total number of

execution cycles, we have also taken a look at other metrics and they are also presented in the right two tables in Figure 6.

	# of cycles	difference (in %)
default	93245923	(0.00%)
DCT ML		
100	81350756	(-12.76%)
200	81465956	(-12.63%)
282	**81560420**	**(-12.53%)**
400	81696356	(-12.39%)
SAD ML		
100	59803641	(-35.86%)
200	70797383	(-24.07%)
234	**74608673**	**(-19.99%)**
400	93528041	(+0.30%)
DCT & SAD	**62928429**	**(-32.51%)**

total number of		difference
$instructions$		
default	137500555	0.0%
DCT	118757904	-13.63%
SAD	66016988	-51.99%
DCT & SAD	46291842	-66.33%
$branches$		
default	23233942	0.0%
DCT	21242800	-8.57%
SAD	10137590	-56.37%
DCT & SAD	8146448	-64.94%

total number of		difference
$loads$		
default	36191189	0.0%
DCT	33387057	-7.75%
SAD	22316745	-38.34%
DCT & SAD	19512613	-46.08%
$stores$		
default	2679153	0.0%
DCT	2450749	-8.53%
SAD	2679174	+0.00%
DCT & SAD	2450770	-8.52%

Fig. 6. $mpeg2enc$ cycle *results (left table) and other metrics (right table).*

The big decrease in the total number of loads in the case SAD is due to the fact that previously the data elements (which are 8 bits) were loaded one by one. However, in our CCU implementation we fully utilized the available memory bandwidth and loaded 4 elements at the same time and thus tremendously diminishing the number of loads. The number of stores remains the same as the intermediate results in the original could be stored in registers and therefore not requiring stores. We must note, that the computation of the SAD was dependent on two variables, namely the height and the distance (in bytes in memory) between vertically adjacent pels. Four resulting possibilities emerge and for each of these possibilities a different microcode was needed.

4 Related Work

In this paper, we introduce a machine organization where the reconfiguration of the hardware and the execution on the reconfigured hardware is done in firmware via ρ-microcode (an extension of the classical microcode to include reconfiguration and execution for resident and non-resident microcode). The microcode engine is extended with mechanisms that allow for permanent and pageable reconfiguration and execution code to coexist. We provide partial reconfiguration possibilities for "off-line" configurations and prefetching of configurations. Regarding related work, we have considered over 40 proposals. We report here a number of them that somehow use some partial or total reconfiguration prefetching. It should be noted that our scheme is rather different in principle from all related work as we use microcode, pageable/fixed local memory, hardware assists for pageable reconfiguration, partial reconfigurations, etc.. As it will be clear from the short description of the related work, we differentiated from them in one or more mechanisms. The *Programmable Reduced Instruction Set Computer (PRISC)* [6] attaches a Programmable Functional Unit (PFU) to the register file of a processor for application specific instructions. Reconfiguration is performed via exceptions. In an attempt to reduce FPGA reconfiguration overheads, Hauck proposed a slight modification to the PRISC architecture in [7]: an instruction is explicitly provided to the user that behaves like a NOP if the required circuit is already configured on the array, or is in the process of being configured. By inserting the configuration instruction before it is actually required, a so-called *configuration prefetching* procedure is initiated. At this point

the host processor is free to perform other computations, overlapping the reconfiguration of the PFU with other useful work. The *OneChip* introduced by Wittig and Chow [8] extends PRISC and allows PFU for implementing any combinational or sequential circuit, subject to its size and speed. The system proposed by Trimberger [9] consists of a host processor augmented with a PFU, *Reprogrammable Instruction Instruction Set Accelerator* (RISA), much like the PRISC mentioned above. *Garp* designed by Hauser and Wawrzynek [10] is another example of a MIPS derived Custom Computing Machine (CCM). The MIPS instruction set is augmented with several non-standard instructions dedicated to loading a new configuration, initiating the execution of the newly configured computing facilities, moving data between the array and the processor's own registers, saving/retriving the array states, branching on conditions provided by the array, etc. PRISM (*Processor Reconfiguration Through Instruction-Set Methamorphosis*) one of the earliest proposed CCM [11], was developed as a proof-of-concept system, in order to handle the loading of FPGA configurations, the compiler inserts library function calls into the program stream. From this description, we can conclude that explicit reconfiguration procedures are present. Gilson's [12] CCM architecture comprises a host processor and two or more FPGA-based *computing devices*. The host processor controls the reconfiguration of FPGAs by loading new configuration data through a Host Interface into the FPGA Configuration Memory. Schmit [13] proposes a partial run-time reconfiguration mechanism, called *pipeline reconfiguration* or *striping*, by which the FPGA is reconfigured at a granularity that corresponds to a pipeline stage of the application being implemented. The PipeRench coprocessor developed in Carnegie Mellon University [14] is focused on implementing linear (1-D) pipelines of arbitrary length. PipeRench is envisioned as a coprocessor in a general-purpose computer, and has direct access to the same memory space as the host processor. The *Reconfigurable Data Path Architecture* (rDPA) is also a self-steering autonomous reconfigurable architecture. It consists of a mesh of identical Data Path Units (DPU) [15]. The data-flow direction through the mesh is only from west and/or north to east and/or south and is also data-driven. A word entering rDPA contains a configuration bit which is used to distinguish the configuration information from data. Therefore, a word can specify either a SET or an EXECUTE instruction, the arguments of the instructions being the configuration information or data to be processed. A set of computing facilities can be configured on rDPA. The input/output data for these computing facilities can be written/read either by routing them through the mesh, or directly by means of a global I/O bus.

5 Conclusion

In this paper, we introduced the MOLEN $\rho\mu$-coded processor which contains a hardwired microcoded configurable unit. The proposed mechanisms allow, at the expenses of three new instructions to perform instructions, entire pieces of code or their combination to execute in a reconfigurable manner. The reconfiguration of the hardware and the execution on the reconfigured hardware are performed by ρ-microcode (an extension of the classical microcoded engines to allow reconfiguration capabilities). We include fixed and pageable microcode hardware features to extend the flexibility and improve

the performance of reconfiguration. The scheme allows partial reconfiguration, partial set of the hardware and caching mechanisms for non-frequently used reconfiguration and execution microcode. As future research directions, we envision: establish additional performance benefits when the caching microcode mechanisms are added to the simulation. Finally, consider more applications, e.g.,: Galois Field multiplications used in error correction for CDs and DVDs, a wide variety of table lookup methods primarily used in IP-routing algorithms, support for random switching patterns of sub-fields within register words, identify graphics functions that can benefit from our reconfiguration scheme and determine the possible potential benefits our mechanism can provide.

References

1. J. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," in *Proceedings of the IEEE Symposium of Field-Programmable Custom Computing Machines*, pp. 24–33, April 1997.
2. J. M. Rabaey, "Reconfigurable Computing: The Solution to Low Power Programmable DSP," in *Proceedings 1997 ICASSP Conference*, (Munich), April 1997.
3. D. C. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," Technical Report CS-TR-1997-1342, University of Wisconsin-Madison, 1997.
4. "Virtex-II 1.5V FPGA Family: Detailed Functional Description ." http://www.xilinx.com/partinfo/databook.htm.
5. C. Loeffler, A. Ligtenberg, and G. Moschytz, "Practical Fast 1-D DCT Algorithms With 11 Multiplications," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pp. 988–991, 1989.
6. R. Razdan, *PRISC: Programmable Reduced Instruction Set Computers*. PhD thesis, Harvard University, Cambridge, Massachusetts, May 1994.
7. S. A. Hauck, "Configuration Prefetch for Single Context Reconfigurable Coprocessors," in *6th Int. Symposium on Field Programmable Gate Arrays*, pp. 65–74, February 1998.
8. R. D. Wittig and P. Chow, "OneChip: An FPGA Processor With Reconfigurable Logic," in *IEEE Symposium on FPGAs for Custom Computing Machines* (K. L. Pocek and J. M. Arnold, eds.), (Napa Valley, California), pp. 126–135, April 1996.
9. S. M. Trimberger, "Reprogrammable Instruction Set Accelerator." U.S. Patent No. 5,737,631, April 1998.
10. J. R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," in *IEEE Symposium on FPGAs for Custom Computing Machines*, (Napa Valley, California), pp. 12–21, April 1997.
11. P. M. Athanas and H. F. Silverman, "Processor Reconfiguration through Instruction-Set Metamorphosis," *IEEE Computer*, vol. 26, pp. 11–18, March 1993.
12. K. L. Gilson, "Integrated Circuit Computing Device Comprising a Dynamically Configurable Gate Array Having a Reconfigurable Execution Means." WO Patent No. 94/14123, June 1994.
13. H. Schmit, "Incremental Reconfiguration for Pipelined Applications," in *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 47–55, April 1997.
14. S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "PipeRench: A Coprocessor for Streaming Multimedia Acceleration," in *The 26th International Symposium on Computer Architecture*, (Atlanta, Georgia), pp. 28–39, May 1999.
15. R. W. Hartenstein, R. Kress, and H. Reinig, "A New FPGA Architecture for Word-Oriented Datapaths," in *Proceeding of the 4th International Workshop on Field-Programmable Logic and Applications: Architectures, Synthesis and Applications.*, pp. 144–155, September 1994.