Reconfigurable Computing

Project Report

Authors: Georgi Todorov / Reto Galli

Oregon State University
Department of Electrical and Computer Engineering
ECE 570 – Advanced Computer Architecture
Winter 2000
Prof. Bob Colwell

Contents

1 Introduction

Our motivation for this research is to get an insight in architectural approaches other than the ones we have seen so far in our studies. Reconfigurable Computing has been recently researched a lot and many believe it is a promising field.

Our approach was to find the newest publication and try to synthesize the great amount of information about this topic. We have read about twenty-five papers and scanned many more during our quest for knowledge. It is worth noticing that the reconfigurable field of computing is not particularly coherent and there are very few papers with throughout overviews.

The structure of the paper is as follows: in part one we show several architectural approaches for reconfigurable computing. Among those are: a co-processor design, both connected via a bus and on-chip designs, a URISC single-instruction architecture and a reconfigurable ALU architecture. Part two is dedicated to the issues of run-time management and software-hardware co-design environment. In part three we try to give some insight in the reconfigurable logic performance. We give examples of various problems that the current state-of-the-art faces. Part four shows some of the applications using reconfigurable logic for performance speed-up. There can be more possible researched applications however the presented ones shall give the reader the feeling what they may be.

2 Different Architectural Approaches for Reconfigurable Computing

In our literature review about reconfigurable computing we have found different architectural approaches. These approaches can be classified into four groups. The oldest and best-developed approach is a reconfigurable logic (RL) that works as a co-processor and is connected to the main processor over the bus [BhSiKa99]. Another approach is to have the main processor and the RL are on the same chip. The RL still acts as a co-processor [GrNe99]. Newer research shows other approaches, which are not (or only as prototype) realized so far. In these approaches there is just one processor with reconfigurable parts. Among those we found a solution with a flexible URISC (Ultimate RISC) core and RL accessed like a memory [Do98] and others with reconfigurable ALUs (RALU) in the pipeline [DA99, SaGrSp98]. We will now describe these four approaches more closely.

2.1 Reconfigurable Co-Processor Connected over the Bus

There exist several systems based on this approach. Most of them are quite autonomous systems with their own memory and are connected to the main system over one of the standard bus systems as, for example, the PCI bus in a PC. The NEBULA system from University of Cincinnati [BhSiKa99] shown in Figure 2-1 is an example for such a system.

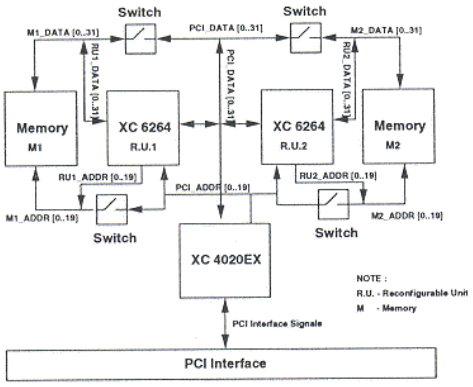

Figure 2-1: NEBULA Architecture

Because those RLs have different worst path delays depending on the actual configuration they have variable clock generators that change their clock rate according to the configuration. The system mentioned above has a variable clock between 139kHz and 100MHz.

A problem of those systems is that they are connected to the main system over the slow system bus (PCI $\rightarrow$ 33MHz). Therefore a lot of speedup is needed in the algorithm to compensate the latency from this slow connection. That's why these systems are only recommended for special applications as image processing where a lot of computations with a lot of parallelism exist.

2.2 Reconfigurable Co-Processor on Chip

One of the main disadvantages of the co-processor connected over the bus is that the communication between the main processor is slowed down slightly by the bus. Furthermore the co-processor is not able to access the onboard cache of the main processor. Therefore all data must be written back to main memory before the co-processor can use it. A huge speedup is needed in the algorithms to compensate those tradeoffs. That's why some researches put the main processor and the reconfigurable co-processor on the same die. The internal bus on the chip is much faster than the external one and therefore one can get rid of the worst bottleneck [GrNe99]. Furthermore the co-processor can share several I/O resources and the cache with the main processor.

One of the problems with this approach is that the realization of a variable clock depending on the actual configuration is much more difficult. DSPs that run at slightly lower clock rates (100-200MHz) than RISC or CISC processors are therefore more suitable for this approach. We will show in section 5.1 that DSPs have a lot of potential for RL.

In Figure 2-2 [GrNe99] is the block diagram of a DSP with a reconfigurable FU / co-processor shown. This architecture actually exists as a prototype and shows an average speedup of six over a large set of DSP benchmarks. For some killer-apps it even provides a speedup up to 34.

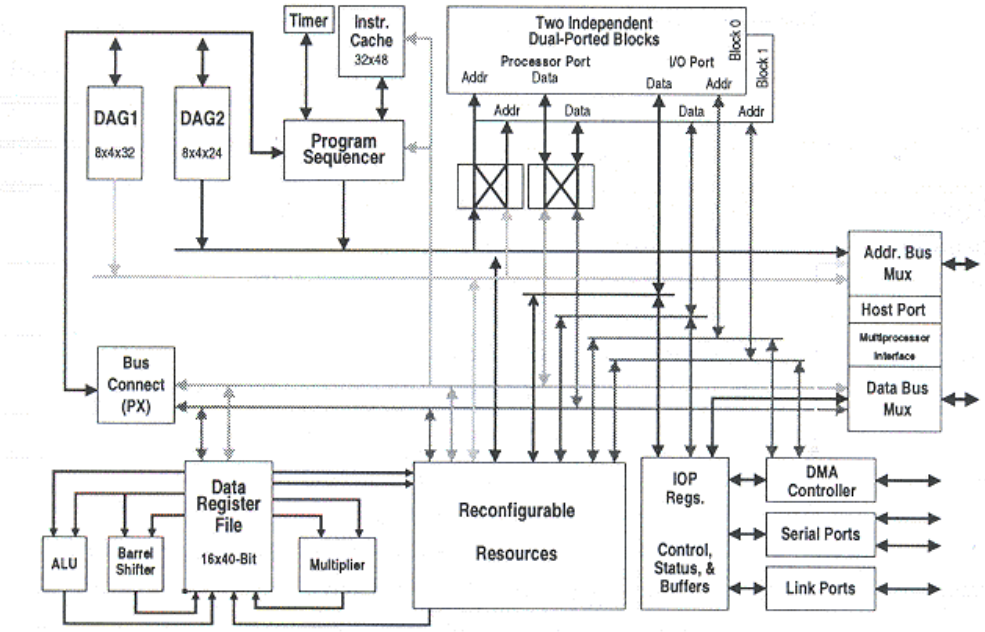

Figure 2-2: DSP with Reconfigurable FU/Coprocessor Combination

2.3 Flexible URISC – Flexible Ultimate (single instruction) RISC

An ultimate RISC system is a minimal processor architecture with only one instruction: *move memory to memory*. A complete description of URISC can be found in [Jo88]. An approach for a reconfigurable processor based on this technique is described in [Do98]. It works as follows:

There exist FPGAs with a so-called FastMap[tm1] interface. This means that the FPGA can be programmed in exactly the same way as you would write to memory. The URISC core moves first the configuration code to the FPGA that is mapped into the memory space. Afterwards it moves the parameters to the FPGA and when this finishes the calculation the core can get the results and move them to another functional unit or the RAM. The URISC core and the FPGAs can easily be placed on the same chip, e.g. the URISC core can be implemented in the FPGA. As a self-modifying system, the Flexible URISC has direct, unmediated access to the host FPGA's FastMap programming interface. Program and circuit data, stored in on-board SRAM, is directly accessible over the on-chip memory bus. This allows a high bandwidth memory transfer and a fast reconfiguration. Tests in [Do98] showed a bandwidth of up to 8Mb/s.

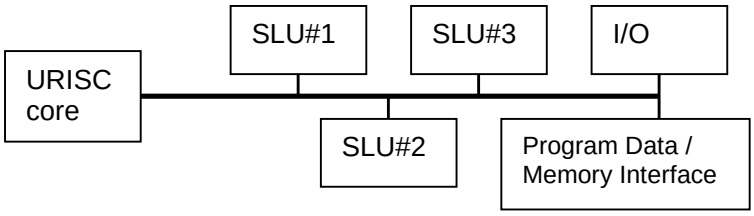

Figure 2-3: Blockdiagram of the flexible URISC system

A disadvantage of this approach is that it requires additional compiler complexity to derive a suitable dataflow.

2.4 Reconfigurable ALU (RALU)

The motivation for this approach is to create a standard processor with a dynamic instruction set. This means in addition to the standard set of instruction each application could add some application specific instructions, which could be used in the program as every other instruction. A multimedia program could for example add an extension like MMX[tm 2] dynamically. A data security application could load a cryptographic extension. Because this functions would fit into the instruction set as every other instruction it shouldn't be a problem for the compiler to handle hardware like that.

There exists some prototype hardware for this approach. One is the CoMPARE system at the University of Technology, Dresden [SaGrSp98]. Their ALU has an additional array of 100,000 transistors. They claim a speedup of 2 to 4 for most applications. Unfortunately they don't specify what test applications they used. To reconfigure their hardware completely they need 1984 clock cycles. However they are able to reconfigure partially. Also reconfiguration should be an infrequent operation, probably taking place just once at the start of an application. The actual prototype runs only at a clock speed of 11MHz. They are working on a faster one.

We have found another project with this approach in [DA99], which describes a project at University of Glasgow (UK) that started in October 1999 and is still in progress. We tried to contact the researches to find out what the newest results of the project are and to ask them some critical questions about the project, but unfortunately we got no answer. So we have to concentrate on the information we have from this single paper.

[1] FastMap[tm] is a trademark of Xilinx,inc and is available in Xilinx XC62000 series.
[2] MMX[tm] is a trademark of Intel and is available in chips of the P6 serie.

The idea of this approach is to take an existing CPU architecture and add an additional reconfigurable ALU (or functional unit) (RALU) into the pipeline. This will result in a CPU with a dynamic instruction set. Each program is able to define a set of custom instructions. A table with the opcodes and the location of the reconfiguration code for this instruction has to be saved in the CPU. When an opcode of such an instruction occurs in the decode stage the according reconfiguration code is loaded, the RALU will be programmed and the instruction will be executed as every other instruction of the static instruction set.

This idea sounds very nice, but when you look closer you realize a lot of problems in the realization of this approach. When one of these dynamic opcodes is detected the RALU has to be reconfigured. This means another memory access to get the reconfiguration code. Even when an additional fast cache for this data is added, it won't be possible to load that data and reconfigure the ALU without delaying the instruction. This would mean, the pipeline had to be stalled or the RALU had to be inserted into an out-of-order superscalar machine where you could delay the instruction in a reservation station.

Unfortunately they give no information what their expectations are about how this RALU will affect the clock speed. It will slow down the clock for sure. Other reconfigurable architectures work with a variable clock. Although this could be possible here too, it would complicate the architecture a lot.

How the operating system and the compilers can handle such a CPU is another issue that must be explored.

This is for sure an interesting approach but a lot more research has to be done before the first CPU like this one will be produced.

3 Dynamic versus Static Configurations

In our research we have found that the hardware reconfigurablity can be more or less dynamic. As many researchers point out, there is a trade-off between performance and flexibility. The role of the compiler can be different leading to more or less complex compilers.

In a static environment the configurations are known at compile time. The transitions between subsequent configurations are also known and it is the job of the compiler to optimize performance. There is no or little need of run- time decisions since the schedule is static.

A dynamic environment implies decisions made at run-time. The computation phases are alternated with reconfiguration phases. Therefore the processing must be halt for the time of reconfiguration. This approach has the flexibility to handle a wider range of applications while spending execution time for reconfiguration.

Our sense is that the dynamically reconfigurable computing is a subject of greater interest. In [Do98] we find a description of the dynamic environment of the Flexible URISC core.

In another work [M^{c}GrLy99] the authors point out the necessity of having dynamically reconfigurable devices with partially reconfigurable sub-section of the FPGA. It is shown that some applications cannot execute on devices that are not dynamically reconfigurable.

We find that the question of choice between dynamic or static configurations is related to the issues of hardware/ software codesign and run- time control. We begin with the former idea.

3.1 Hardware/ Software Co-design.

Reviewing multiple papers about reconfigurable computing one can't help noticing the great deal paid to the hardware/ software codesign. During our research we have found that different parts of the control section can be designed either in software or in hardware, and sometimes in a combination of both. Verifying the correct HW/SW interaction becomes a key problem in the design of a combined system.

[KrPy98] gives a top-level representation of their co-synthesis environment (Figure 3-1). The synthesis starts with a Java code.

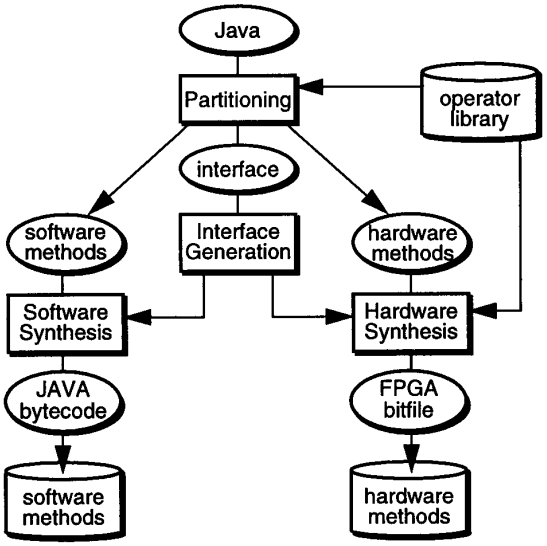

Figure 3-1: Co-synthesis environment

A decision how to split the hardware and software tasks is made in the partitioning section. Then the hardware and software synthesis is done in parallel. The product of the first is the FPGA bitfile. The product of the second is the Java bytecode that is to be run. The authors of this paper use a coprocessor system. In this case both the Java bytecode and the FPGA bitfile will execute in parallel, governing the microprocessor and the coprocessor respectively.

The FPGA bitfile corresponds to the states of the dynamically reconfigurable logic configuration. Its synthesis goes through several steps shown on Figure 3-2.

Starting from high descriptive language the synthesis propagates through different levels. The final level in as a layout description. The difficulty comes from the need to generate multiple configurations and a corresponding finite state machine. The authors of [KrPy98] show the role of the operator library as a designer's choice. They decided to use a strong library approach with a library of generic operators. The operator library is also responsible for providing realistic estimates on technology dependent metrics for the synthesis stages. The metrics are module area, latency, reconfiguration time, etc.

The final product of the hardware synthesis is a gate level description of each configuration and the finite state machine, governing the configuration transitions.

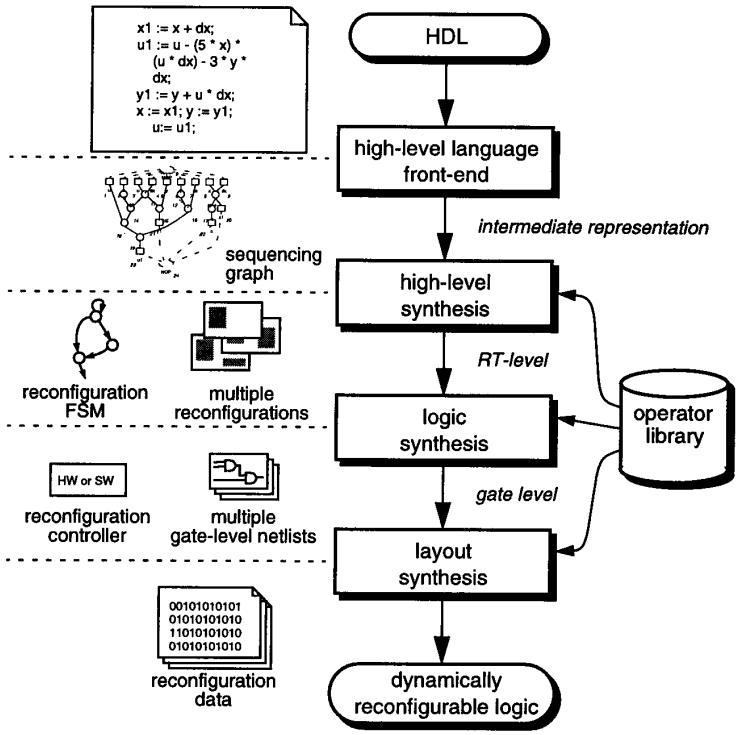

Figure 3-2: Synthesis for dynamically reconfigurable systems

[MeLóJa98] reports a hardware operating system to support reconfigurable computing. The system is designed for auto-configuration. Because there is no description of the system we can't evaluate its feasibility. However the idea of auto-configuration is not to be abandoned, since it gives the advantage of fast reconfiguration.

In our research we have found that different systems use more or less complex compilers. The role of the compiler is proportional to the control portion done in software. We couldn't find a research specifically on the compiler's design. It seems that each group of researchers uses a specific compiler. The compiler's complexity is a function of partially vs. non-partially reconfigurations, as well as dynamically vs. statically reconfigurable designs. The design of well-adapted, universal compiler is needed.

3.2 Run-time Configurations

The ideal hardware module would have a high speed processing for a great range of applications. So far, ASICs provide a high speed but for a specific set of applications. In another approach a universal hardware handles a large range of applications with lower speed. Exploiting the run-time configurability of FPGAs can be the key to overcoming their reduced capacity and speed compared with custom ASICs. The trade-off between area and speed is to be done dynamically. The main parameter is the time the reconfiguration takes place. For efficient computing this time must be small.

In our research we have found several approaches to dynamically schedule the hardware to meet the needs of the software application. The ideas correspond to the general approaches in the reconfigurable design and in the software-hardware codesign trade-offs. We discuss the first group of ideas in section 1 and the second one in section 3.1.

The following topics are noticeable: the role of the run-time manager, a self-controlling circuit approach, a self-modifying circuit approach, a run-time software approach. We now take a look in each of those.

3.2.1 Run-time reconfiguration manager

The reconfigurable hardware implies a great deal of managing the available hardware. [SuLuCh98] shows a possible design. This is an example where the different modules can be designed either in hardware or in software. The method for efficient run-time reconfiguration is dividing the manager into three components: a monitor, a loader and a configuration store Figure 3-3.

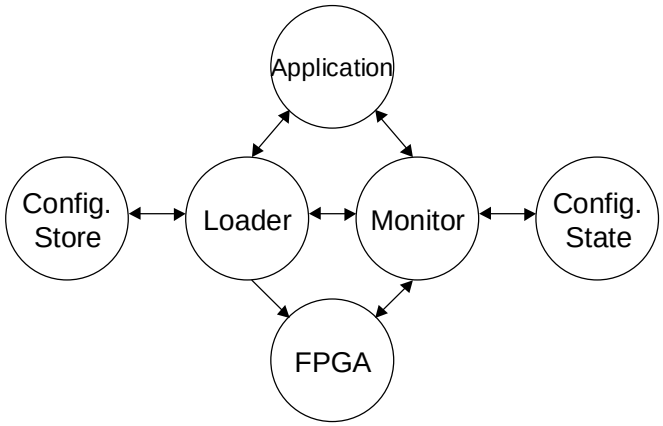

Figure 3-3: Run-time reconfiguration manager

The method can be implemented in software or in hardware, or a combination of both. Some important aspects are:

(a) exploitation of compile-time information for optimizing run-time performance,

(b) flexibility of implementing the reconfiguration manager in software or in hardware,

(c) support for both partially configurable and non-partially configurable FPGAs.

The configuration state includes information about the circuits currently operating in the FPGA.

The monitor uses this information and listens for a request from the application to change the state. If one comes the monitor notifies the loader to install the new circuit. This functional block needs to make quick decisions whether the FPGA configuration needs changing. Some of the

transitions can be data driven therefore part of the monitor resides on the dynamically reconfigurable FPGA. In some cases like image processing the number of cycles for many operations are independent of the data. Thus the monitor can be simplified to contain few counters.

The loader configures the FPGA using data from a configuration store. When it is done the loader signals the monitor and the execution can continue. The loader can set a new clock for the new circuit. For higher speed the loader can be implemented in hardware.

The configuration store contains a directory for the circuit configurations. These configurations are usually stored in the form of address- data pairs. The address indicates the location of a particular cell while the data specify the configuration. This is in essence a fast memory that is very scarce. So a technique to compress states with same configuration information can be used. This technique involves adding an offset to the address so that it points to the right place. Thus many addresses can point to same data. Another method is using same low level configurations for different functional blocks. For example an adder can be used in the design of a multiplier.

This framework can be used with generic reconfiguration managers or with application specific ones. The first case provides flexibility for the expense of inefficiency: time and area. The second method is highly efficient for one or few applications.

There are two reconfiguration methods used with this reconfiguration manager: combined reconfiguration method and partitioned reconfiguration method. The first method gives good results when all possible configurations are know at compile time. Also the sequence of these configurations is also known. Therefore devices supporting partial configuration will benefit from an incremental reconfiguration, rather than full reconfiguration. The second method uses a more complex reconfiguration manager. The idea is to divide the required configuration into subcomponents. Thus the state transition depends on the multiple conditions generated by each subblock.

As already mentioned the reconfiguration sequence can be more or less known at compile time. Thus the monitor will work in more or less static environment. There can be three cases:

(a) The duration for which the current configuration remains valid is known at compile time, and the next configuration is also known.

(b) The duration for which the current configuration remains valid is not known, although the next configuration is known.

(c) Both the duration for which the current configuration remains valid and the next configuration are unknown.

Case (a) is the simplest and the monitor can be as simple as a counter. An example is a video processing system when the hardware is reconfigured to a known state after a fixed number of cycles.

Case (b) requires run-time input from the FPGA or from software to decide when the next configuration is required. Same is true for case (c) however, since the choice of the next configuration is determined at run time, all possible next configurations will have to be produced at compile time or at run time.

The reconfiguration manager shown above is a general idea. Parts of it can be designed in hardware or in software. Now we continue with an idea of a self- controlling circuitry implemented in hardware.

3.2.2 Run-time Self-Controlling Reconfiguration

[M^{c}GLy99] shows the design and physical implementation of a self-controlling dynamically reconfigurable system. The FPGA contains both the controller part and the data processing part. The controller is a part of any FPGA. However designing a complex enough controller, capable of dynamically scheduling, is not a trivial task.

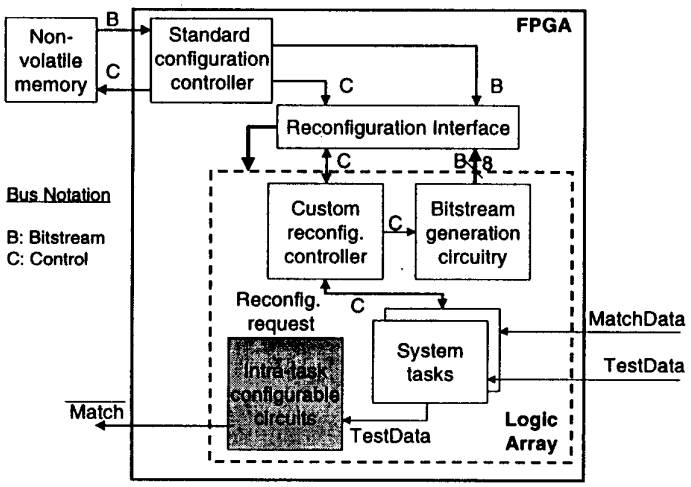

Figure 3-4: System block diagram for pattern matching application

Figure 3-4 shows the main components of the self-controlling system. The operation cycle consists of 10 steps:

1. At power-on, the standard configuration controller transfers bitstreams from the non-volatile memory to the array.
2. The serial bitstream is used to configure the initial application logic, the reconfiguration controller and the bitstream generator circuitry.
3. Configuration control is transferred to the reconfiguration controller and the system begins to execute.
4. Data pass through the system and are compared in the match circuitry.
5. A reconfiguration request is triggered by a new value of MatchData.
6. The value is transferred to the reconfiguration controller and initiates a reconfiguration request.
7. The next reconfiguration is selected and an associated ID is passed to the bitstream generation circuitry.
8. The reconfiguration controller manages the dynamic loading of the internally generated bitstream onto the array, which reconfigures the pattern matching logic.
9. Indication of successful completion is sent to the system tasks and the pattern matching continues.
10. Repeat at step 4.

Once the initial setup has been done the system doesn't need external control. It adjusts itself according to the application data. During reconfiguring the *Match* signal is suppressed to avoid erroneous reconfiguration signals.

The application speed depends on how well the hardware is adapted to it. A novel way of obtaining system flexibility over a wide range of applications is to implement the controller on the FPGA. Then a fully customized implementation of a controller can be developed for each application. The positive effect of this approach is that the controller is as close to the reconfigurable logic as possible. Also fewer discrete devices are needed. This reduces the complexity, optimizes the area and provides operational failure gap. However there are several potential pitfalls. The first issue is gaining array control points from within the FPGA. XC6216 has a well

supported such a feature via the I/O ring. The second problem is the initial configuration. During this time there are two controllers in the system. This is a potential for controller contention. Also the standard controller can configure the array so that it no longer controls the configuration port while still actively using it. In this case the instantiation of the custom reconfiguration controller cannot complete.

Having the controller inside the FPGA doesn't necessarily mean a self- modifying system. The authors in [M^{c}GLy99] decided to not take the risk of having a self-modifying controller.

3.2.3 Run-time Self-Modifying Circuitry

The description of the system in [Do98] has been already given in part 1. This design can map the host FPGA's configuration memory into its own memory space in the SRAM through the system bus. Thus the FPGA is being capable of self- accessing and self- referencing. This is a potential of self- modifying.

In general self- modifying designs haven't gained much publicity since the early computing. Techniques of self- modifying code had been applied to save memory space. In modern engineering these practices are rare. The motivation of this approach however comes from the reduced time of reconfiguration. To be self-reconfigurable the system needs to be partially reconfigurable. The self- modifying circuit can stay alive while part of the datapath is reconfigured.

Self- modifying hardware is an alternative of the existing approaches. It shows some possible advantages in terms of reconfiguration speed. However further research is expected in this area before its feasibility will become clear.

3.2.4 Run-Time Software

In this section we take a look at the other side of the coin, the software part. As mentioned in section 2.1, there exists a notion of "hardware- software co-design". A great number of papers deal with the software aspect of reconfigurable systems. In [HaRo98] the authors show a run-time software environment for their XC62xx (co-processor) based system. It is an extended Linux system.

The extensions include: four new calls to add, remove and manage configurations; a modified scheduler to handle the coprocessor; and a new device to map the coprocessor's flip- flops into a Linux process' address space. The process data structure has been extended to reflect whether the process uses or no the coprocessor.

A system call *ucp_add_config* loads a configuration into the RAM. The arguments of this function are a pointer to the main memory where the reconfiguration data reside, the data length, clock settings and coprocessor register setting. The return value is an ID tag upon which the configuration can be downloaded in the coprocessor. The set of calls contains also: *ucp_rm_config* (deletes a configuration from RAM) and *ucp_run_config* (executes a configuration from RAM). The last call: *ucp_clk*, is used to turn on and off the internal coprocessor clock.

The OS scheduler determines whether or not a process needs the coprocessor and, if yes, which configuration ID needs to be downloaded in the coprocessor. The OS makes a context switch by turning the coprocessor's clock off. Then the state (registers) is copied to the RAM and the new configuration can take place. Then the clock is turned on. As far as the coprocessor is concerned it runs only the current configuration, so it doesn't see the context switch.

The approach in this section is different from the ones seen so far. The OS takes charge of controlling the FPGA. The project is in a beginning stage. The authors believe they can enlarge the class of applications accelerated by the coprocessor.

4 Performance

Performance is one of the weaknesses of the existing reconfigurable designs. We have found that there is no publication with a throughout comparison of different design performances. We have found some approaches for improving performance, among them an VLIW-like approach and a loop-mapping approach. These are shown in the next subsection. In the multiple publications we have also found bits of data showing how fast can the FPGAs reconfigure. We make an attempt to summarize those in the last subsection.

4.1 Methods to speed up the reconfigurable logic execution time

These are more or less general computing approaches. However when applied to reconfigurable systems they may have different flavors.

4.1.1 ILP for Reconfigurable Computing – VLIW approach

The performance of reconfigurable logic depends on the existing level of parallelism in the application. Such a hardware system will benefit if there is a large amount of data to be processed or/and there is a big number of instructions to be executed. The authors of [CaWa98] address the second concept. They find the problem of extracting high instruction level parallelism for reconfigurable logic similar to the problem of finding high instruction level parallelism for VLIW computers. The idea behind *VLIW* (very large instruction word) is to execute many simple instructions in parallel, combining and issuing them in a complex instruction.

The central idea borrowed from VLIW is the notion of *hyper block*. It is formed from a continuous group of basic blocks, usually including basic blocks from different control paths. The blocks don't include rarely taken blocks. Thus a hyper block has a great chance to execute without interruption. The whole program execution can be considered as jumps among different hyper blocks. The challenge for the VLIW compiler is to decide which blocks to include in a hyper block.

[CaWa98] proposes that the compiler schedules which hyper blocks should be executed by the reconfigurable logic and when the program flow should be handled by a microprocessor (coexisting architecture). Simulation results are provided to support the theoretical assumptions.

4.1.2 Mapping Loops onto Reconfigurable Architectures

The reonfigurable systems have been considered so far only as application- specific. The goal of many researchers is to achieve high general- purpose performance in order to compete with the microprocessors' domination. One aspect of a general high performance is how to deal with loops. Configurable logic can be very effective in speeding up regular, repetitive computations, such as loops. A loop is usually a set of operations on a wide set of data. The same set of operations is executed many times so it can be optimized by reconfigurable logic. Once the hardware is specifically configured for the loop the execution time is expected to be little. [BoPr98] presents a way this optimization can be done independently from the complexity of the operations in the loop. To optimize the execution time also means that the reconfiguration time must be minimized, since it is a part of the execution time.

To find an optimal reconfiguration sequence in not a trivial task. If considering only one loop iteration, one may not be able to find a state change sequence that is nearly optimal. Thus the idea is to unroll the loop as many times as needed until an optimal sequence is found. [BoPr98] shows the loop configuration time expense to be $O(pm^3)$, where p is the number of possible operations, each with m possible configurations.

For example, let the reconfigurations between the four operations in a loop take as many cycles as shown in Table 4-1.

	ADD	SUB	DIV
ADD	0	1	5
SUB	1	0	1
DIV	5	2	0

Table 4-1: Example reconfiguration delays

If one iteration of the loop executes sequentially:

ADD
DIV
SUB

there will be eight cycles needed to reconfigure the hardware between the instructions. However if the loop is unrolled twice, assuming parallel iterations, then the two iterations will execute in eight cycles, speeding up the performance twice.

ADD1
ADD2
DIV1
DIV2
SUB1
SUB2

With further reordering it may be possible to achieve even better performance.

4.1.3 Applying microcode techniques

Yet another technique borrowed from the field of general computing is applying microcode techniques. [AcKaLe98] considers design of a PCI-SCI adapter. The Packet Management Unit is described as a finite state machine with 60 states and more than 80 state transitions. The first approach to implement this state machine in a conventional way failed since the cases in the VHDL code got very complicated. Therefore the authors decided to split each transition into several steps copying the idea of microcode usage. This approach helped keep high the target frequency as well as make debugging easier.

4.2 Reconfigurable logic performance

When considering performance of FPGAs one needs to recall that they are RAM- based devices. The possible reconfigurations are stored in memory where they are available for the reconfiguration controller or manager. [M^{c}GrLy99] reports reconfiguration times of 40ns for an ASCII-text search application. The practical reconfiguration time was measured as high as 80ns. This time corresponds to the access times of SRAMs available on the market at the moment. The main observation made in the article is that the reconfiguration speed of this device is slower than the potential logic execution speed in the array.

It can be possible to lower the number of transitions between configurations by applying one of the techniques shown in the previous section.

[Do98] points out another bottleneck in the performance of reconfigurable logic. Reconfiguration state information is transferred through the system bus or a dedicated SRAM interface. This places the upper bound of the reconfiguration speed to be the bus speed. The authors used a simple task of filling a buffer with a known constant to determine the number of instructions being executed within a given time period. The conclusion is that the bandwidth of 8Mbit/s is promising. This is on the background of 7Mbit/s for first generation XC62xx systems. A simple application of pipelining is pointed to be the next step in increasing the performance of the prototype.

Another aspect of reconfigurable hardware performance is the adjustable clock frequency. As mentioned in section 1, [HaRo98] shows a clock frequency between 200kHz and 50MHz. In other publications we also find that the clock can be turned on and off besides being modulated. This research is still in a developing stage. Authors report computational power of 9600 32-bit computations per second or 38400 16-bit computations per second. The results are pointed to be useful for image processing acceleration. Another aspect of this clock manipulation is power consumption saving.

Some performance information can be found in works where the configurable logic is a supplement of a general use microprocessor. [SaGrSp98] shows an attempt to decrease the CPI of commercial superscalar processors. The architecture can handle customized instructions. (A similar architectural idea is shown in part 1 of this research.) Authors report prototype modeling of DSP loops. Extensions like MMX (Intel), MAX (HP), VIS (Sun), AltiVec (Motorola), MDMX (MIPS) and some multimedia instructions from PA-RISC MAX-1 are believed to be good candidates for this FPGA optimization architecture. Speed-ups of 2-4 are expected over this wide range of applications.

[PlMi98] reports an implementation for accelerating satisfiability algorithms. The Boolean satisfiability problem (SAT) has many practical applications, some of those being computer- aided design of digital systems, automated reasoning and machine vision. SAT shows whether there exist a solution for an equation of type

$X1 \wedge X2 \wedge X3 \wedge \ldots \wedge Xm$, Xi being a clause of a number of OR-ed boolean variable.

This type of problem has an exponential worse case delay. The authors show a significant speed-up of the DIMACS benchmark suite using their FPGA implementation versus a software computation. One of the conclusions in this work is the importance of the balance of compilation time, hardware reconfiguration time and hardware execution time. The future work involves implementation of other instance- specific architectures for minimum- cost performance.

[ShLuCh99] reports text- searching and fingerprint- matching algorithms using run-time reconfigurable logic. The first application can process 20 million characters per second. The second one searches 250,000 fingerprint records per second. The authors compare three different configuration of the cell grid. The results show critical paths between 30 and 105 ns with corresponding reconfiguration latencies of 245 and 1 clock cycles. The clock cycle time is related to the critical path delay. This work is one more example of accelerated computing when there exist great quantity of parallel data. The future research is aimed to multimedia applications.

As a conclusion of this subsection we can say that the performance of reconfigurable logic depends on the reconfiguration time, the execution time and the read time. This dependency also involves the compiler's quality, the amount of available hardware and the amount of available data parallelism. As it has been shown, the main bottlenecks are the reconfiguration time and the time to read the current state from a static memory. As a countermeasure designers seek applications with huge data parallelism.

5 Applications

There are some particular fields where reconfigurable computing shows some advantages. For example DSP, where there exists a lot of parallelism and therefore hardware execution brings a lot of speedup compared to software execution. Searching in large databases is another field where research goes in the direction of reconfigurable computing. And one field where completely different advantages count is computing in space. Here issues like maintainability, redundancy, reducing hardware and fast computation on low clock rates count a lot.

5.1 Digital Signal Processing (DSP)

Digital Signal Processing is a field that shows a lot of characteristics that match with the abilities of reconfigurable computing. All reconfigurable DSP systems we have found in literature work with fixed DSP and a reconfigurable co-processor. [GrBr99] provides a good overview why to use reconfigurable computing in DSP.

Most DSP applications need a lot of bit level manipulation. Error control coding is an example. Other DSP tasks as filtering or data compression show a huge amount of parallelism and therefore show a large speedup when executed in hardware. The execution of many DSP applications are quite deterministic, configuration pre-fetching scheduled at compile time should be a useful technique in DSP as shown in [Ha99].

It is rather impractical to realize high precision calculations in RL because of the large amount of hardware that would be used to compute high precision in parallel. The more hardware is used the longer is the reconfiguration time. That's why most of this DSP systems split the tasks as follows: The DSP executes the heavy control and high-precision portions of the applications and the RL computes low precision portions with lot of parallelism. The control of the RL from the DSP is easy to realize because most DSP processors have an on-chip DMA controller.

DSP processors are also good candidates for on-chip RL, because their clock frequencies (up to 200MHz) are still in a range where RL can work.

5.1.1 Video Image Processing

One specific field in DSP where we have found applications with reconfigurable computing is video image processing.

The analog and digital color conversion can be handled by an FPGA system [LuAnSi98]. Application include linear and non- linear filtering, edge detection, image rotation, histogram equalisation, color identification, motion tracking, and creation of video effects. The system is Xilinx 62xx based and resides on a daughter board on a PC.

For example, the color detection works at a resolution of 640 by 480 pixels using 24-bit RGB color. The detection is performed using six 8-bit comparators computing the maximum and the minimum values for each color components. Their output is combined to produce the signal representing the presence of the target color.

As can be seen, this image processing involves massive data transformation. The FPGA part of the system can be bought from vendors at the price of 1000USD. A customized configuration can be done to perform the said applications. If a specialized hardware for image processing is used the price will be considerably higher. Therefore the authors conclude that this system is feasible because of its low price and simplicity. Simple, yet computationally powerful, the system can be used for all the wide range of image processing applications.

Similar applications can be found in [TrZeZa98]. This system consists of three cascaded sections: Filtering, Edge Detection and Region Labeling. The authors show reconfiguration timings of 210, 280 and 150 ns respectively for each of the sections.

5.2 Searching in a Database

Fast search algorithms in databases use hash functions of the data. Each entry is hashed. This leads to a single number, which can be put into a look-up table. To achieve good results several hash numbers of the data with different parameters have to be computed. In [ShLuCh99] they show that a large speedup can be achieved when the computing of the hash function is done with reconfigurable logic. Hash functions are mainly calculated with exor's , exnor's and circular shifts. These operations are much faster when they are executed in hardware than in software. Single hardware solutions are not applicable because of the different parameters we need for the database search. There exists a database system based on this approach to match fingerprints.

5.3 Computers for Outer Space

Computing in space is a very special field that has not much in common with computing on earth [BrBe99]. Components are exposed to extreme mechanical vibrations during launch, microgravity and wide temperature ranges. Another big problem is the high level of ionizing radiation. Compared to computing on earth a lot more components fail under these circumstances. Because you can't just go and replace a component in a satellite this problem has to be solved with redundancy. To have each component 2 or 3 times leads to a huge amount of additional hardware, which means more weight. More weight means much more expensive launch of a satellite. Another problem with conventional hardware is how to replace a single component in a chain with the other redundant component. Reconfigurable computing solves most of these problems. With little more hardware you get a high redundancy where you can easily replace failed parts automatically or over radio from control center.

Another issue is that you have hardware you don't need for the whole mission. For example just to bring the satellite into position then you won't use it anymore. It would be nice to use this hardware for something else afterwards. One can say that there are other solutions for these things. A conventional microprocessor can do that part and later it can do other computations. But there is another problem with that. On a satellite you have very small power budgets, as little as 20W in total for the whole computing. This means you can't have a microprocessor with a high clock rate. Therefore the desired performance can only be achieved with fast hardware algorithms.

Another fact is that lots of the tasks computers in satellites have to do are DSP tasks, as image compressing before transmitting images to earth, signal restoring in communication satellites and so on.

Space organizations as NASA [FiWiGr98] are putting a big effort into research for this field. In 2000 FedSat will be launched [BrGaWi98], one of the first satellites with mostly reconfigurable computing on board.

6 Conclusions

This paper shows several architectural approaches for reconfigurable computing. Issues related to run-time management and HW/SW co-design are presented. Further the paper shows methods to speed-up reconfigurable logic as well as tries to summarize the performance achieved so far. An overview of possible applications is present.

The conclusion we make from this research overview is that the field of reconfigurable logic is in its beginning stage. So far researches concentrate on specific applications rather than on general-purpose computing. Probably this will be the trend for the next 5 to 10 years.

Due to its early stage the reconfigurable computing research lacks throughout performance overviews. We were unable to find papers comparing reconfigurable architecture performance with this of other types of architectures, for example microprocessors. As a result it is hard to say what the shown speed-ups really mean.

Most of the research is done considering one or two applications with systems existing only in prototype stage. Applications that don't exploit a great amount of data parallelism fail to achieve higher performance due to reconfiguration latencies. The latencies correspond to the current SRAM access time and bus speeds.

There are lots of problems to be overcome however we noticed that a lot of research has been put in this area. As a result applications that explore huge amounts of data parallelism have been reported to have some performance speed-ups while running on reconfigurable hardware. Problems of hardware failure and replacement find their natural solutions in the field of reconfigurable computing.

As a final word we can say that the raconfigurable logic research has started recently and it is an incoherent field with lots of ideas flooding around and deep funding pockets provided by some organizations.

Literature:

[AcKaLe98] G.Acher,W.Karl,M.Leberecht, “PCI-SCI Protocol Translations: Applying Microcode Concepts to FPGAs”, In Proceedings of FPL the 8th International Workshop, FPL’98, Springer-Verlag, , ISBN 3-540-64948-4, June 1998.

[BoPr98] K. Bondalapati, V.K . Prasanna, “ Mapping Loops onto Reconfigurable Architectures”, In Proceedings of FPL the 8th International Workshop, FPL’98, Springer-Verlag, , ISBN 3-540-64948-4, June 1998.

[BhKaSi98] D. Bhatia, P. Kannan, K. Simha, “REACT: Reactive Environment for Run-time Reconfiguration”, In Proceedings of FPL the 8th International Workshop, FPL’98, Springer-Verlag, , ISBN 3-540-64948-4, June 1998.

[BhSiKa99] D. Bhatia, K.S. Simha, P.V. Kannan, “NEBULA: A partially and Dynamically Reconfigurable Architecture”, In Proceedings of FPL the 9th International Workshop, FPL’99, Springer-Verlag, ISSN 0302-9743, August 1999

[BrBe99] G. Brebner, N.Bergmann, “Reconfigurable Computing in Remote and Harsh Environments”, In Proceedings of FPL the 9th International Workshop, FPL’99, Springer-Verlag, ISSN 0302-9743, August 1999

[BrGrWi98] J.Brown, S.Gardner, A.Wicks, L.Boland, E.Graham, “The FedSat Spacecraft Design”, 49th International Astronautic Congress, Melbourne, Australia, October 1998

[CaVe99] J.M.P.Cardoso, M.P.Véstias, “Architectures and Compilers to Support Reconfigurable Computing”, ACM Crossroads, 1999
http://www.acm.org/crossroads/xrds5-3/rcconcept.html

[CaWa98] T. Callahan, J. Wawrzynek“Instruction-Level Parallelism for Reconfigurable Computing”, In Proceedings of FPL the 8th International Workshop, FPL’98, Springer-Verlag, , ISBN 3-540-64948-4, June 1998.

[Da99] M. Dales, “The Proteus Processor – A Conventional CPU with Reconfigurable Functionality”, In Proceedings of FPL the 9th International Workshop, FPL’99, Springer-Verlag, ISSN 0302-9743, August 1999

[Do98] Adam Donlin, “Self Modifying Circuitry – A Platform for Trackable Virtual Circuitry”, In Proceedings of FPL the 8th International Workshop, FPL’98, Springer-Verlag, , ISBN 3-540-64948-4, June 1998.

[FiWiGr98] K.Figueiredo, K.Winiecki, T.Graessle, U.Patel, “Study and Utilisation of Adaptive Computing in Space Applications”, NASA GSFC, MAPLD Conference, September 1998

[GrNe99] P. Graham, B. Nelson, “Reconfigurable Processors for High-Performance, Embedded Digital Signal Processing”, In Proceedings of FPL the 9th International Workshop, FPL’99, Springer-Verlag, ISSN 0302-9743, August 1999

[Ha99] S.Hauck, “Configuration prefetch for single context reconfigurable co-processors”, In proceedings of the 6th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA ’98), ACM/SIGDA, ACM, 1999

[HaRo98] G. Haug, W. Rosenstiel, “Reconfigurable Hardware as Shared Resource in Multipurpose Computers”, In Proceedings of FPL the 8th International Workshop, FPL’98, Springer-Verlag, , ISBN 3-540-64948-4, June 1998.

[HoSe98] S. Holmström, K. Sere, “Reconfigurable Hardware – A Study of Codesign”, In Proceedings of FPL the 8th International Workshop, FPL’98, Springer-Verlag, , ISBN 3-540-64948-4, June 1998.

[Jo88] D.W. Jones, “The Ultimate RISC”, Computer Architecture News, 16(3):48-55, June 1988

[KrPy98] R. Kress, A. Pyttel, “ High-Level Synthesis for Dynamically Reconfigurable Hardware/ Software Systems”, In Proceedings of FPL the 8th International Workshop, FPL’98, Springer-Verlag, , ISBN 3-540-64948-4, June 1998.

[LuAnSi98] W.Luk, P.Andreou, A.Derbyshire, F.Dupont-De-Dinechin, J.Rice, N.Shirazi, D.Siganos, “A Reconfigurable Engine for Real-Time Video Processing”, In Proceedings of FPL the 8th International Workshop, FPL’98, Springer-Verlag, , ISBN 3-540-64948-4, June 1998.

[McGLy99] G. McGregor, P. Lysaght, “Self Controlling Dynamic Reconfiguration: A Case Study”, In Proceedings of FPL the 9th International Workshop, FPL’99, Springer-Verlag, ISSN 0302-9743, August 1999

[MeLóJa98] P.Merino, J.C.López, M.Jacome, “A Hardware Operating System for Dynamic Reconfiguration of FPGAs”, In Proceedings of FPL the 8th International Workshop, FPL’98, Springer-Verlag, , ISBN 3-540-64948-4, June 1998

[MiQu98] N. Miller, S. Quigley, “A Novel Field Programmable Gate Array Architecture for High Speed Arithmetic Processing”, In Proceedings of FPL the 8th International Workshop, FPL’98, Springer-Verlag, , ISBN 3-540-64948-4, June 1998.

[PlMi98] M. Platzner, G.De Micheli, “Acceleration of Staisfiability Algorithms by Reconfigurable Hardware”, In Proceedings of FPL the 8th International Workshop, FPL’98, Springer-Verlag, , ISBN 3-540-64948-4, June 1998.

[SaGrSp98] S. Sawitzki, A. Gratz, R.G. Spallek, “Increasing Microprocessor Performance with Tightly-Coupled Reconfigurable Logic Arrays”, In Proceedings of FPL the 8th International Workshop, FPL’98, Springer-Verlag, , ISBN 3-540-64948-4, June 1998.

[ShLuCh98] N. Shirazi, W. Lik and P.Y.K. Cheung, “Run-Time Management of Dynamically Reconfigurable Designs”, In Proceedings of FPL the 8th International Workshop, FPL’98, Springer-Verlag, ISBN 3-540-64948-4, June 1998.

[ShLuCh99] N. Shirazi, W.Luk, D.Benyamin,P.Y.K. Cheung, “Quantitative Analysis of Run-Time Reconfigurable Database Search” In Proceedings of FPL the 9th International Workshop, FPL’99, Springer-Verlag, ISSN 0302-9743, August 1999

[TrZeZa98] A.Trost,A.Zemva,B.Zajc, “Programmable Prototyping System for Image Processing”, In Proceedings of FPL the 8th International Workshop, FPL’98, Springer-Verlag, , ISBN 3-540-64948-4, June 1998.