Reconfigurable Instruction Set Processors: A Survey

Francisco Barat and Rudy Lauwereins
Katholieke Universiteit Leuven.
{Francisco.Barat, Rudy.Lauwereins}@esat.kuleuven.ac.be

Abstract

Reconfigurable instruction set processors have the capability to adapt their instruction sets to the application being executed through a reconfiguration in their hardware. Through this adaptation, they are expected to achieve a great improvement in performance compared to fixed instruction set processors. In this paper, we discuss the different hardware aspects that have to be considered during the design of such a reconfigurable processor. The topics discussed include the coupling of the processor and the reconfigurable logic, configuration, instruction coding and scheduling, granularity, hardware cache and reconfigurability. A classification of current reconfigurable processors is done according to the discussed topics.

1 Introduction

Reconfigurable processors are a new type of processor that combines a microprocessor with reconfigurable logic. The objective of this approach is to add the benefits from microprocessors and reconfigurable logic.

The reconfigurable logic will provide hardware specialization to the application being executed. It will provide similar benefits to those offered by application specific instruction set processors (ASIP). ASIPs have specialized hardware that accelerate the execution of the applications it was designed for. A reconfigurable processor would have this same benefit but without having to commit the hardware into silicon. Reconfigurable processors can be adapted after design, in the same way that programmable processors can adapt to application changes.

Reconfigurable instruction set processors (RISP) are a subset of reconfigurable processors. They will be the focus of this paper.

Many systems have been designed with very different characteristics. The main objective of this paper is to present and classify the design decisions that have to be taken to create such kind of processors.

Research on this type of processors has been quite active in the last years. The essential enabler of this has been the fast growth of FPGA capacity. In Table 1, we can see a sample of reconfigurable processors. In the table, we present the characteristics of these processors according to the issues studied in this paper. Characteristics that do not apply to a specific processor are left blank.

We have included in this table those processors that we believe are the most significant to RISP evolution. There are many more processors, especially from before 1994. Most of these processors are of the attached processor type (explained in section 2.1) which is not the focus of this paper.

This paper is divided in four main sections. In section 2, the integration of the reconfigurable processing unit (RPU) with the processor is discussed. Section 3 describes the characteristics of the reconfigurable logic. Finally, conclusions are presented in section 4.

2 Processor integration

The design of a reconfigurable processor can be divided in two main tasks. The first one is the interfacing between the microprocessor and the reconfigurable logic. This includes all the issues related to how data is transferred to and from the reconfigurable logic, as well as synchronization between the two elements. The second task is the design of the reconfigurable logic itself. Granularity, reconfigurability, interconnection are issues included in this task.

In this section, we discuss several issues related to the interfacing of the processor with the reconfigurable logic, and the operation of reconfigurable processors.

2.1 RPU coupling

The position of the RPU relative to the microprocessor affects performance. The benefit obtained from executing a piece of code in the RPU depends on communication and execution costs [10]. The time needed to execute an operation in the RPU is the sum of the time needed to transfer the processed data and the time required to process it. If this total time is smaller than the time it would normally take in the processor alone, then an improvement can be obtained.

The RPU can be placed in three main places relative to the processor [13]:

- Attached processor: The reconfigurable logic is placed on some kind of I/O bus (e.g. PCI bus). Example: PRISM-1 [6].

0-7695-0668-2/00 $10.00 © 2000 IEEE

- Coprocessor: The logic is placed next to the processor. The communication is done using a protocol similar to those used for floating point coprocessors. Example: Garp [3]
- Functional unit: The logic is placed inside the processor. The instruction decoder issues instructions to the reconfigurable unit as if it were one of the standard functional units of the processor. Example: OneChip98 [11].

With the first two interconnection schemes, sometimes called loosely coupled, the speed improvement using the reconfigurable logic has to compensate for the great overhead of transferring the data. This usually happens in applications where a huge amount of data has to be processed using a simple algorithm that fits in the RPU. Most systems built until recently were of this kind. Their main benefit is the ease of constructing such a system using a standard processor and standard reconfigurable logic. Another benefit of this approach is that the microprocessor and RPU can work in different tasks at the same time.

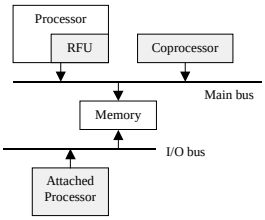

Figure 1 Basic types of RPU coupling

With the coprocessor scheme, sometimes called tightly coupled, the communication costs are practically none and because of this, it is easier to obtain a speed increase. This configuration seems the most promising one because it can be used to accelerate almost any application. For any given application, it is usually possible to find a set of instructions that can be used to increase the processor's performance.

From now on, we will focus on processors with a RPU as a functional unit. We will call this unit, a reconfigurable functional unit (RFU). A reconfigurable processor can have one or more RFUs. As a normal functional unit, an RFU executes instructions that come from the standard instruction flow. A typical RISP may look like the one in Figure 2.

When reconfigurable logic is placed inside the processor, it does not have to be placed necessarily inside a functional unit. Other places of interest can be the external interface, the decoding logic or the control logic. By placing the reconfigurable logic next to the I/O pins, it is possible to build custom interfaces to other elements in the system, thus eliminating the need of external glue logic.

The first reconfigurable processors where of the attached processor or co-processor type, but as gate capacity is increasing, more and more processors are being designed with RFU approach. This is due to the fact that the expected performance is higher for the tightly coupled configuration. Nevertheless, attached processor systems still have an important role in some applications such as stream processing.

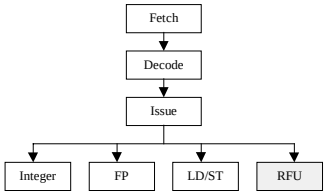

Figure 2 RISP architecture

2.2 Instruction types

The design of the interface to the reconfigurable unit will depend on the characteristics of the instruction types that we want to implement. Two main types of instructions can be implemented on a RFU [13]:

- Stream based instructions (or block based instructions): they process large amounts of data in a sequential or blocked manner. Only a small set of applications can benefit from this type. Most of them are suitable for a coprocessor approach. Examples: FIR filtering, discrete cosine transform (DCT)…
- Custom instructions: these instructions take small amounts of data at a time (usually from internal registers) and produce another small amount of data. These instructions can be used in almost all applications as they impose less restrictions on the characteristics of the application. Example: bit reversal, multiply accumulate (MAC), variable length coding (VLC) and decoding (VLD)…

Instructions can also be classified in many other manners, such as execution time, pipelining, internal state, etc. If the type of the reconfigurable instructions closely resembles the fixed instructions supported by the microprocessor, the integration process will be easier.

The type of instructions supported is closely related to the applications that are expected to be executed on the processor. For example, multimedia applications usually require stream processing. This fact can be checked in Table 1.

2.3 Instruction coding and register access

Reconfigurable instructions are usually identified by a special opcode. Which reconfigurable instruction is executed is specified using an extra field in the instruction word. This extra field can specify:

- Address: The address of the configuration data for the instruction is specified in the instruction word. Example: DISC [5].

- Instruction number: An instruction identifier of small length is embedded in the instruction word. This identifier indexes a configuration table where information, such as the configuration data address, is stored. The number of reconfigurable instructions at one time is limited by the size of the table. Example: OneChip98 [11].

The first approach needs more instruction word bits but has the benefit that the number of different instructions is not limited by the size of a table, as in the second case. If the configuration table can be changed on the fly, the processor can adapt to the task at hand on runtime. The major drawback of this approach is that specialized scheduling techniques have to be used during code generation. The compiler will have to schedule during the lifetime of the program what instructions are stored in the configuration table.

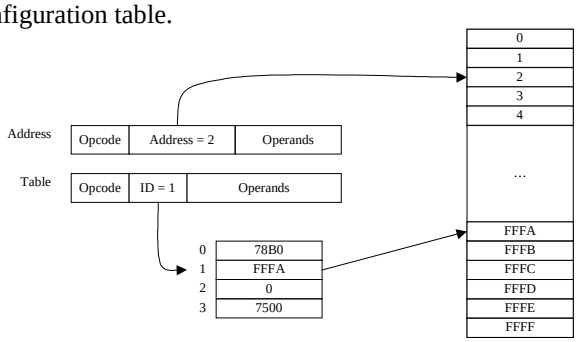

Figure 3 Fixed instruction selection formats

The instruction word also specifies the operands to be passed to the RFU. The operands passed to the RFU can be immediate values, addresses, register ids, etc. The operands specified can be source or destination of the operation. The operands can be coded in several ways:

- Hardwired: The contents of all registers are sent to the RFU. The registers actually used depend on the instruction configured inside the RFU. This allows the RFU to access a bigger amount of registers but makes code generation more difficult. The actual selection of which registers are used is done inside the RFU. This is the approach taken in Chimaera [9].
- Fixed: The operands are in fixed positions in the instruction word and are of fixed types. If there are different opcodes for reconfigurable instructions, they could have different encoding formats. Example: OneChip98 [11].
- Flexible: The position of the operands is configurable. The degree of configuration can be very broad. If a configuration table is used, it can be used to specify the decoding of the operands. Example: DISC [5].

In Figure 4, we can see hardwired coding for the source operands and fixed coding for the destination operand. The contents of all the registers are transferred to the RFU through special lines. The result from the RFU is routed to the register indicated in the instruction word. Hardwired coding allows a bigger number of registers to be accessed without a major complication in the hardware design.

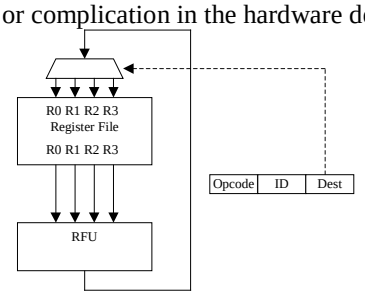

Figure 4 Hardwired coding

Fixed coding for both source and destination operands can be seen in Figure 5. In this case, there are only two source operands, which are selected from the register file. This method needs more instruction bits but allows a greater flexibility to specify operands. It is the method most commonly used, as can be seen in Table 1.

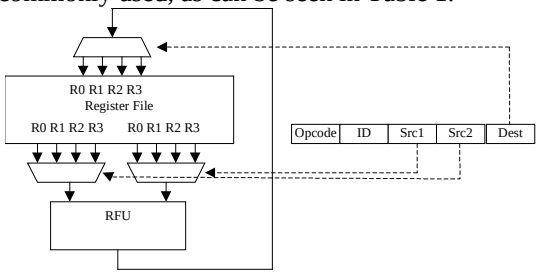

Figure 5 Fixed coding

Finally, in Figure 6, flexible coding for source operands is shown. In this case, one operand is selected from the register file and the other operand is also selected from the register file or is treated as a constant included in the instruction word. The way in which the second operand is interpreted depends on the configuration table. This configuration table has the same structure as the one used to identify the instruction. None of the existing RISP use this kind of operand coding.

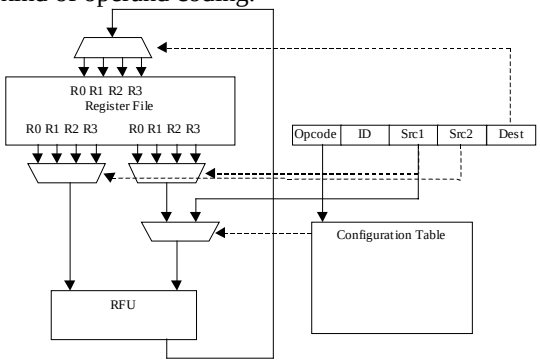

Figure 6 Flexible coding

The register file accessed by the RFU can be shared with other functional units (such as the integer pipeline) or be dedicated (such as the floating point register file in some architectures). The dedicated register file would need

less ports than if it was shared. This will simplify its design. A major drawback of separate files is register heterogeneity.

All current reconfigurable processors use the same register file for fixed and reconfigurable instructions. This will most likely change when more research is done on reconfigurable superscalar and VLIW processors. Splitting the register file allows the design of bigger and faster register files by reducing the number of ports.

2.4 Memory access

Reconfigurable instructions access memory through memory ports. With these ports, the instructions can make specialized load/store operations or implement stream-based operations. If the memory hierarchy supports several accesses at the same time, then the number of memory ports can be greater than one.

2.5 Instruction scheduling

RFUs can be included in any of the types of processors available today. Due to instruction issuing, the type of processor affects the design of the functional unit.

In standard RISC and CISC processors, where instruction level parallelism is not exploited, adding a RFU is quite straightforward. The common way is to wait for the RFU to issue a "done" signal, [4] before executing the next instruciton.

RFU designs for VLIW processors would normally involve fixed duration instructions. Unknown duration on VLIW would result in pipeline stalls, with a major loss in performance. No VLIW processors currently exist that include reconfigurable hardware.

Variable length instructions can be dealt with quite efficiently on super-scalar processors. The RFU can be used as one of the standard RFUs through reservations stations or some other mechanism.

If we only have one pool of reconfigurable logic but several connection interfaces, then the pool can work as if there were several RFUs. In this manner, reconfigurable logic is shared among the RFUs. This technique can be used in super-scalars and VLIWs.

Most RISPs are only able to execute one instruction at the same time. They are based on both CISC and RISC designs. The only superscalar processor that we know of is the OneChip98, which is based on the superscalar version of DLX.

3 Reconfigurable logic design

In this section, we focus on the design of the reconfigurable logic itself. The design of the reconfigurable logic will determine, amongst other things, how many instructions fit inside it, the size of the configuration stream, and the type of instructions that will give the most performance. In Figure 7 we see a typical reconfigurable functional unit.

3.1 Granularity

A very important aspect of the RFU design is the granularity of the reconfigurable resources. The building blocks for fine-grained logic are gates, (efficient for bit manipulation operations), while in coarse-grained RFUs the blocks are bigger (therefore better suited for bit parallel operations).

Granularity also affects the size of the configuration stream and the configuration time. With fine-grained logic, more information is needed to describe the instruction. Coarse-grained logic descriptions are more compact, but when the size of the application data does not match that of the logic, the performance decreases due to unused power.

The optimal granularity varies from design to design. For bit-oriented algorithms, a fine-grained approach seems the best choice, while for computation intensive applications, the coarse-grain approach can be a better solution.

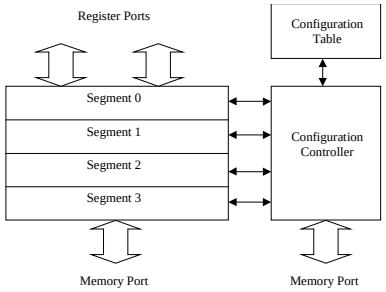

Figure 7 Structure of a RFU

Most of the early designs included standard fine-grained FPGA resources (e.g. DISC [5], OneChip [10]). In several of the most modern approaches, the RFU is implemented with non-standard cell structures (e.g. Garp [3], Chimaera [9]). They are still fine-grained, except in the case of PipeRench. Coarse grained RFU seem the most likely to achieve high performance with small penalties for reconfiguration times, power consumption and delay.

Related to the granularity is the segment size. A segment is the minimum hardware unit that can be configured and assigned to an instruction. Segments allow instructions to share the reconfigurable resources.

If segments are used, the configuration of an instruction is done in a hierarchical manner. The instruction is assigned a set of segments, and inside those segments, the processing elements are configured.

3.2 Interconnect

Interconnect is used to connect the processing elements (i.e. gates or segments) to obtain the desired functionality. The interconnect that connects the elements inside a

segment is referred to as intra-segment interconnect. Inter-segment interconnect is used to connect different segments.

Intra-segment interconnect is related to granularity. It also affects the speed obtained. In typical FPGAs, there are different levels of intra-segment interconnect. With coarse grained architectures, the interconnect tends to be done using buses and crossbar switches.

Inter-segment interconnect only appears in RFU that support multiple segment instructions. It is used to transmit data between the different segments. There are several kinds of inter-segment interconnect:

- Fixed: The position of the segments of an instruction inside the RFU is fixed at design time.
- Relative: The position of the instruction's segments is specified relative to each other. This gives a little slack to position the instruction inside the RFU.
- Relocatable: The position of the segments is not fixed at all. They can be placed anywhere on the RFU

In Figure 8 we can see different configuration options for an instruction of three segments and an RFU of four segments. With fixed coding, there is only one position where the instruction can be placed. With relative interconnect, there are two possibilities. Finally, with relocatable interconnect, there are many possible configurations (only three are presented).

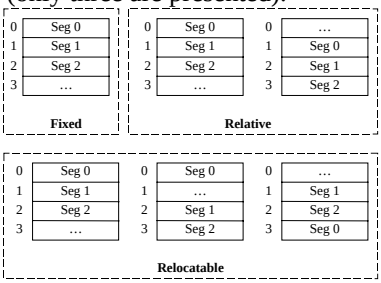

Figure 8 Inter-segment interconnection

The type of inter-segment interconnect determines the complexity and size of the interconnect, and the size of the configuration stream used for interconnect description. Fixed interconnect is the simplest one and requires the least configuration bits. Relative interconnect, is very similar to fixed interconnect, in the sense that no extra logic is required and that the size of the configuration stream is very similar. Relocatable interconnect is the most expensive one.

The design has to be resilient to misconfigurations. In case the logic is misconfigured, it should not be destroyed. This imposes certain restrictions on the design, but will ensure a longer life of the system. The method normally used consists in allowing only one current driver per interconnect [9].

3.3 Reconfigurability

The reconfigurable logic inside the RFU can be configured at different moments. If the RFU can only be configured at startup, we say that the unit is not reconfigurable (it is configurable). The reconfigurable logic would get its configuration from some external source (e.g. ROM) in a manner similar to conventional FPGAs. In this type of RFU (e.g. Nano Processor [4]), the total number of special instructions depends on the size of the reconfigurable logic. With this approach, we have an ASIP that can be customized after being committed to silicon.

If we can configure the RFU after initialization, the instruction set can be bigger than the size allowed by the reconfigurable logic. If we divide the application in functionally different blocks, the RFU can be reconfigured to the needs of each individual block. In this manner, the instruction adaptation is done in a per block basis. Most of the reconfigurable processors belong to this kind.

3.4 Configuration process

Reconfiguration times depend on the size of the configuration data, which can be quite large. These times depend on the configuration method used. In the PRISC processor [6], the RFU is configured by copying the configuration data directly into the configuration memory using normal load/store operations. If this task is performed by a configuration unit that is able to fetch the configuration data while the processor is executing code, a performance gain can be obtained. We will focus in this more common approach.

The configuration controller in Figure 7 is in charge of reconfiguring the logic. It has a memory port to read the new configuration data. As configuration data is normally contiguous in memory, the configuration process can access external memory using fast access modes designed for high throughput.

The reconfigurable logic is simpler if the RFU blocked during reconfiguration. If the RFU can be used while reconfiguring, it is possible to increase performance. Usually, there are several configuration planes, of which only one is active at a given time (a configuration plane stores the configuration of the RFU). New configuration data can be loaded to the other planes. Changing the active plane is a very fast operation (usually one clock cycle).

If the RFU is divided in segments that can be configured independently from each other, we do not have to reconfigure the whole RFU at a time, thus reducing configuration time.

Since configuring the RFU takes some time, prefetching the instruction configuration data can reduce the time the processor is stalled waiting for reconfiguration. The insertion of prefetching instructions should be done automatically by software tools.

3.5 Caching the hardware and relocatability

If all the segments are alike and interconnection between them is relocatable, a cache of instructions can be

implemented. In [5], each time a reconfigurable instruction is executed, a configuration table is used to see if the instruction is configured in the RFU. If the instruction is already configured, then it is executed. If it is not configured, its configuration data is loaded automatically, replacing some segments of the reconfigurable fabric (usually, the least recently used).

Hardware caching can be used even if the RFU is not relocatable. If the RFU has several configuration planes, one plane can be assigned to one instruction. The caching then works with planes instead of segments.

4 Conclusions and future work

In this paper, we have presented a broad picture of reconfigurable instruction set processors. An analysis of the main hardware issues has been done. The two main aspects that have to be studied are the interfacing of the reconfigurable logic with the microprocessor and the design of the reconfigurable logic itself. They both involve many decisions, which we have tried to enumerate as thoroughly as possible.

Future work will continue along these main lines. Both the hardware and the software side have to be studied further. Several experiments will have to be done in order to determine which is the best RISP architecture for the MPEG-4 domain. Similarly, during the selection of the architecture, tests will have to be done to determine the best way to compile code.

Acknowledgements

This work is supported in part by IWT project Irmut, FWO project G.0036.99 and IMEC.

References

[1] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Palo Alto, CA, 1996.

[2] T.G. Rauscher and A.K. Agrawala, *Dynamic Problem-Oriented Redefinition of Computer Architecture via Microprogramming*, IEEE Trans. Computers, Vol. C-27, No. 11, Nov. 1978, pp. 1,005-1,014.

[3] J.R. Hauser, and J. Wawrzynek, *Garp: A MIPS Processor with a Reconfigurable Coprocessor*, Proc. IEEE Symp. FCCM, April 1997, pp.12-21.

[4] M.J. Wirthlin, B.L. Hutchings, and K. L. Gilson, *The Nano Processor: A Low Resource Reconfigurable Processor*, Proc. IEEE Workshop on FCCM, 1994, pp. 23-30.

[5] M.J. Wirthlin, and B.L. Hutchings, *A Dynamic Instruction Set Computer*, Proc. IEEE Symp. FCCM, 1995, pp. 99-107.

[6] P.M. Athanas, and H.F. Silverman, *Processor reconfiguration through instruction-set metamorphosis*, IEEE Computer, March 1993, pp. 11-18.

[7] M. Wazlowski et al, *PRISM-II compiler and architecture*, Proc. IEEE Workshop FCCM, 1993, pp. 9-16.

[8] R. Razdan, and M.D. Smith, *A High-Performance Microarchitecture With Hardware-Programmable Functional Units*, Proc. 27th Int'l Symp. Microarchitecture, 1994, pp. 172-180.

[9] S. Hauck, et al, *The Chimaera Reconfigurable Functional Unit*, Proc. 5th IEEE Symp. FCCM, 1997, pp. 87-96.

[10] R.D. Wittig, and P. Chow, *OneChip: an FPGA processor with reconfigurable logic*, Proc. IEEE Symp. FCCM, 1996, pp. 126-135.

[11] J.A. Jacob, and P. Chow, *Memory interfacing and instruction specification for reconfigurable processors*, Proc. ACM/SIGDA Int'l Symp. FPGAs, 1999, pp. 145-154.

[12] C.R. Rupp, et al, *The NAPA adaptive processing architecture*, Proc. IEEE Symp. FCCM, 1998, pp. 28-37.

[13] S.C. Goldstein, et al, *PipeRench: a Coprocessor for Streaming Multimedia Acceleration*, Proc. Int'l Symp. Computer Architecture, 1999, pp. 28-39.

Table 1 Characteristics of several reconfigurable processors

	PRISM-I [6]	PRISM-II [7]	Nano Processor [4]	PRISC [8]	DISC [5]	OneChip [10]	Garp [3]	Chimaera [9]	NAPA [12]	OneChip98 [11]	PipeRench [13]
Year	1993	1993	1994	1994	1995	1996	1997	1997	1998	1999	1999
Processor Type	M68010	Am29050	RISC?	R2000	CISC?	DLX	MIPS	?	RISC	S-DLX	?
Application Type	GP	GP	GP	GP	GP	GP	GP	Multimedia	GP	GP	Multimedia
Coupling	Attach	Copro	RFU	RFU	RFU	RFU	Copro	RFU	Copro	RFU	Attach
Inst. Types	All	All	Custom	Custom	All	All		Custom		Stream	Stream
Duration	Var.	Var.	Fixed	Fixed	Var.	Var.	Var.	Fixed	Var.	Var.	Var.
Inst. Coding			Fixed	Fixed	Fixed	Fixed	Fixed	Fixed		Fixed	
Configuration Table			No	No	Yes	No	Yes	Yes		Yes	
Operand Coding			Fixed	Fixed	Wired	Fixed	Fixed	Wired		Fixed	
Shared Register File			Yes	Yes	Yes	Yes	Yes	Yes		Yes	
Memory Ports				No	Yes	Yes	Yes	No	Yes (2)	Yes (1)	Yes
Granularity	Fine	Fine	Fine	Fine	Fine	Fine	Fine	Fine	Fine	Fine	Coarse
Reconfigurable	No	No	No	Yes	Partial	No	Yes	Yes	Yes	Yes	Yes
Configuration Method				SW	SW		SW	Ctlr.	Ctlr.	Ctlr.	
RFU Blocked during Reconfiguration				Yes	Yes		Yes	No		No	No
RPU Segmented				No	Yes		Yes	Yes	Yes	No	Yes
Can Support Prefetching?				No	No		No	?		Yes	
Relocatable Hardware				No	Yes		Yes	Yes	Yes	No	Yes
Instruction Caching				No	Yes		Yes	Yes		Yes	