University of York
Department of Computer Science
Real Time Systems Group

# Using Dynamically Reconfigurable Hardware in Real-Time Communications Systems

## Literature Survey

**Alison Carter**

**November 2001**

# CONTENTS

## 1 Introduction

The development of field programmable systems on a chip (FPSC) heralds an emerging technology of dynamically reconfigurable hardware. The promise of architectures that can be altered on the fly, according to prevailing conditions, adds a new dimension to the design of real-time systems. This survey investigates the potential for dynamic reconfigurability, using current and projected technologies, and looks at possible applications in the field of real-time communications.

The study begins by categorising the types of architecture and strategy that can be used in any reconfigurable system. Having explored the general concept of reconfigurability, the first part of the required infrastructure to be investigated is the *Field Programmable Gate Array* (FPGA). The development and future direction of the hardware are discussed, with reference in particular to suitability for dynamic reconfigurability. This leads on to the methods used to design and implement systems on FPGAs, and the functionality of libraries available for them.

The second issue explored is the scheduling of real-time tasks, both in software and hardware systems. Methods in current use are described, and their relevance to implementation on dynamically reconfigurable systems are discussed.

Finally, the question has to be asked whether dynamically reconfigurable hardware would be genuinely useful in any current or envisaged applications. Examples in the communications area are explored as possible applications for this technology.

## 2 Reconfigurable Systems

"A system can be a product, process or service that converts a set of inputs into a set of outputs" is the broad definition offered by [Stoddart99]. In this context, the system is also assumed to be electronic (rather than mechanical or manual, for instance).

A reconfigurable system is one that changes its form in order to alter its function. For it to be more than "configurable" it must be possible to change it either whilst it is in use, or by taking it out of use for a short time. There are different models for how systems can be reconfigured, in terms of the granularity, the intelligence and the control method required. This section proposes paradigms from various sources outside of computer science and electronics, and looks at their characteristics in terms that might be applied to electronic systems.

### 2.1 Paradigms from other disciplines

#### 2.1.1 The Swiss Army knife.

A penknife has a set of predefined forms created by opening each one of the blades. An equivalent type of reconfigurability is the child's "transformer" toy which can be manipulated into two or three different vehicle and/or robot forms. This is organised by a central "controller", who just chooses one of a set of prepared configurations. The only advantage is the convenience of carrying one tool instead of many separate ones.

#### 2.1.2 Lego

Some reconfigurable systems consist of small blocks that can be put together in any order and then dismantled for rebuilding. An example is the Lego toy, where there are different types of brick but they can be joined in various ways. Taking the building-block size down to an infinitesimal level gives modelling clay or Plasticine. This model assumes a central intelligent controller, which chooses the blocks and reconfigures them according to requirement. The function is defined almost entirely by the physical arrangement of the blocks.

#### 2.1.3 The team

Each member of a football team or army unit has a variety of different capabilities, possibly overlapping in specialisations, and all with some general functions. A reconfiguration means restricting individuals to perform a certain subset of their possible tasks in a specified relative location. Like the "Swiss Army knife" model above, there is a central specialised controller, but each part has intrinsic intelligence (albeit possibly limited) used to minimise the amount of central control required. Executing the reconfiguration may involve communication between the individuals.

#### 2.1.4 The ant colony

Ants have specialised functions within their society (worker, soldier, queen, etc.). According to changing circumstances, the colony can adapt to cope best with their current environment, such as invasion by a predator. There is no overall central control, so this model is an extension of the "team" idea to an entirely distributed intelligence. Communication is vital to the reconfiguration process.

#### 2.1.5 The immune system

The body's immune system is prompted into attacking a particular invading organism by detecting its "foreign" presence. This makes the defensive cells whose shape is attuned to this particular invader proliferate more rapidly than others [Weir97]. These cells, or bacteria that mutate to become immune to particular antibiotics, are following a *selective evolution* model, but much faster than normal evolutionary systems. This is an often-borrowed idea in hardware and software systems, such as the self-repairing and evolutionary systems on a chip in [Moreno98][Ortega97]. In the biological prototype, there is no central organisation, and no real intelligence, but simply a distributed response to the current environment.

### 2.2 Characteristics

#### 2.2.1 Control

In order to reconfigure a system, there must either be an overall controller, initiating and executing the change, or else the parts of the system must respond to external stimuli by changing their own organisation. The control mechanism should be categorisable as

- Central, external intelligent controller: the system can be "downloaded" with a new arrangement from outside as required
- Central, internal intelligent controller: part of the system can calculate and execute rearrangement of the remainder
- Distributed, intelligent: each part of the system can decide on the need for rearrangement, and negotiate changing itself, or other parts.
- Distributed, unintelligent: each part of the system is modified according to predefined rules in response to external events.

#### 2.2.2 Blocks

The blocks making up an electronic system can be thought of as pieces of circuitry which may possibly be parameterisable or programmable. They could be as small as single transistors, or as large as a computer network. In general, complexity and programmability increase together. In terms of function, the blocks may be

- Predefined, single-function: a block of hardwired circuitry that performs a specified function according to its structure.

- Parameterisable/adaptable: whilst the overall function is fixed, certain parameters can be adjusted.
- Intelligent, multi-function: a system containing one or more processors which can be programmed to alter completely the function it performs.

It is, of course, possible to conceive of a heterogeneous system comprising blocks of different sizes and with different levels of intelligence.

#### 2.2.3 Configuration

The function of the overall system is affected by the arrangement and internal configuration of the constituent blocks. Configuration may involve

- Simple choice: one of several possible blocks is selected to be active
- Arrangement defines function: the interconnection of blocks (feeding of outputs from some into inputs of others) defines the functionality of the system as a whole.
- Arrangement forms part of function: when the blocks themselves are programmable or parameterisable, they and their interconnection may be altered to change the system function.

### 2.3 Summary

Reconfigurable systems can be categorised by the complexity of their constituent blocks and the way in which they are reconfigured. Non-technical paradigms discussed above fit this categorisation as shown in Table 2-1. The concept of "parameterisable" (alterable in a small way) is not present in these examples, but is included as it is appropriate for electronic systems discussed later.

| **↓Control Blocks→** | **Predefined** | **Parameterisable** | **Intelligent** |
|---|---|---|---|
| **Central external** | Swiss army knife<br>Lego | | |
| **Central internal** | | | Team |
| **Distributed** | Bacteria | | Ant colony |

**Table 2-1: Categorisation of Reconfigurable Systems**

## 3 Field Programmable Arrays

This section reviews the development of the technology of field programmable gate arrays (FPGAs) and similar devices, and their potential for use in dynamically configurable systems.

### 3.1 History of development

#### 3.1.1 Gate Arrays

The possibility of dynamically reconfigurable hardware has grown from the idea of the *gate array*, first proposed in the 1960s as a way of making custom integrated circuit design easier and cheaper [Read85]. The gate arrays (or *masterslices*) consisted of partially processed silicon chips, with ready-made gates that were not yet connected. These could be prepared in bulk, and the customisation performed with only one or two layers of metal. This, in turn, encouraged the development of automated CAD tools to implement application specific integrated circuits (ASICs), replacing the "polygon pushing" approach of designing each transistor and interconnection individually.

There are several choices to be made in the design of a gate array:

- what should the basic cell's function be, and is it fixed?
- how many should there be, and how many input/output pins?
- how should they be arranged on the chip (in rows, or *sea of gates*)?

In theory, a digital circuit can be constructed entirely from 2-input NAND (or NOR) gates [Boole1854], so any combinational or sequential system could be formed on a gate array providing enough of these gates, sufficient pins and room for interconnect. Early gate array designs used this concept of providing fixed logic gates that could be connected to build the circuit. In the GEC AOI (and-or-invert) gate array, each cell consisted of a 2-input AND, a 2-input OR, and an inverter. These were unconnected, but could be joined or left unused as appropriate to create the required functions. Such an approach, although flexible and amenable to automation, is not efficient in terms of area used and power consumption. It also does not allow for tri-state devices or any memory other than simple registers.

Later devices, such as the UK5000 [Kirk84], had logic cells that were incompletely formed gates with unconnected transistors. The personalisation of the logic cells would create different functions, which could be connected as needed.

The gate array was originally not reconfigurable: once the customisation layers were added, its function was fixed. It could be used wherever the length of time and cost of development was more important than optimising for speed or power, such as for small runs or where time-to-market was vital. It was also used for prototyping, allowing systems to be implemented on silicon and tested prior to investing in a full-custom design.

### 3.1.2 FPGAs

The next important breakthrough was the advent of field-programmable devices. Instead of customisation taking place in the silicon foundry, it could be carried out electrically on the packaged chip, via its pins. The technology used is what is often called an *antifuse*. That is, a normally open circuit that is made permanently conductive when a 5mA programming current is forced through it [Smith97]. This meant greatly reduced costs and time to market for ASIC developers, but once programmed, the setup could not be changed. Using EPROM technology FPGAs could be erased with ultraviolet light, and reprogrammed, but this is a slow process requiring manual intervention.

FPGAs next developed to be *electrically reprogrammable,* allowing embedded systems to be modified or upgraded in the field without physical replacement of hardware components. The technology could be based on EEPROMs, using high voltages (12V or more) to program and erase the array. This could now truly be termed "reconfigurable hardware".

The most flexible form of the FPGA uses SRAM technology to store look-up tables characterising both cell function and interconnection. This means that programming the chip's function amounts to writing data into memory addresses. The disadvantage is that there must be continuous power supplied so that the memory retains its contents, or a facility for download from PROM at power-up. Most current FPGA chips using this technology require the entire chip to be programmed at one time. However, there have been dynamically and partially reprogrammable technologies around since before 1990 [Dettmer90].

### 3.1.3 Field Programmable Systems on a Chip

Quite early in the development of semicustom ASICs, it was realised that some types of component need to be predefined as tailor-made blocks, rather than constructed from cells by each designer. Obvious examples are memory and clock generators. An FPGA chip would therefore include these special blocks.

It seemed at one time that a combination of programmable and handcrafted parts would offer the ideal solution for systems on a chip. For example, a processor core could be supplied with memory and FPGA on-chip to make a complete programmable system. Eventually, the emphasis would shift from “an FPGA with special bits” to “a system on a chip with reconfigurable bits”. However, the ability to implement processors and other specialised blocks on a general purpose FPGA seems to be overtaking this with the provision of soft cores to use in FPSC designs.

## 3.2 Current FPGA technology

A comparison of FPGAs commercially available in 2000 [Krupnova00] shows little variation in the architecture of the basic cell or the functionality. Both one-time programmable and reconfigurable arrays are still used, and programmable combinational logic elements will be based on AND-OR arrays and look-up tables [Salcic98],[Seals97]. FPGA chips usually include programmable registers, with a choice of clocking (global, input, signal) and reset. The memory available is getting bigger and more configurable. Input/output blocks on the chips are programmable as input, output, or bi-directional, with most offering various interfacing capabilities (such as TTL compatibility). Arrangement of cells is usually a rectangular grid, with routing channels. The interconnection between cells can be effected by antifuses between crossed wires as in Figure 3-1 [Seals97], or connections gated by memory elements as in Figure 3-2 [Salcic98].

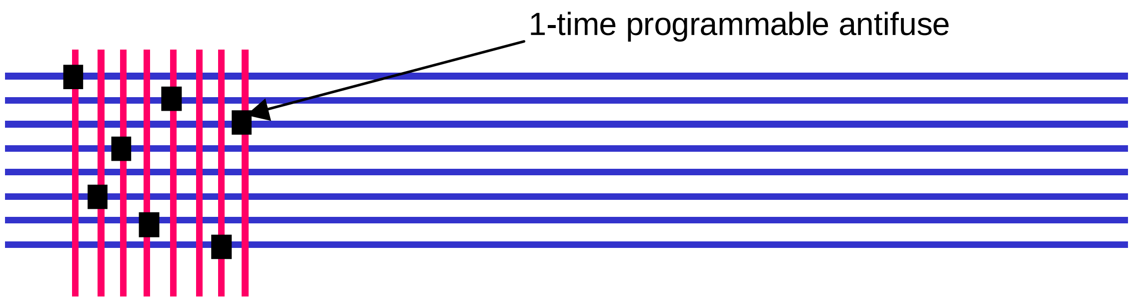

**Figure 3-1:** Antifuse interconnections

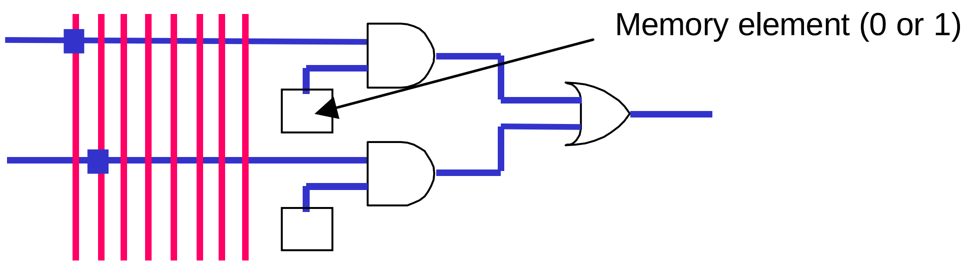

**Figure 3-2:** SRAM-gated connection

To get an idea of the physical capabilities of commercially available FPGAs it is sufficient to look at the Web sites of the two leading manufacturers, Altera and Xilinx. Table 3-1 summarises the facilities available from their current chip families available in August, 2001.

| Family | Logic block | Number | Memory (bits) | Other features |
|---|---|---|---|---|
| Xilinx Spartan-II [Xilinx01S] | Configurable logic block (CLB) is LUT-based | 96-1176 CLBs<br>15k – 200k gates | 16k – 56k block<br>0 – 73.5k in CLB | 86-284 IO pads |
| Xilinx Virtex-II [Xilinx01V] | | 64-15360 CLBs<br>40k - 10M gates | 72k-3456k block dual-port, plus<br>0-1920k in CLBs | 88-1108 IO pads<br>Specialised multipliers |
| Xilinx XC4000 [Xilinx99] | | 64-3136 CLBs<br>1.6k – 85k gates | 2k – 98k (interchangeable with logic) | 64 – 448 IO pads<br>Dynamic reconfiguration |

| | | | | |
|---|---|---|---|---|
| Altera Apex-II [Altera01A] | Logic array block (LAB) is LUT-based logic element (LE), plus embedded system block (ESB) | 16640-89280 LEs 1.9M – 7M gates | 416k - 1488k in addition to logic | 492 – 1440 IO pads |
| Altera Mercury [Altera01M] | | 4800-14400 LEs 120k – 350k gates | 48k – 112k in addition to logic | 303 – 486 IO pads |
| Altera FLEX [Altera99] | | 208 – 1296 LEs 2.5k – 16k gates | 282 – 1500 simple registers | 78 – 208 IO pads |

**Table 3-1:** Commercially available FPGA chip families

## 3.3 Field Programmable Analog Arrays

Alongside the development of FPGAs for digital circuits, there has been parallel work on reprogrammable analogue arrays (FPAAs). They can be based on various circuit elements, such as amplifiers or analog integrators [Pierzchala94]. An example is commercially available from Anadigm (reprogrammable from EEPROM) [ESE01]. They market a module with an on-board microcontroller to reconfigure the FPAA dynamically in 0.1ms. There is nowhere near so large a market for such devices as for digital FPGAs, and they are much smaller with less support software.

## 3.4 Programming and Dynamic Reconfigurability

In order to program an FPGA, the memory elements defining the logic functions, memory characteristics, connections and other parameters must be filled with appropriate data values. This data is termed a “bitstream”, and may be loaded with the FPGA as an active or passive participant [Salcic98]. In active mode, the FPGA loads its internal memory elements from an external chip (ROM) which has been written in the required format. Passive programming involves a microprocessor or microcontroller sending the bitstream as a serial or parallel input. The programming procedure could be regarded as similar to downloading an executable program to an embedded microcontroller’s internal memory, either from external ROM or from a host processor.

FPGA’s can be once-only programmable, in which case the configuration is simply a fast design technique. They may have persistent but erasable memory, so that they can be upgraded occasionally, but still have basically a fixed design. If they have volatile (SRAM) memory, the bitstream is loaded on power-up, and may (dependent on the architecture of the FPGA) be altered dynamically while the chip is running. Time taken to program an FPGA varies according to size and technology, but is of the order of a second.

Advances in dynamic reconfigurability are mostly at the research stage. In 1995, the idea of storing multiple versions of the look-up tables in an FPGA was patented by MIT [DeHon95]. The different versions, referred to as “contexts” are stored locally, and switched in response to a simple instruction, this allowing the chip to switch quickly between several predefined functions (as in the “Swiss Army knife” model described in 2.1.1 above). It is easy to take this approach with more hardware, and taking more time, simply by storing several different bitstreams in ROM and downloading as required.

Reconfiguring an FPGA requires large amounts of data transfer. [Jeong00] explores the use of *partial* reconfiguration, by adjusting the cosynthesis algorithm to schedule tasks in hardware or software taking account of known size and completion times. It has been simulated, but not executed on real hardware. The work done at the University of Glasgow on defining and implementing a run-time reconfiguration manager (RAGE) generalises the requirements for managing an FPGA-based system [Burns97]. The Xilinx XC6200 [Bradley96] has been particularly useful for exploration of dynamically reconfigurable systems, because of its partial reconfiguration abilities. However, it is unfortunately no longer available. The current XC4000 quotes dynamic reconfiguration as one of its features [Xilinx99].

## 3.5 Future Trends

At the hardware level, increases in numbers of gates and clocking speeds in FPGAs mirror those in other silicon products [Rachko00], and this seems set to continue in the same manner. Figure 3-3 illustrates this growth with the speed and size of Intel processors over the past 30 years (data taken from <Intel>. A direct comparison of clock speeds (say 1500 MHz for processors and 200MHz for FPGAs) and gate counts is meaningless due to the parallel nature of hardware algorithms.

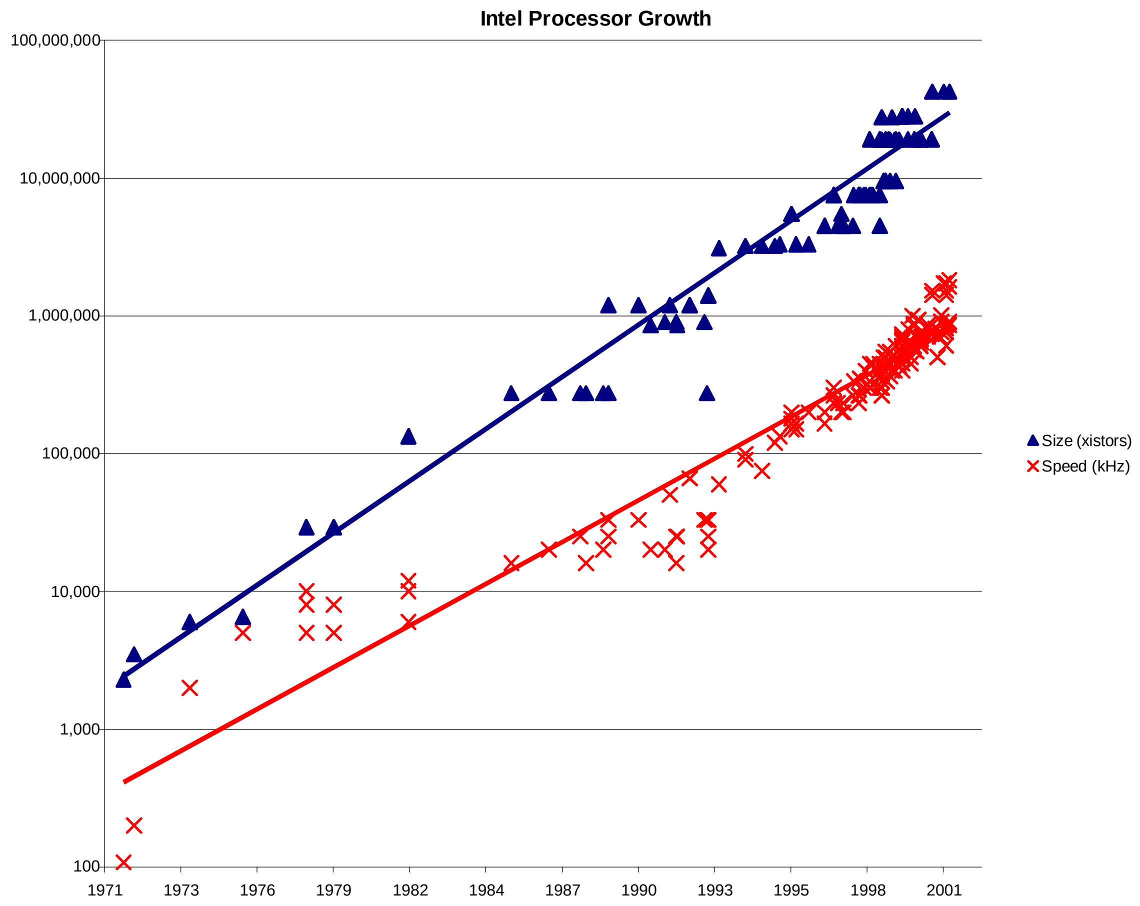

Figure 3-3: Development of Intel Processor Power

Of more interest are potential qualitative changes in FPGA functionality. Looking at United States patents covering FPGAs <Delphion> shows that most development is commercial rather than academic (72 of the 200 recent FPGA patents are from Xilinx), which suggests a mature technology. Of particular relevance here are any developments concerning dynamic and/or partial reconfiguration. Seven of the 200 recent FPGA patents address these directly For example, [Hartmann00] suggests a pipelined logic structure where each stage of the pipeline may be switched to a different (predefined) function by an intelligent controller as required during execution. The concept of “memory planes” within the FPGA can be used to provide more than one (predefined) configuration which can be quickly swapped [Mohan00]. This type of approach could allow for a “double-buffering” arrangement where one plane of memory was updated whilst another was being used, allowing for dynamic updates to the circuit.

## 3.6 Summary

In this section, field programmable gate arrays have been presented at a hardware structural level, where they can be categorised in terms of the models proposed in section 2.2 above as consisting of small, predefined or parameterisable blocks which can be rearranged by a central external controller.

The technology is well established, and has a capability for reconfiguration within a timescale which could be called "dynamic" rather than just "occasional upgrade". Development of multiple memory planes and sectioned, partial reconfigurability is apparent in the literature although not in large-scale commercial use at present.

## 4 FPGA System Design

The view of an FPGA given in section 3 above is analogous to describing the registers and instruction set of a microprocessor which, in practice, is usually programmed using a compiler and predefined libraries. Similarly, a commercial FPGA is sold with design software and prepared component blocks. This section describes the ways in which FPGAs are used, what it means to reconfigure them, and the methods employed to implement systems on them.

### 4.1 Modes of use

The initial application for Gate Arrays, both foundry- and field-programmed, was in rapid prototyping. Previously, the design cycle for application specific circuits involved manual construction of a hardware prototype from discrete components, which was obviously error-prone and not a good indicator of finished system timings. This is still an important aspect of usage of FPGAs, but the emphasis is beginning to shift towards use in final products.

As an example, military systems need high reliability and long-term stability but rapid development cycles to be ahead of competitors. The DoD RASSP project [Richards94] was specifically looking to prototype signal processing systems for applications such as radar. A similar current European project, Espadon, is now designing a methodology for the use of *commercial off-the-shelf* (COTS) components, including the use of FPGAs in the deployed systems [Madahar00].

The use of multiple FPGAs in realistic communication systems poses challenges of speed and complexity. In a paper describing the construction of an IP packet forwarder using 19 FPGA chips [Miyazaki99]. Miyazaki states that "FPGA-based emulators … cannot be applied to real-time telecommunications data processing, which often requires at least 20MHz". Advances in technology have overtaken this, but the requirement of multimedia applications for ever more bandwidth will always leave a gap between the capabilities of custom hardware and that achievable in reconfigurable systems. The question is whether development time from the definition of a standard to the release of products will render custom design insufficiently responsive to the market.

The reconfigurability of FPGAs makes them ideal for field upgrades to systems. As described in section 3.4 above, a new bitstream can be downloaded containing a new version of the circuitry. This makes upgrading a circuit as simple as, say, loading a patch for an operating system. It can be supplied on disk, or via the web. In fact there is growing interest in internet-enabled FPGA chips allowing remote upgrades of the circuit design.

The next level of flexibility is dynamic reconfiguration: altering the configuration while the system is operating (or at least with small, insignificant down-time). Just because the hardware might be capable of this (as discussed in section 3.4) does not necessarily mean that support software is available to recompute the configuration in the required time.

### 4.2 What is hardware, and why reconfigure it?

There are obvious advantages to a reconfigurable system, such as flexibility and ease of modification or upgrade. This could be taken to refer to any programmable system. The word "programmable" is used in many contexts: for instance, the *programmable logic array* (PLA) is an unchanging hardware block, and an FPGA chip is "programmed" by its bitstream. It is therefore useful to propose definitions of programmable software and reconfigurable hardware.

- A software system consists of an algorithm represented and executed as a set of instructions. This usually takes place serially, or with limited, well-defined parallelism, on a single or defined set of processors.
- Reconfigurable hardware consists of a set of interconnected components, each with a (currently) defined function, usually operating with extensive parallelism. The algorithm performed depends on the interconnections and the individual component functions. Reconfiguration involves specifying new component functions and/or interconnections.

Characteristics of software implementations mean that they are comparatively easy to alter and upgrade. For any system whose potentially reconfigurable parts could be programmed entirely in software using existing processors, it would be difficult to find any reason for looking for hardware implementations. To justify the use of hardware for any particular application, it must be possible to say one or all of:

- It meets timing constraints that equivalent software does not
- It contains elements (e.g. analogue interfaces) that cannot be implemented in software
- It can be designed more easily, more reliably, and/or more quickly using reconfigurable hardware

In order to justify using FPGA-type reconfigurable hardware, rather than standard chips or specially designed ASICs, some of the following conditions must hold

- It meets power/size requirements that full hardware implementation does not
- It needs a flexibility of reconfiguration not available in a fixed hardware implementation
- It can be designed more easily, more reliably, and/or more quickly using reconfigurable hardware

The use of general processors and the lack of parallelism makes software implementations less efficient in terms of speed than hardware equivalents for some types of algorithm. [Pryan01] describes the use of FPGA-based DSP solutions as a faster alternative to programming DSP processors.

In a detailed analysis of the types of FPGA available, and the uses to which they can be put, [Hauck98] gives the following types of application:

- Hardware logic emulation (faster than software simulation)
- Multimode hardware (switching between predefined functions, rather than implementing several separate hardware blocks)
- Coprocessors
- Multi-FPGA implementations of specific algorithms (e.g. encryption)
- Trainable systems (e.g. neural networks)

However, he notes that "it is not clear that any application has already been developed that can drive wide-scale adoption of this technology". Some interesting concepts are put forward, such as the idea of "virtual hardware" by analogy with virtual memory, for multimode (switchable) systems with sets of alternative hardware configurations ready for loading.

In real-time systems with specific time-response constraints, it may be necessary to implement some programmable parts in hardware. If there is also a limitation on the size/power-consumption (such as in hand-held devices), then hardware reconfiguration is an obvious solution.

Hardware reconfiguration can also take the place of software emulation in the increasingly important area of interoperability in networked systems. It is already easy to get FPGAimplementable simple processors <FPGA CPUs>, so the idea of completely reconfiguring a processor to “look like” something else is not far away.

Another reason for using reconfigurable hardware is where interfacing to external devices requires reconfigurable analogue characteristics that cannot be anticipated in the original design.

### 4.3 Hardware-Software Codesign

Because of the differing advantages of hardware and software implementations, most systems consist of a mixture of the two. Software, at the lowest level, consists of a sequence of instructions, and obviously requires a processor to interpret and execute these, and memory in which to store the program and the working data. In addition, some parts of the function may be implemented directly in hardware, interacting with the software at appropriate points. The system design then centres around portioning into which parts are to be implemented in software and which in hardware.

A traditional design flow for codesign is shown in Figure 4-1 (adapted from [Douglass00])

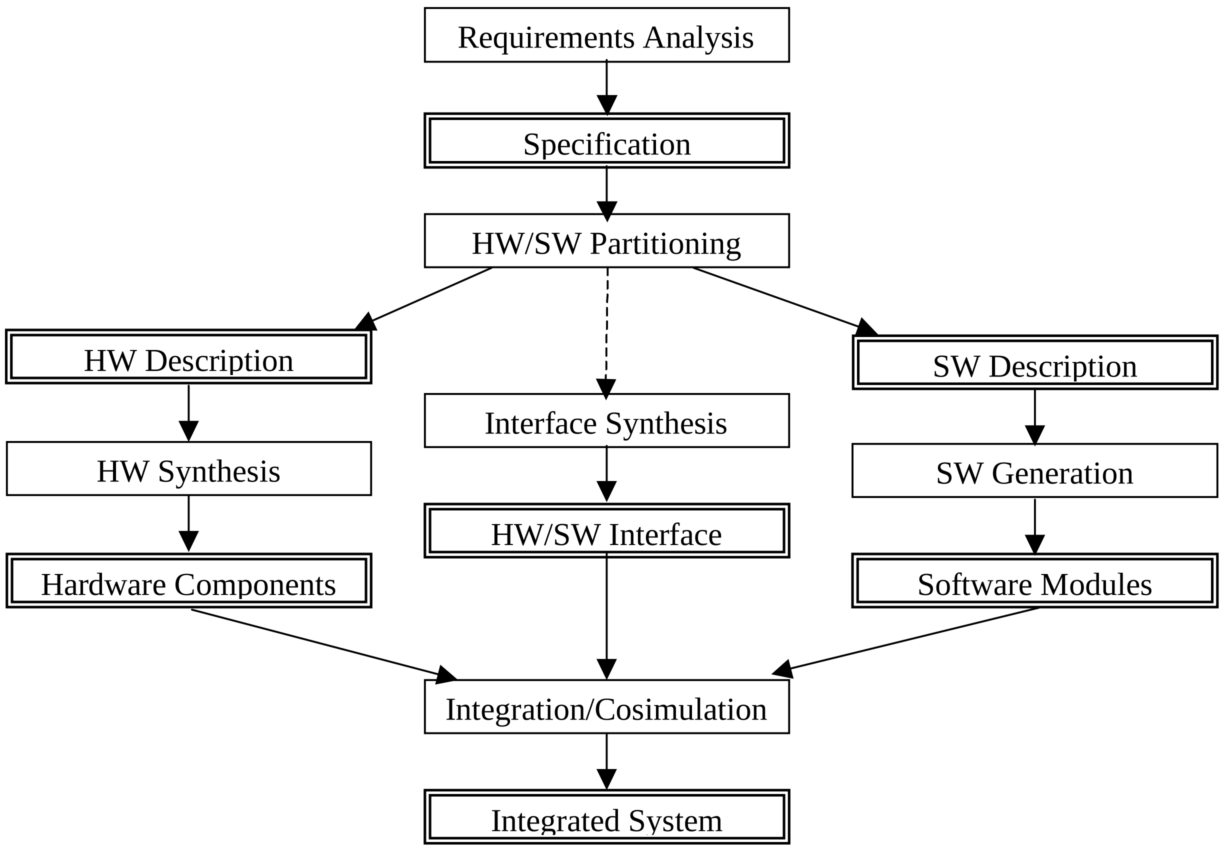

Figure 4-1: Traditional Hardware/Software Codesign Flow

This assumes that a hard partitioning choice is made early in the design process, after which the two parts are implemented separately and subsequently integrated. Current design techniques are moving towards independent system design and simulation, with later partitioning. The use of dynamically reconfigurable hardware adds to the complexity of the codesign task.

- Hardware blocks communicating with the software may be reconfigured
- The hardware on which the software runs may be reconfigured
- The boundary between hardware and software may be moved dynamically
- The hardware design/implementation software becomes part of the finished product, not just part of the design process.

## 4.4 Design methodologies currently employed for FP devices

FPGA hardware vendors provide software to enable designers to target systems to their particular chips, and general CAD companies sell tools which can be used with different FPGA hardware. The general procedure involved in implementing a circuit on a single FPGA is shown in Figure 4-2. An example of a current FPGA design system is the Mentor Graphics *FPGA Advantage* suite <Mentor> consisting of *Renoir* (block-level design and synthesis), *ModelSim* (simulation) and *LeonardoSpectrum* (place and route).

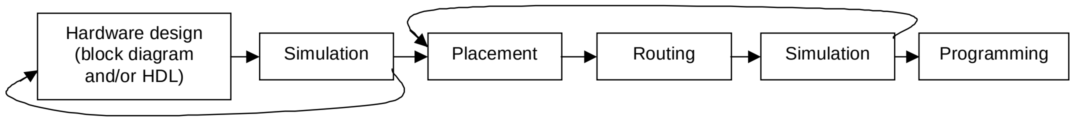

Figure 4-2: FPGA Design Procedure

The designer has to model the system hardware using functional blocks, which may be taken from a library or programmed in a hardware description language (VHDL or Verilog) [Ashenden00]. The design is hierarchical, with the bottom level being elements that can be implemented directly in the look-up tables of the FPGA (simple logic gates) or other basic building blocks (e.g. memory elements). The placement and routing stages assign these blocks to specific places in the array on the chip, designate the values to be programmed into the LUTs to perform the required function, and route the necessary connections between them. This is largely automated, but the placement stage in particular will usually benefit from manual guidance from the designer.

It is no longer necessary, when using commercial FPGAs, to build the design right down to gate level. There is a growing collection of commercially available IP (predesigned high-level blocks whose value is the *intellectual property* involved in their design) including communications peripherals (e.g. Ethernet controllers) and simple processors [Aycinena01] <FPGA CPUs>. Table 4-1 shows examples of some of the IP cores currently commercially available (2001). There are also freely available cores on the Internet <Free IP>.

| Vendor | IP Core Function |
|---|---|
| Mentor Inventra | 10/100Mbps Ethernet Controller |
| Mentor Inventra | M8052 Microcontroller |
| ARC Cores | Bluetooth Controller |
| Eureka Technology | 8086 Microprocessor |
| 4i2i | Streaming video CODEC |

**Table 4-1:** Some Commercially Available FPGA IP cores

The design hardware is simulated in is original block/HDL form, and also after automatic synthesis into lower-level hardware blocks. These blocks are initially vendor-independent, so that the design, and any library and IP blocks used, are not specific to a particular FPGA chip.

At the final place-and-route stage, the user selects a particular FPGA chip on which to implement the design. An estimated gate-count, together with the known number of interconnections required, can be used to guide the selection. The tool would normally be installed with a family of target chips from a particular vendor. Certain IP blocks may be optimised for specific target families, but generally any design may be targeted to any large enough chip. Resimulation, taking into account the physical layout, can check that timing and power constraints are still met, and if not the placement and routing can be repeated with different user-defined guidance parameters.

All of this is currently focussed on quick time-to-market and design re-use for one-time programmed (or possibly later upgradeable) single-chip devices.

## 4.5 Implications for dynamically reconfigurable systems

There are two problems with this design process: the size and complexity of realistic systems, and the time and manual input needed for the implementation.

Many current studies of reconfigurable hardware look at single-chip FPGA systems. Whilst useful as insights into the technology and its applications, the real challenge is in the reconfiguration of large, complex systems. These may incorporate fixed processors, and several (may be different types) of FPGA. Several projects have used large arrays of FPGAs to implement massively parallel algorithms [Hauck98]. In any such architecture, reconfiguration of the FPGA must either be carried out within a tight specification so that it does not affect the rest of the system, or other parts (e.g. software) must be altered accordingly.

Reconfiguration of an FPGA using a standard design system would necessitate running several design – synthesise – layout cycles in advance to have different implementations ready to load. That is, switchable configurations ("multimode" hardware) can be implemented, and the bitstreams stored ready to load as required. Switching time depends on the speed with which the hardware can be configured. There are ways in which this can be improved on. If partial reconfiguration is possible, the hardware for the different modes can be designed with minimal differences, so that partial reconfiguration can take advantage of smaller changes [Heron99]. Suppose, for example, a system on a single FPGA chip consisted of a microprocessor core with variable logic alongside it. It would obviously be a waste of time to run the whole chip layout cycle again if the microprocessor itself did not change: it should be possible to mark parts of the array as "in use", and replace other parts with a new design. In a system using the "virtual hardware" approach, similar to context switching of software processes, [Levinson00] points out that the information stored in memory in an FPGA can be split into permanent and transient. The "permanent" information defines the configuration, whereas the "transient" is the current contents of any registers at present time. In any state-change that maintains the hardware configuration, only the registers need to be reloaded.

For truly dynamic reconfigurability, either the hardware design cycle needs to be dramatically shortened in time, or there must be intrinsic reuse of parts of the implementation (not just the design, but the physical positioning on the FPGA). The former may be possible if the hardware is designed automatically, under program control, rather than with manual guidance. The latter is more implementation-dependent.

## 4.6 Summary

Most of the time spent in implementing hardware on an FPGA is in the design and layout, rather than the programming, so this stage needs to be made as efficient as possible if dynamic reconfiguration is required. The reuse of predesigned blocks (analogous to software class libraries) is already speeding up design. Taking into account this higher-level view of the design process, reconfigurability scenarios can be modelled as shown in Table 4-2.

| **↓Control Blocks→** | **Predefined** | **Parameterisable** | **Intelligent** |
|---|---|---|---|
| **Central external** | Multimode hardware | Virtual hardware with context switching | Hardware emulation of different processors |
| **Central internal** | Multimode hardware including on-board switching | Trainable hardware (e.g. neural network) | Soft processor modifying its own parameters |
| **Distributed** | | Self-modifying hardware e.g. majority decisions for fault tolerance | Multiple small reconfigurable processors |

**Table 4-2: Model for FPGA System Reconfigurability**

The following types of reconfiguration scenarios could be envisaged:

- Software runs on a fixed processor, controlling predefined updates to the hardware
- Software on a fixed processor *redesigns* and reconfigures hardware
- Hardware is autonomously self-modifying
- Software runs on a reconfigurable processor (either multimode or dynamically redesigned)

## 5 Scheduling Real-Time Systems

"If something anticipated arrives too late it finds us numb, wrung out from waiting, and we feel - nothing at all. The best things arrive on time." [Dorothy Gilman, *A New Kind of Country*, 1978]

In real-time systems, the timing of interaction with the environment is part of the specification [Burns96]. If reconfigurable hardware is to be included in real-time systems, any timing analysis must take into account possible different hardware configurations, and the time taken to switch between them. This section looks at methods for ensuring timing requirements are met in the scheduling of software and hardware operations, and how these might be affected by reconfigurability.

### 5.1 Worst-Case Execution Times in Software

The primary requirement of a real-time system is that the timing of its outputs should be predictable, either individually or statistically depending on the application. This predictability becomes more difficult to achieve the more complex the application. Inputs may be periodic (with a regular arrival frequency), sporadic (with at least a known minimum inter-arrival time) or entirely random.

In order to design a software system that will meet timing requirements, it is necessary to predict how long tasks will take to execute. At least, the worst case execution time (WCET), which is an upper bound on the time taken, is needed as it usually does not matter if a task completes early. The WCET needs to be *tight* (not too much of an over-estimate), but also *safe* (never exceeded) in order to be useful [Engblom00]. The calculation (or estimation) of WCET involves both high-level (source code) and low-level (microcode) analysis, and tends to be highly specific to a particular compiler and architecture [Bernat00]. This would be a big problem for reconfigurable systems, as the architecture might be liable to change.

The high-level analysis consists of looking at how paths through the code are followed, such as how many times a loop may execute, when different conditions exist for *if-then-else* branches, and where functions are called. This information is usually not entirely available by static analysis, and has to be augmented by programmer annotation [Li95] where value ranges of variables cannot be inferred from the code. For simple tasks, it may be possible to analyse all possible paths through the code, but this can easily become infeasible for larger programs. The problem can alternatively be expressed as a collection of constraints, and solved analytically.

Low-level analysis means working out how long machine-code instructions take to execute. The straightforward clock-periods per instruction cycle calculation can be affected by pipelining, instruction caching and data caching. As processors become more complex, with more "speeding up" mechanisms, the exact calculations of the time taken to execute any one instruction becomes more context-dependent and difficult to determine.

### 5.2 Scheduling tasks on a single fixed processor

Before looking at the complexities of scheduling tasks on reconfigurable processors, it is necessary to understand methods used to do this a traditional processor. Real-time applications generally require more than one task to be executed, and each has its own constraints to be met. In order to share the processor time between them, the following are needed:

A mechanism for starting the next task running. This can be as simple as a procedure call (cyclic executive) or interrupt service routine, but will more usually be a queue of processes, with saved states. The process at the head of the queue can be loaded for execution at regular intervals (round robin), immediately (preemptive) or when the current task completes (non-preemptive).

A method for allocating priorities (unless all processes are considered equal). This decides which processes get to the front of the queue and therefore get more immediate servicing. The priorities may be statically or dynamically assigned.

Metrics (such as WCET) to allow priorities to be calculated, and so that an analysis of the feasibility of the schedule can be made. Along with metrics for the processes themselves, the constraints and scheduling goals need to be expressed mathematically (e.g. is it minimum time, or minimum lateness that is important).

In addition, each scheduling method makes assumptions about the characteristics of the processes for which it is designed (e.g. periodicity).

### 5.2.1 Cyclic executive:

If the inputs to a system are known to be periodic at multiples of a basic period $T$, and $\sum_{k=1}^{N} C_k < T$ (where $C_k$ is the WCET of process $k$ associated with a particular periodic input), a non-preemptive cyclic executive scheme can be calculated in advance [Burns96],([Liu00] calls this a "clock driven" system). There is no need for saving process states and context switching: tasks can be called as procedures and each runs to completion. Given simple enough input timing, such a statically defined schedule could be used for a reconfigurable system in the case of multimode hardware, where different executives could be defined for each pre-defined configuration. The actual reconfiguration procedures could be considered as additional tasks to be included in the schedule.

### 5.2.2 Round Robin

This is an algorithm in which all eligible processes are given time in turn, then preempted at the end of a time-slice (or quantum) if they are still running. Processes are put to the back of the queue when they complete, in an effort to give fair share to all. It is possible to introduce a weighting factor, to give extra quanta to some "more important" processes [Liu00]. The method has the advantage of simplicity, and is used in operating systems such as POSIX, but does not cope well with sporadic processes, or processes with precedence constraints.

The use of a round robin scheduler could be imagined for switching between different configurations in a dynamic system. Here, the task switching also involves setting up a new configuration. The reconfiguration could not be seen as extra tasks in this case, as each has to complete, and be done in a specific order in relation to processes. The time-slice length will affect the overall efficiency of the system, because of the overhead involved in task switching, which could now become much larger. For example, if the time-slice is set at 10ms, and the switching takes 1ms, there is a 10% overhead added to execution time, whereas if the time-slice is increased to 50ms, there is only a 2% overhead. However, as the time-slice gets longer there will be more occasions when a process cannot use its whole slice, and an extra switchover is needed or time is wasted. Every process $i$ will complete in a time less than $qN\left\lceil \frac{C_i}{q-s} \right\rceil$, where $q$ is the time quantum, $N$ is the number of processes, and $s$ is the context switching time.

### 5.2.3 Priority-Driven Scheduling

Where ready processes are taken from the queue according to priority, the allocation of these priorities becomes the main issue.

**Rate Monotonic** [Liu73]: Rate-monotonic scheduling is particularly designed for the servicing of periodic events, as found in real-time systems with synchronous inputs. The RMS scheduling algorithm simply gives highest priority to the task with the shortest period, and allows higher priority tasks to preempt. It will effectively schedule $N$ processes if the sum of the *utilisations* $\frac{C_i}{T_i}$ of all the processes is less than $N\left(2^{1/N}-1\right)$. This levels out around 70% (limit = $\ln 2 \approx 0.6931$) as process numbers increase, as shown in Figure 5-1

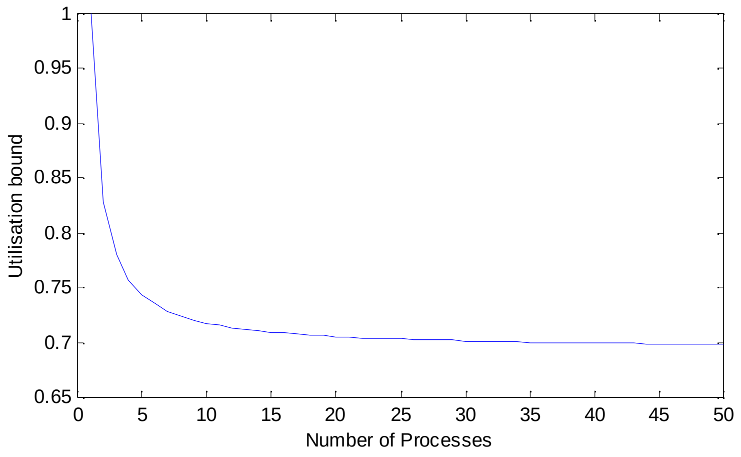

**Figure 5-1**: Utilisation bound for guaranteed rate monotonic scheduling

Although a set of processes with given timing constraints may be schedulable by this method, the *response time* (from input, causing the queuing of a process, to output, assumed to be completion of a process) is dependent on the completion of higher priority processes.

**Deadline:** If a real-time process is invoked to capture input in an embedded system, the important factor in scheduling is actually meeting a deadline, rather than taking a particular amount of time. There are therefore scheduling algorithms that assign priorities according to deadlines by which a task must start or complete. If the task with the earliest finishing deadline is given highest priority, there is no need to have periodic tasks. This is called "deadline driven" [Liu73], "deadline monotonic" [Burns96] or "Earliest Deadline First" (EDF) [Liu00]. It can be scheduled if for a given set of processes if *any* other method can ([Liu73] for periodic processes, [Liu00] more generally). It is also possible to consider starting deadlines (release times), rather than finishing deadlines for processes, and to extend the analysis to allow for release jitter [Audsley95].

**Critical Sections:** Interaction between processes, either in terms of competition for resources or temporal sequencing, means that simple priority assignments by period or deadline can lead to priority inversion and deadlocks [Audsley95]. To overcome this, processes can be made to *inherit* priorities dynamically from those they are blocking. This is insufficient to avoid possible long chains of processes blocking each other, so further action can be taken to assign priorities according to use of resources. This can be done by assigning *priority ceilings* and only allowing processes whose priorities are higher than this to enter a critical section that uses this resource [Burns96].

**Varying priority with resources:** An example of dynamic priority assignment in a changing system that might be relevant to reconfigurability is given in [Ma00]. Here the problem of energy efficient scheduling for mobile applications is addressed by assigning tasks two different types of priority. One represents the importance of the task ($P_I$ = 0 for unimportant, 1 for vital), and the other represents its energy consumption ($P_E$ = 0 for using all the energy, 1 for using none). The overall task priority is then assigned according to a formula which can be simplified as

$$P(t) = (1-r)P_E E(t) + rP_I(1-E(t))$$

where $E(t)$ represents the ratio of residual energy at time $t$ to the total energy budget, and $r$ is an adjustable parameter to give more or less weight to the energy-saving aspect of the algorithm. As energy is used up, $E(t)$ decreases and the priority is driven more by the importance of the task, as shown in Figure 5-2. Here, the residual energy is assumed to be decreasing linearly with time. When there is plenty of energy, paradoxically more emphasis is given to the energy consumption aspect. The overall task priority determines the period of operation in an example application of a portable GPS wayfinder.

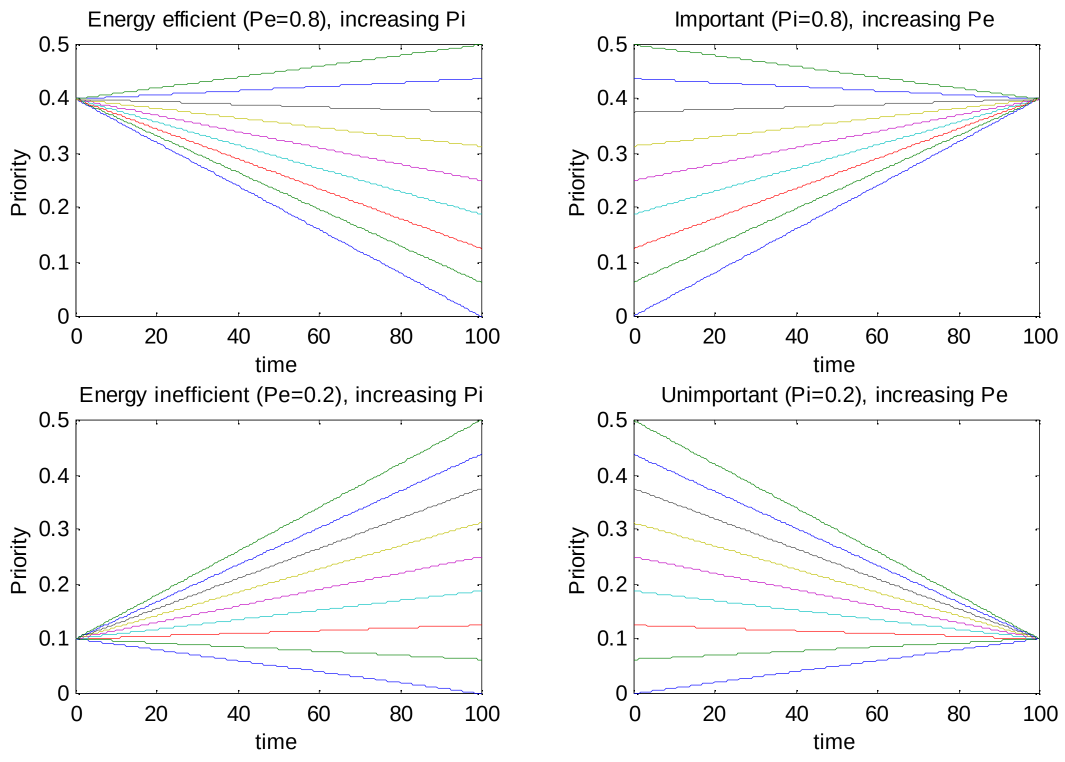

**Figure 5-2:** Priorities (with $r$ = 0.5) over time using Ma's formula

This is an example of a schedule having a metric (power remaining) and a goal (avoiding running out of power) which dynamically alters the behaviour.

**Value-Based Systems:** An approach to allocating scarce resources when scheduling has been developed under the category of *value-based* methods [Burns98]. These methods assume that resources are limited, and will at some stage become insufficient to allow all processes to meet their specified criteria (deadlines). It is then necessary to decide which should be neglected and which must be scheduled, so that the system can cope with a temporary period of overload. The *value* is a metric attached to a process, or group of processes, which may vary according to time and context. It is used to decide which processes are admitted to (or excluded from) the run queue when resources are limited. This mechanism is in addition to any priority assignment acting within the queue.

**Heterogeneous Systems:** Many analyses of scheduling algorithms assume a homogeneous set of task characteristics, whereas in reality there will probably be a mixture of periodic and aperiodic, real-time and non-critical, those with known WCET or deadlines and those without. [Liu73] suggests that a combination of Rate Monotonic and Deadline Driven priority assignment can be very effective. Even in areas such as DSP programming, which is characterised by simple periodic inputs and outputs, there is an emerging trend towards the need for more sophisticated scheduling [Dubin01]. It is likely that in dynamically reconfigurable systems, scheduling will need to use a combination of algorithms at different times and in different circumstances.

#### 5.2.4 Multiprocessor Scheduling

If there is more than one processor available, or if some of the operation can be carried out in hardware, then the scheduling becomes more complex and issues of synchronisation must be addressed. [Stallings98] categorises multiprocessor implementations according to their granularity (assuming each application contains many tasks) as

- Independent, which is really multiple applications rather than parallelism in a single application.
- Coarse, where individual processors perform tasks that could be run on a single, multi-tasking system, but all at once rather than switching between them.
- Medium, where strongly interacting process threads of a single application are scheduled to run on several processors.
- Fine, where a single instruction stream contains parallelism that can be distributed over several processors, such as in an occam Transputer environment.

The issues of static versus dynamic scheduling still apply. It is still common for the programmer to have to specify the allocation to multiple processors, rather than any automatic allocation. Methods of automatic scheduling can take into account not only the timing of processes, but also the communication / synchronisation between them, so that communicating processes run simultaneously.

Of course, there is no reason to assume homogeneity of processors. At one extreme, all processors are interchangeable, and at the other each is dedicated to a particular task.

### 5.3 Real-time characteristics of hardware systems

Hardware systems are intrinsically parallel, and their timing characteristics have traditionally been modelled in a different manner from software systems. There is still the concept of a "worst-case execution time", but the factors that can affect the timing are not all the same as for software:

- The same low-level hardware block can take different times to execute, depending on the loading (fan-out), so at different places in the circuit, similar blocks have different delays.
- The time taken by a hardware block to complete processing may depend on its inputs (for example, a multiplier may take different times for different data)
- The requirement to synchronise the arrival of data from several different sources means that the WCET of a hardware block depends on the slowest path through it.

#### 5.3.1 Datapath model of synchronous systems

As an example of timing analysis of hardware systems, the lossless discrete integrator (LDI) filter described in [Rouse94] and shown in Figure 5-3 is a synchronous signal processing structure with several parallel branches. The input is a periodic data stream which is input at a specified frequency, and an output is calculated at the same frequency, but after some delay. Delay (memory) elements are clocked at the input frequency, so their output represents a previous data value. The other components are fixed point adders and multipliers (each multiplying by a different, constant coefficient), and have an inherent delay which is to some extent data-dependent. The delay also depends on topology, technology and physical layout.

Simulation packages typically use very conservative delay estimates, but these can be back-annotated more accurately from a physical implementation.

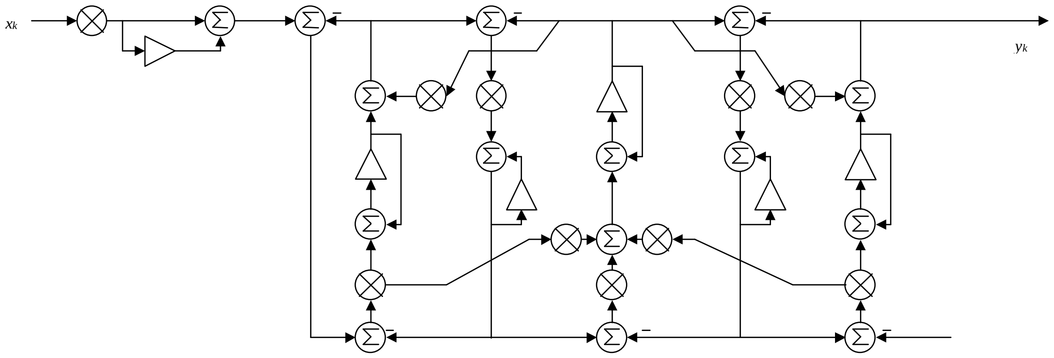

**Figure 5-3:** LDI Filter

The correct timing operation can be confirmed by ascribing a worst-case execution time to each component, and identifying the *critical path* from one delay element to the next (the path with the longest time). This can be done using *signal flow* graph techniques. For example, given the times adder = 1, multiplier = 2, there are two, equal critical paths of 14, as shown in Figure 5-4.

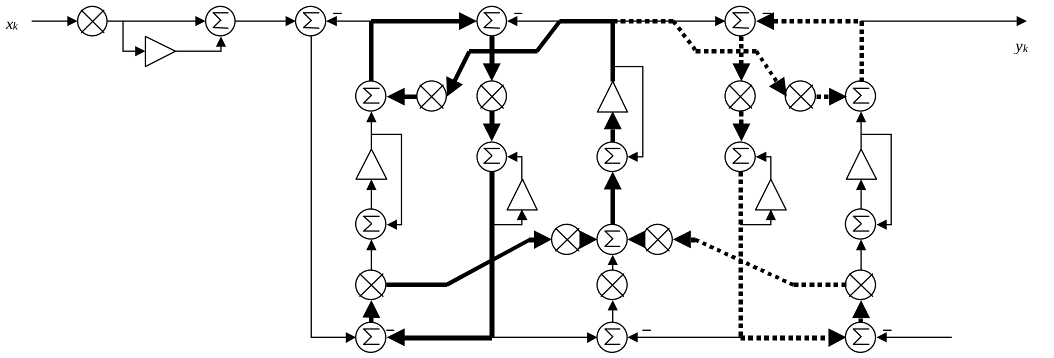

**Figure 5-4:** Critical Paths

Hardware systems do not generally have all their memory elements (registers) clocked at the same rate like this one. In this case, a *controller*, in the form of a finite state machine, is responsible for enabling capture of data at each register at the appropriate times. This makes the design of the controller similar to designing a scheduler for a multiprocessor software system (it fact, it can be considered a multiprocessor system where each processor has one dedicated task, as described in 5.2.4 above). There is, however, the added high-level design decision about how many hardware components are needed. For example, if an algorithm requires two independent additions, this could be done with a single adder, the first result being stored in a register while the second is calculated. Alternatively, if speed is important, two separate adders could be used.

With more complex synchronous systems, the techniques above apply, but at a block level, rather than individual components. There is still a static analysis of paths through the system, based on WCETs of the blocks, and synchronisation occurring when data is captured in memory.

### 5.3.2 Self-timed systems

A self-timed system is a hardware system with the equivalent of dynamic scheduling. That is, elements of the system have a method of signalling that they have completed, and are inhibited from continuing until all their inputs are ready. This is done at logic gate level, enabling micro-pipelines to be established, with components running at their actual speed rather than using worst-case estimates . Methods used in self-timed logic are reviewed here, as their dynamic nature and distributed control of timing may find parallels in the scheduling of reconfigurable systems [Grass97].

One method used is the introduction of a *space* token between bits. Instead of data items of just 0 and 1, there are three values (0, 1 and *space*). This may be implemented by using three different voltage levels, or using two wires for each signal. A transition from *space* to 0 or 1 signals that a calculation is complete. Conversely, the presence of a *space* on an input inhibits further processing. Although this method allows for effective self-timed operation, it introduces overheads on the time (and on the hardware and power consumption).

Another method is to introduce auxiliary circuitry to generate a *ready* signal when the calculation is complete, which is fed into the *start* control of the next function(s) in line. This assumes that the delay is data-dependent, so the auxiliary circuit uses the input data, and possibly extra information from the logic block, to calculate and implement an appropriate delay.

A third method involves somehow detecting activity in the logic block. Depending on the technology, this might involve measuring currents, or looking for internal voltage level changes. In any case, the assumption is that the circuitry is normally quiescent, and there is some way of telling when nothing is taking place inside it.

## 5.4 Real-time reconfigurable systems

The problem of scheduling in real-time reconfigurable systems depends on the type of reconfiguration being undertaken. [Levinson00] looks at the problems associated with stopping and restarting processes that are running in hardware on FPGAs in order to implement preemptive scheduling. Instead of the handful of registers and required to specify the state of a software process (in addition to its memory contents), the hardware process may have a large number of internal registers. In this example, the (changing) register contents are disentangled from the (semipermanent) parts of the FPGA definition bitstream representing the hardware layout, requiring the ability to*read back* the configuration of the FPGA.

## 5.5 Summary

The scheduling of real-time applications often assumes a simple model of processes (e.g. independent and periodic). For implementation on dynamically reconfigurable hardware, the issue of scheduling reconfiguration (a relatively slow operation), and the necessity for the right hardware to be in place for particular processes, adds to the complexity. Taking the reconfiguration scenarios mentioned in section 4.6 above, different scheduling methods would be appropriate

**Multimode hardware:**

Standard software scheduling techniques within each mode, plus global scheduling of switchover between modes taking account of precedence of operations

**Virtual hardware with context switching**

Standard software scheduling techniques with modification to take account of long switching times

**Hardware emulation of different processors**

Standard software scheduling techniques appropriate to each processor

**Trainable or Self-modifying hardware**

Self-timed or hardware controller, plus software scheduling making allowance for reconfiguration times

**Soft processor modifying its own parameters**

Extension to software scheduling with dynamic adjustment to suit processor changes (value based techniques could be useful here)

**Multiple small reconfigurable processors**

Extension to multiprocessor scheduling techniques to allow for reconfigurability.

## 6 Real-Time Communications Applications

This section looks at the applications in the real-time communications field that might benefit from the use of dynamically reconfigurable hardware. The aspects that are particularly relevant are

- *interoperability* of different standards
- *multifunction* in a single device or system (adaptation to application or data)
- *adaptation* to environment

### 6.1 Software Radio

The concept of *software-defined radio* has conflicting definitions, but a detailed discussion in [Pereira00] is summed up as

> "we envision Re-configuring on demand not only the terminal but also the serving network(s) and the services they provide ….Upon this open framework, we envision truly 'platform'-independent applications, no longer exclusively developed by or for operators, capable of adjusting themselves to the serving network capabilities … and the terminal characteristics, negotiating with the network to obtain the best possible service taking into account the user profile."

The basic premise is that everything is reconfigurable. This ranges from the protocols at all levels (e.g. using GSM or UMTS), through network management (e.g. reconfiguring a network to cope with different loading profiles), to applications (e.g. an application that normally provides streaming video switching to still monochrome images if the quality of service is insufficient). The general message is that nothing is predefined.

Much of this is, as in its name, software defined. However, there are elements of such a system that are ideally suited to reconfigurable hardware. This is particularly the case for mobile handsets, where small size and low power consumption are important.

The CAST project (Configurable Radio with Advanced Software Technology) [Madani00] aims to demonstrate an architecture for intelligent reconfiguration of the physical layer in wireless communication networks. This covers interoperability between GSM and UMTS in the bottom level of a communications system, together with adaptation to prevailing conditions and user requirements. The hardware configuration proposed for a demonstrator consists of a board with four Xilinx XCV600 FPGAs, with controller and memory, plus a programmable DSP processor board, and reconfigurable analogue components. This gives a mixture of hardware and software reconfiguration.

The economic necessity for rapid time-to-market in an environment of complex, evolving standards is one of the reasons for the interest in reconfigurability in this area [Dick00]. There are also constraints on the size and power consumption of mobile equipment, together with enormous versatility requirements in terms of applications, loading and physical environments.

### 6.2 Adaptive Signal Processing

A common DSP task is the creation of a filter which can adapt according to conditions. For example, the removal of cockpit noise in an aircraft pilot's voice communications equipment needs to adapts as that noise varies. This is a smaller, and better defined task than that described in 6.1 above. In general, it will involve recalculating the coefficients in the numerical implementation of a digital filter, which is implemented in software on a DSP processor. However, given a hardware implementation of such a filter, it could be necessary to reconfigure that hardware to give optimum performance with the new coefficients. [Dempster95] describes an algorithm for minimising the number and complexity of adder blocks in a given filter implementation. The architecture is entirely dependent on the actual coefficient values, as it searches for common "power of 2" shifts in the set of numbers to maximise the sharing of calculation. It would therefore need to be reconfigured in order to alter the filter in any way. The viability of this idea for adaptive filtering depends on the time taken to recalculate the architecture and reconfigure the hardware, and the frequency of the required reconfiguration, compared with the speed/efficiency advantage gained by using a hardware implementation. It is suggested in [Courtney00] that a more regular, less efficient multiplier architecture gives better reconfiguration performance that one specifically tailored to the coefficient structure.

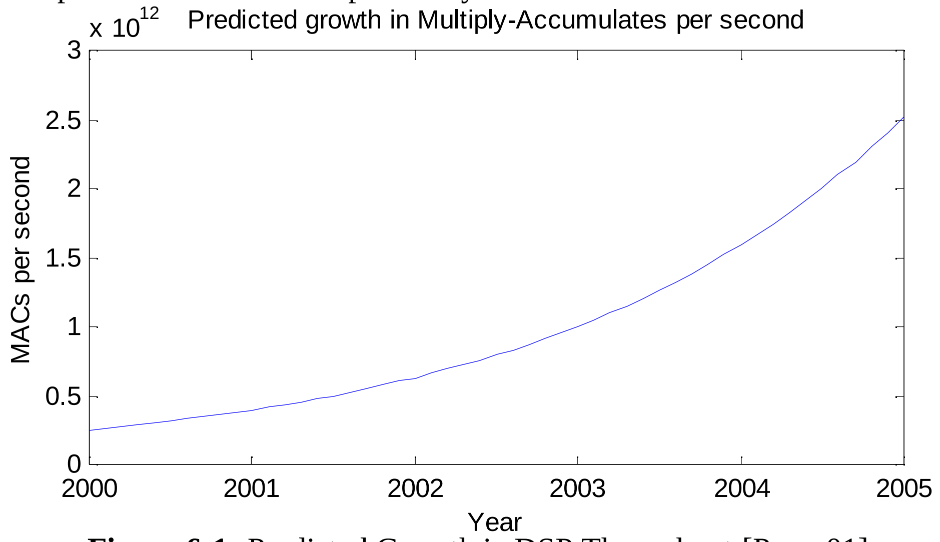

**Figure 6-1:** Predicted Growth in DSP Throughput [Pryan01]

An adaptive approach to system identification (such as might be used to characterise the channel in a communications application) is described in [Pasquato99]. This tries to match a given (noisy) system response with an IIR (Infinite Impulse Response) filter, having first arrived at a rough estimate using an FIR (Finite Impulse Response) structure. Such a system could theoretically be implemented by reconfiguring hardware from an FIR to an IIR, with or without hardwired coefficients in the circuitry. A suitable scenario could involve different hardware configurations in sequence:

- General FIR with adaptable coefficients (in registers)
- Mapping of FIR to IIR
- General IIR with adaptable coefficients (in registers)
- Calculation of structure of efficient, fixed coefficient IIR
- Fixed coefficient IIR

## 6.3 Reconfigurable Protocols

With the rapid growth in bandwidth of communication systems, the type of application and content is set to change unpredictably in the future. In a presentation of the idea of dynamic protocol architectures [Crane98], it is asserted that "no fixed set of protocols *can* satisfy the needs of all future applications". Whilst it is obvious that no set of currently defined protocols will be suitable for all future unknown demands, the more useful interpretation given is that architectures should be designed in which an interpreter can demand-load protocols from a library. This can then be updated as required, and allows communications programs access to dynamically loadable, and parameterisable, protocols, which can be used to maintain the quality of service required by an application.

The use of reconfigurable hardware for adaptable protocols is already being explored by the PRO$^3$ project <PRO3>. This aims to design a protocol processor for both data and telecommunications applications, which includes reconfigurable hardware. This will handle low-level protocol functions, while the higher layers are dealt with by a processor core included on the chip.

In the area of ad-hoc (peer-to-peer) networking, distributed intelligence in the communicating devices sets up and controls the protocols, including the routing [Haas99]. Here we are dealing with mobile devices, so they must be small, with low power consumption, and there is the inherent reconfigurability of a network which routing nodes can join and leave as they wish. As with other communications applications, there will be a quality of service to maintain, consistent with the application, which may include real-time components.

## 7 Summary

Current FPGA systems allow fast, cheap implementation of systems, with the facility for upgrades in the field. Some small amount of dynamic reconfigurability exists currently, but without real support for commercial exploitation. The length of the design process required for an unpredicted change precludes dynamic, on-demand alteration of systems, but multimode systems (loading predefined alternatives as required) are certainly feasible. One possible field of further research is methods of redesigning hardware which minimises the layout change required, and therefore might allow reconfiguration in real time.

Even with multimode hardware, the implications of reconfigurability for scheduling real-time systems are that extra delays are introduced while the system reconfigures. The improvement in quality of service afforded by the reconfiguration must be worthwhile. Standard scheduling algorithms would need to be adapted to take into account this extra factor.

It is obvious that adaptable systems are needed in many applications, but this ability to change could usually come from altering software, rather than reconfiguring the hardware. Applications which need to be fast and low power, have high intrinsic parallelism, or for other reasons must be implemented in hardware, are possible candidates for this technique. For multimode hardware, replication is an alternative to reconfiguration if there are no size or cost constraints. Those applications investigated include software radio, adaptive signal processing and reconfigurable protocols. All of these might be used in a hand-held device which must be small and low power, as well as needing high speed.

## 8 References

[Altera99] Altera Corp., *FLEX 8000 Programmable Logic Device Family*, Data Sheet v10.01, 1999

[Altera01A] Altera Corp., *Apex II Programmable Logic Device Family*, Data Sheet v1.1, 2001

[Altera01M] Altera Corp., *Mercury Programmable Logic Device Family*, Data Sheet v1.1, 2001

[Ashenden00] Ashenden, P., *A Designer's Guide to VHDL*, Morgan Kaufmann, 2001 ISBN: 1-55-860674-2

[Audsley95] Audsley, N.C. et al, *Fixed Priority Pre-emptive Scheduling: An Historical Perspective*, Real-Time Systems, March 1995

[Aycinena01] Aycinena, P, *Focus Report: PLD Tools*, Integrated System Design, September 2001

[Bernat00] Bernat, G., Burns, A. and Wellings, A. *Portable Worst-Case Execution Time Analysis Using Java Byte Code*, 6th International EUROMICRO Conference on Real-Time Systems , 2000

[Boole1854] Boole, G., *An investigation into the laws of thought, on which are founded the mathematical theories of logic and probability*, Macmillan, 1854 (republished Dover, 1958, ISBN 0-486-60028-9)

[Bradley96] Bradley, K F, and Watson, J, *Reconfigurable Processing with Field Programmable Gate Arrays*, ASAP 1996

[Burns96] Burns, A. and Wellings A., *Real-Time Systems and Programming Languages*, Addison-Wesley, 2nd Edition. 1996

[Burns98] Burns, A. Prasad, D., Bondavalli, A., di Giandomenico, F. et al, *The Meaning and Role of Value in Scheduling Real-Time Systems*, Journal of Systems Architecture, 1998

[Burns97] Burns, J., Donlin, A., Hogg, J., Singh, S. and de Wit, M, *A Dynamic Reconfiguration Run-Time System*, IEEE Symposium on Field-Programmable Custom Computing Machines, 1997

[Courtney00] Courtney, T., Turner, R., and Woods, R, *An Investigation of Reconfigurable Multipliers for use in Adaptive Signal* Processing, IEEE Symposium on Field-Programmable Custom Computing Machines, 2000

[Crane98] Crane, J. S., Pryce, N. G. and Magee, J. N., *A Dynamic Protocol Architecture for Multimedia Communications*, Technical Report, City University, 1998

[DeHon95] DeHon, A, *Dynamically Programmable Gate Array With Multiple Contexts*, US Patent 5742180, 1995

[Dempster95] Dempster, A G and Macleod, M D *Use of minimum-adder multiplier blocks in FIR digital filters*, IEEE Trans Circuits and Systems II, vol 42, no 9, September 1995

[Dettmer90] Dettmer, R., *User-programmable logic: chasing the gate array*, IEE Review, Vol 36 No 5, 1990

[Dick00] Dick, C., harris, f., and Rice, M. *Synchronisation in Software Radios – Carrier and Timing Recovery Using FPGAds*, IEEE Symposium on Field-Programmable Custom Computing Machines, 2000

[Douglass00] Douglass, B.P., *Doing Hard Time*, Addison Wesley Longman, 2000, ISBN 0201489375

[Dubin01] Dubin, J. *Rapid Prototyping of Real-time DSP Software Using DSP/BIOS Kernel*, Embedded Systems Show, 2001

[Engblom00] Engblom, J. and Ermedahl, A., *Modeling Complex Flows for Worst-Case Execution Time Analysys*, IEEE Real-Time Systems Symposium, 2000

[ESE01] Embedded Systems Engineering, vol. 9 no1 Dec/Jan 2001

[Grass97] Grass, E., V. A. Bartlett and I. Kale, *Completion-detection techniques for asynchronous circuits*, IEICE Transactions on Information & Systems, vol. E80-D, no. 3, 1997.

[Haas99] Haas, Z.J. et al, *Guest editorial: wireless ad hoc networks*, IEEE Journal on Selected Areas in Communications, Volume: 17 Issue: 8, Aug. 1999

[Hartmann00] Hartmann, A C, *Dynamically Reconfigurable Logic Networks Interconnected by Fall-through FIFOs for Flexible Pipeline Processing in a System-on-a-Chip*, US Patent 6096091, Aug 2000

[Hauck98] Hauck, S, *The Roles of FPGAs in Reprogrammable Systems*, The Proceedings of the IEEE, Vol 86 #4, April 1998

[Heron99] Heron, J-P and Woods, R.F., *Accelerating Run-Time Reconfiguration on FCCMs*, IEE Symposium on Field-Programmable Custom Computing Machinery, 1999

[Jeong00] Jeong, B, S. Yoo, S.Lee and K. Choi, *Hardware-Software Cosynthesis for Run-time Incrementally Reconfigurable FPGAs*, Asia-South Pacific Design Automation Conference, 2000

[Kirk84] Kirk, I. H., *Developing Gate Array Software*, Silicon Design Vol 1, No. 8, 1984.

[Krupnova00] Krupnova, H., and G. Saucier, *FPGA Technology Snapshot: Current Devices and Design Tools*, Proc. 11th International Workshop on Rapid System Prototyping, 2000

[Li95] Li, Y-T., Malik, S. and Wolfe, A., *Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software*, IEEE Real-Time Systems Symposium, 1995

[Liu00] Liu, Jane W. S., *Real-Time Systems,* Prentice Hall, 2000

[Liu73] Liu, C. and Layland, J., *Scheduling Algorithms for Multiprogramming in Hard Real Time Environment*, Journal of the ACM, 1973

[Levinson00] Levinson, L, Männer, R., Sessler, M., Simmler, H., *Preemptive Multitasking on FPGAs*, IEEE Symposium on Field-Programmable Custom Computing Machines, 2000

[Ma00] Ma, T. C-L and Shin, K. G., *A User-Customisable Energy-Adaptive Combined Static/Dynamic Scheduler for Mobile Applications*, IEEE Real-Time Systems Symposium, 2000

[Madahar00] Madahar, B, *Environment For Signal Processing Application Development and PrOtotypiNg (ESPADON)*, NATO Symposium on Commercial Off-The-Shelf Products in Defence Applications, Brussels, 2000

[Madani00] Madani, K., *Configurable radio with Advanced Software Technology (CAST) – Initial Concepts*, IST Mobile Summit, 2000.

[Miyazaki99] Miyazaki, T., Murooka T., Katayama M. and Takahara A., *Transmutable Telecom System and Its Application*, IEEE Symposium on Field-Programmable Custom Computing Machines, 1999

[Mohan00] Mohan, S. and Trimberger, S.M., *Method for Configuring FPGA memory planes for virtual hardware computation*, US Patent no 6047115, April 2000

[Moreno98] Moreno, J.M., Madrenas, J., Faura J., Cantó E., Cabestany J. and Insenser J.M., *Feasible Evolutionary and Self-Repairing Hardware by Measurement of the Dynamic Reconfiguration Capabilities of the FIPSOC Devices*, Springer Verlag Lecture Notes in Computer Science, 1998

[Ortega97] Ortega, C. and Tyrrell, A., *Biologically Inspired Reconfigurable Hardware for Dependable Applications*, IEE Colloquium on Hardware Systems for Dependable Applications, 1997

[Pasquato99] Pasquato, L. and Kale, I., *System Identification via Hybrid FIR-IIR Adaptive Filtering*, IEEE Instrumentation and Measurement Technology Conference, 1999

[Pereira00] Pereira, J.M., *Re-Defining Software (Defined) Radio: Re-Configurable Radio Systems and Networks*, IEICE Transactions on Communications, Vol E83-B No 6, June 2000

[Pierzchala94] Pierzchala, E., Perkowski, M, *High Speed Field Programmable Analog Array Architecture Design*, FPGA 1994

[Pryan01] Pryan, D, *Xilinx Xtreme DSP Initiative*, Embedded Systems Show, 2001

[Rachko00] Rachko, V., *Bridging the FPGA Design Gap*, Electronic Component News, September 2000

| | |
|---|---|
| [Richards94] | Richards, M. A., *The Rapid Prototyping of Application Specific Signal Processors (RASSP) program: overview and status*, IEEE International Workshop on Rapid System Prototyping, 1994 |
| [Read85] | Read, J W, *Gate Arrays: Design and Applications*, 1985, ISBN 0-00-383012-5 |
| [Rouse94] | Rouse, C.J. and Carter A.J., *Exploring Delay/Area Trade-Offs of an LDI Filter using a Natural Based Algorithm*, Proc. Int. Symposium on Circuits and Systems, June 1994. |
| [Salcic98] | Salcic, Z., *VHDL and FPLDs in Digital Systems Design, Prototyping and Customisation*, Kluwer, 1998, ISBN 0-7923-8144-0 |
| [Seals97] | Seals, R. C. and Whapshott, G. F., *Programmable Logic: PLDs and FPGAs*, Macmillan, 1997, ISBN 0-333-65570-2 |
| [Smith97] | Smith, M J S, *Application-Specific Integrated Circuits*, Addison Wesley, 1997, ISBN 0-201-50022-1 |
| [Stallings98] | Stallings, W, *Operating Systems: Internals and Design Principles*, Prentice-Hall, 1998, ISBN 0-13-917998-4 |
| [Stoddart99] | Stoddart, A.G., *Systems Engineering: is it a new discipline?*, IEE Control and Computer, v10#3, 1999 |
| [Weir97] | Weir, D. and Stewart J., *Immunology 8th Ed*, Churchill-Livingstone, 1997 |
| [Xilinx99] | Xilinx, Inc., *XC4000E and XC4000X Series Field Programmable Gate Arrays*, Product Specification, v1.6, 1999 |
| [Xilinx01S] | Xilinx, Inc., *Spartan-II 2.5V FPGA Family: Functional Description*, Preliminary Product Specification, v2.1, 2001 |
| [Xilinx01V] | Xilinx, Inc., *Virtex-II 1.5V Field-Programmable Gate Arrays*, Advance Product Specification, v1.6, 2001 |

## 9 Links

| | |
|---|---|
| <Delphion> | Delphion IP Network, http://www.delphion.com/ |
| <Free IP> | The Free IP Project, http://www.free-ip.com/about.htm |
| <FPGA CPUs> | FPGA CPU news, http://www.fpgacpu.org/ |
| <Intel> | Intel Corp., *Microprocessor quick reference guide*, http://www.intel.com/pressroom/kits/quickreffam.htm |
| <Mentor> | Mentor Graphics Corporation, *FPGA Advantage version 5.0*, at http://www.mentorg.com/fpga-advantage/ |
| <PRO3> | EC Framework 5 IST, *Protocol Processor Project, IST Project 11499*, at http://www.pro3-processor.com/ |