Object-Oriented Reconfigurable Processing for Wireless Networks

Andrew A. Gray, Clement Lee, Payman Arabshahi, Jeffrey Srinivasan
Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive, MS 238-343, Pasadena, CA 91101 USA

***Abstract* – We present an outline of reconfigurable processor technologies and design methods with emphasis on an object-oriented approach, and both full and partial dynamic reconfiguration. A specific broadly applicable architecture for implementing a software reconfigurable network processor for wireless communication applications is presented; a prototype of which is currently operating in the laboratory. This architecture, its associated object oriented design methods, and partial reconfiguration techniques enable rapid-prototyping and rapid implementations of communications and navigation signal processing functions; provide long-life communications infrastructure; and result in dynamic operation within networks with heterogeneous nodes, as well as compatibility with other networks. This work builds upon numerous advances in commercial industry as well as military software radio developments to space-based radios and network processing. The development of such radios and the network processor presented here require defining the correct combination of processing methods ("objects") and developing appropriate dynamic reconfiguration techniques as a function of system goals and operating parameters.**

I. Introduction

Processors in use in the telecom industry, especially within the rapidly growing wireless communication sector, are increasingly called to task on a diversity of applications, and faced with ever-stringent operational constraints. Many potential applications for these processors often cannot be anticipated at design time. Moreover, the value of adapting to new requirements and continuing cost-efficient, high-performance operation within such dynamic networks is extremely high, driving the need for reconfigurability [1-5].

The reconfigurable processor architecture is typically a composite of generic microprocessors, field programmable gate arrays (FPGAs), digital signal processors (DSPs), as well as traditional digital and mixed-signal applications specific integrated circuits (ASICs) and discrete analog-circuits. It is currently determined by experienced design engineers who decide upon and consider numerous tradeoffs involving, among others, complexity, cost, development time, mass, size, flexibility, power consumption, and reliability in order to achieve target system requirements and performance metrics. Given the variety of processors and development platforms available in the commercial market, and the increasing complexity of problem solutions, these judicious decisions are generally extraordinarily complex while at the same time being sensitive and potentially error prone and costly.

The work described in this publication was carried out as part of a task funded by the Technology and Applications Program (TAP) at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

To address these issues, we present design methods that facilitate flexibility (leading to prolonged processor lives), and dynamic reconfigurability, along with a generic reconfigurable processor architecture. Dynamic partial reconfiguration of the network processor will greatly increase the value of the processor within the network. For instance the processor's ability to perform a very large number of temporally separated (time-multiplexed) functions will dramatically increase network elasticity and compatibility (see Fig. 1).

The paradigms will also empower the design engineer with tools required for rapid prototyping and implementation of a wide variety of signal processing functions. We focus on a "network processor" as a hardware/software object of choice and example within a wireless communication network.

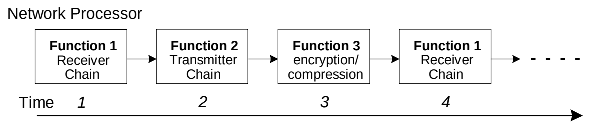
Network Processor
Function 1 Receiver Chain
Function 2 Transmitter Chain
Function 3 encryption/ compression
Function 1 Receiver Chain
- - - -
Time
1
2
3
4

Fig. 1. Time multiplexed processor functions.

II. Architecture Definition

For our purposes we assume that all processing performed by the reconfigurable network processor (shown in Fig. 2) is entirely digital, and consider the case of conventional continuous-time processing; although reconfigurable analog systems are also possible [6,7].

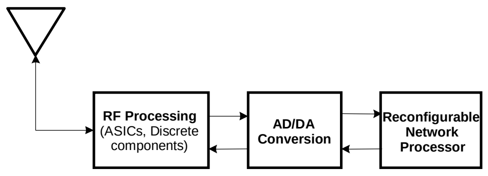
RF Processing (ASICs, Discrete components)
AD/DA Conversion
Reconfigurable Network Processor

Fig. 2. Generic model of a network processor.

Our key considerations are flexibility and computational performance for which the primary measure is fixed/floating operations per second.

Figures 3 and 4 represent an abstract depiction of three general classes of processor types under consideration for our generic architecture definition: ASICs, FPGAs, DSPs, and microprocessors [8,9].

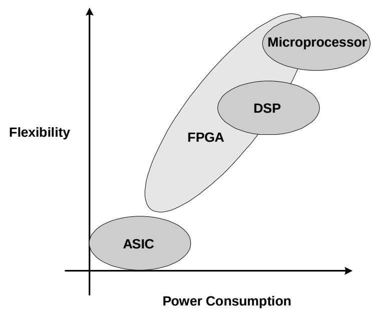

Fig. 3. Illustration of flexibility versus power consumption.

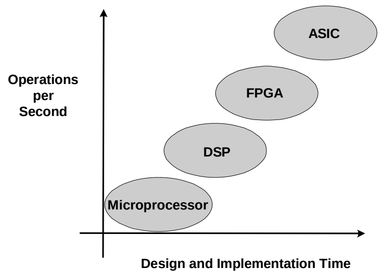

Fig. 4. Illustration of processing power versus design time.

FPGAs offer reconfigurability on the bit-level at the cost of larger chip integration areas and slower maximum clock frequency as compared to custom and semi-custom very large scale integrated (VLSI) processors. FPGAs may compete with or complement microprocessors. Indeed FPGAs are becoming so sophisticated that the majority of certain microprocessor and DSP cores may be implemented in them. Finally, the key performance advantages of FPGAs over software processors are their high computational performance and low power consumption. These are primarily the result of the very high degrees of parallelism and pipelining possible in designs implemented in FPGAs [10].

At the same time, FPGAs cannot yet implement the most powerful and sophisticated microprocessors. In addition the large quantity of developed software and variety of development tools available for such processors offers advantages in using them rather than FPGAs.

The generic reconfigurable processor architecture presented below is motivated by these factors, both positive and negative. The reconfiguration of the architecture in our design has to be performed by a controlling agent which should reside in the software processor given the current state-of-the art FPGA design tools [10]. This software processor also needs computational power to perform other functions (e.g. transmit, receive), and applications that find a logical implementation in software (e.g. data compression).

Accordingly for the network processor outlined above, we have chosen an architecture based on an FPGA and a reduced instruction set computer (RISC) microprocessor that acts as the software processor. The primary application in mind for the network processor is in satellite networks or dense ad hoc wireless sensor networks. Again, the primary motivations have continued to be flexibility and computational power; if less processing power or less flexibility were desired a different architecture would be chosen. Figure 5 illustrates a conceptual framework of the design.

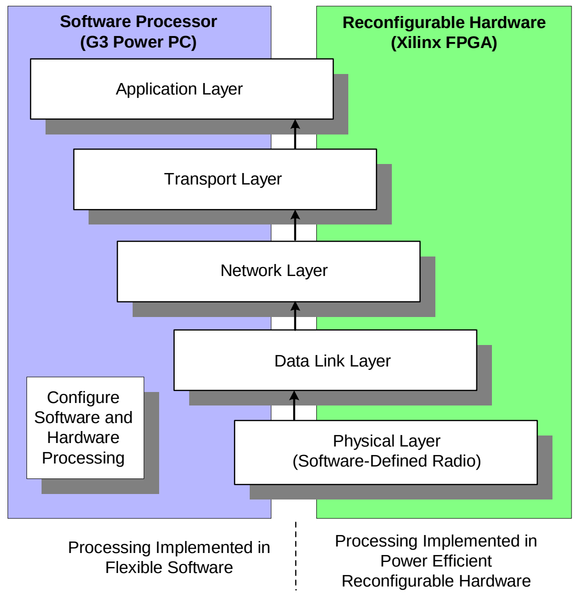
Software Processor (G3 Power PC)
Reconfigurable Hardware (Xilinx FPGA)
Application Layer
Transport Layer
Network Layer
Data Link Layer
Configure Software and Hardware Processing
Physical Layer (Software-Defined Radio)
Processing Implemented in Flexible Software
Processing Implemented in Power Efficient Reconfigurable Hardware

Fig. 5. Network layers implemented in a reconfigurable processor. Note the degree of overlap of various layers within the domain of either the software processor, or the reconfigurable hardware.

Hybrid software and reconfigurable hardware design, as well as reconfiguration of network layers in the processor, calls for a unified object-oriented design paradigm encompassing hardware, software, and the interface between the two. The processor architecture in Fig. 5 was developed to allow the algorithms and signal processing of various layers to be accomplished in a logical way: with some functions logically being implemented in software and some logically in hardware. The goal is to develop design tools that facilitate this logical placement of processing.

The prototype of Fig. 5 assumes a 1 million gate Xilinx FPGA and a 700 MHz PowerPC processor. Note that the reconfiguration of the software and hardware processors is defined by software control.

III. Object Oriented Design and Reconfiguration

The first step in implementing a system design in the architecture shown in Fig. 5 is to represent it in an implementation-independent form. This is done to provide a context in which to evaluate various implementation choices against numerous design constraints. It is crucial that a well-defined route exist from the implementation-independent model to the chosen implementation technology. Additionally it is essential that subsets of the design ("objects"), either hardware, software or a combination – for which partial near real-time reconfiguration is desired – be logically and temporally separable from other parts of the design without interrupting the flow of data processing.

There has been little work on broadly applicable design methods, which allow the design engineer to define such objects in the context of a reconfigurable processor. A primary problem is that although both hardware and software development and implementation are generally well understood as logically separate entities or classes of objects, there are relatively few techniques that apply to the design of systems as a whole, and particularly for the instance when partial reconfiguration with time constraints is desired.

We model the implementation "code" as being in one of three general classes:

1. *Software:* signal processing, networking, data processing, applications, etc; objects in C++ for a microprocessor.
2. *Hardware:* firmware, hardware description language (HDL) code, e.g., Verilog, VHDL, or even C++ for FPGA design.
3. *Interface:* software interface drivers and/or HDL decode/encode interface modules.

A. Software Objects

Objects in a software class are identical to those of traditional object oriented programming. Coad and Yourdon [11-13] present a set of quality design principles based on the following parameters, that result in better, "maintainable" object oriented software designs:

- *Coupling:* Interaction coupling between classes should be kept low by reducing the complexity of message connection and decreasing the number of messages that can be sent and received by an individual object. Inheritance coupling between classes should be high.
- *Cohesion:* A service in a class should carry out one and only one function. The attributes and services should be highly cohesive. A specialization should actually portray a sensible specialization.
- *Clarity of design:* A consistent vocabulary should be used. The names in the model should closely correspond to the names of the concepts being modeled. The responsibilities of a class should be clearly defined and adhered to. The responsibilities of any class should be limited in scope.
- *Generalization-Specialization depth:* It is important not to create specialization classes, which are conceptually not a real specialization, e.g. created for the sake of reuse.
- *Keeping objects and classes simple:* Excessive numbers of attributes in a class should be avoided – an average of one or two attributes for each service in a class is usually all that is required.

B. Hardware Objects

Applying concepts of object-oriented (OO) design to hardware is not new [14]. As integrated circuits continue to increase in complexity, higher levels of abstraction shift from convenience to necessity. Complex circuits increase the demand for the reuse of objects. The reconfigurable hardware class is defined by its attributes and methods. The reconfigurable hard object, an instance of this class, will specify the state (values for these attributes) and behavior (set of methods to be applied).

Traditional HDL (i.e. verilog, VHDL) consists of structural or register-transfer level (RTL) and behavioral constructs. However, many behavioral constructs are not synthesizable and are impractical to use for algorithmic design specification or verification. Object oriented languages like C++, Java, etc. have been used to specify a system at a higher level of abstraction thus increasing productivity, readability, and reusability.

Polymorphism is a major object oriented concept that still poses a problem during synthesis of hardware. Polymorphism allows an entity (variable, function, object) to take a variety of representations. Polymorphism requires that the control flow of an object oriented specification not be explicitly given but results from the values of variables while executing the program. The main synthesis problem consists of the construction of a control flow graph. Refer to [15] for an example solution.

Given the current stage of many synthesis tools, it is convenient to rely on traditional HDL (verilog, VHDL). However, algorithmic design and verification can be done with higher level object oriented design tools or languages like C++, Java, Matlab, Signal Processing WorkSystem (SPW). However this requires an intermediate step of converting the design to HDL for the synthesis tools. Various C++ and Java tools, Matlab and SPW already have this capabiltiy.

Dynamic aspects of OO hardware design are those such as inter-object communication, object creation, adaptation, and destruction. These are distinctly different from corresponding concepts in software OO design. For instance communications between hardware objects are defined more in terms of lower-level mechanisms, rather than high-level primitives. In a similar fashion dynamic creation and use of hardware objects within an FPGA (through computing, communication, scheduling, destruction, ...) frequently happens in a parallel fashion, and must be managed concurrently too. This is unlike the software OO model, and more akin to multitasking within an operating system. Accordingly the FPGA is configured and managed by additional system software (controlling agent) running on one of the system processors. The controlling agent also has the task of facilitating communications between FPGA-based hardware and software objects [16].

C. Interface Objects

The interface class is really a combination of software and hardware processing (classes). Rather than include an entire signal processing module however, we minimize the logical extent of the module. This is because interface objects exist primarily to solve multirate data input/output timing issues and data format conversions that exist on the boundary between hardware and software. An example object could be a data buffer (either in hardware or software) between hardware output software input.

D. Partial Reconfiguration

Software programming and FPGA reconfiguration can occur in three different ways. A *static* download refers to a complete reconfiguration or mode of operation which is determined by the system itself. A *pseudo-static* download refers to the use of over-the-air download to completely reconfigure the system. The controlling agent now determines the mode of operation [3].

The third option, *dynamic* reconfiguration, has the most flexibility, allowing over-the-air partial reconfiguration to occur during a communications link or other signal processing task. There are in turn two forms of dynamic reconfiguration. The first involves dynamically reconfiguring objects not involved with the signal processing task. The second dynamically reconfigures objects that are involved with the current running signal processing task.

Hardware and software objects must be designed and implemented with dynamic partial reconfiguration in mind. Partial reconfiguration of objects will benefit greatly from a streamlined, object oriented, hierarchical design. Strong emphasis should be placed on minimizing object coupling.

By way of an example of the first type of dynamic reconfiguration, consider a multi channel receiver with 4 independent BPSK receivers in an FPGA. In order to partially reconfigure channel one from a BPSK to a QPSK receiver while leaving the other channels intact, channel one must be an independent module with no coupling with the rest of the channels. However, objects or modules within the channel one module will have some degree of coupling. However, since each module has only one function, interchanging modules should be fairly simple. For example, 2nd order loop filter could easily be reconfigured to a 1st order filter, assuming the inputs and outputs are the same (resolution, rate, etc.).

The second form of dynamic partial reconfiguration is more difficult to accomplish. The reconfiguring controlling agent must now generate and transmit a signal to the object to be reconfigured letting the object know that it needs be reconfigured. That object may then have to buffer data (both input data and data it is currently processing) before sending a signal to the controlling agent “ready to reconfigure.” After receiving this signal the controlling agent then initiates the configuration of that object. The reconfiguration time of the FPGA must be known for this buffering scheme to work.

IV. Dynamic Network Reconfiguration

Currently the design tools to accomplish dynamic partial reconfiguration on FPGAs are in relatively early stages of development. However, there can be little doubt that given the current rate of FPGA tool and device development these problems will be solved, and the object-oriented design and reconfiguration methods outlined earlier will find practical and powerful use in dynamic partial reconfiguration of network processors such as that illustrated in Fig. 5.

Dynamic configuration of network processors for wireless communications is very compelling in order to address:

1. Dynamic network conditions such as weather, multipath and fading introduced by transmitter or receiver motion.
2. Operation within networks with heterogeneous nodes and compatibility with other networks.

Figure 6 illustrates a simple example of dynamic reconfiguration necessitated by weather. Reconfiguration is desirable with minimal (ideally zero) interruption in the data link. One or more of the network layers may be reconfigured dynamically. In this instance, the object-oriented design and configuration of the processor is crucial for enabling minimal transmission interruption. As the channel starts to degrade in Fig. 6 objects in the network layers, implemented as depicted in Fig. 5, may be reconfigured according to the level of degradation.

The priorities with which objects are reconfigured for a given network may be predetermined and stored by the reconfiguration agent (in this case residing in the software controlling agent) or may reside in the protocol stack itself. The latter notion gives rise to the notion of network protocols for dynamically reconfiguring network processors which currently do not exist; however the development of such protocols and their use in dynamic networks of the future will likely be required to make the most efficient use of the reconfigurable network processor technologies described here.

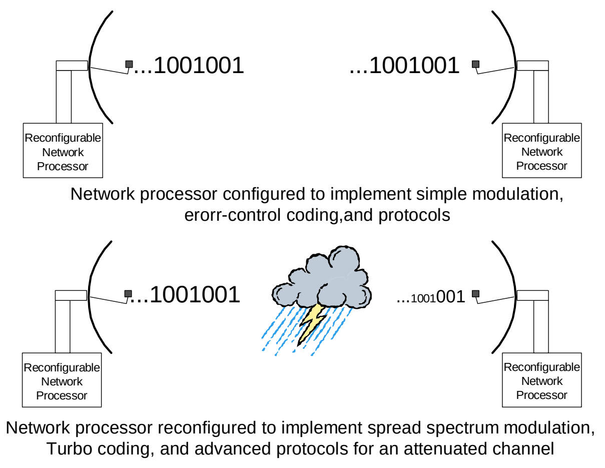
...1001001
...1001001
Reconfigurable Network Processor
Reconfigurable Network Processor
Network processor configured to implement simple modulation, erorr-control coding,and protocols
...1001001
...1001001
Reconfigurable Network Processor
Reconfigurable Network Processor
Network processor reconfigured to implement spread spectrum modulation, Turbo coding, and advanced protocols for an attenuated channel

Fig. 6. An example of a dynamic, network driven reconfiguration: protocols of the future may include built-in dynamic processor reconfiguration features.

V. Laboratory Prototype

A prototype of the reconfigurable processor illustrated in Fig. 5 is currently operating in the laboratory. To date the physical (BPSK radio), data link, and transport layers have been implemented and demonstrated. The processor has been demonstrated in a two-way communications link with data rates as high as 1 Mbps per channel. Examples of NASA science missions planning to use the reconfigurable architecture of Fig. 5 include the *Starlight* mission for Autonomous Formation Flyer in 2006; and the *Neige* experiments on Mars premier orbiter in 2005. The reconfigurable processor may also be appropriate for future mission to Mars as well.

VI. Conclusion

We have provided an overview of software reconfigurable processor technologies and the object-oriented design methods that facilitate efficient reconfiguration, and in particular dynamic partial reconfiguration of these processor technologies. A specific reconfigurable network processor was developed as a manifestation of these technologies and a prototype is currently operating in the laboratory at JPL. Such processors and the object-oriented design methodologies presented here greatly increase the diversity of applications of network processors as compared to traditional processor technologies and can provide long-life satellite communications and dynamic wireless network infrastructure support.

Dynamic reconfiguration enables a multi-mission role for the processor, greatly increasing its value of. Finally, it appears logical that network-driven dynamic reconfiguration of the future will reside in the network protocols themselves; this appears to be an interesting area of future research and development.

References

[1] N.J. Drew and M.M. Dillinger, "Evolution toward reconfigurable user equipment," *IEEE Communications Magazine*, vol. 39, no. 2, Feb. 2001.
[2] E. Buracchini, "The software radio concept," *IEEE Communications Magazine*," vol. 38, no. 9, Sept. 2000.
[3] W.H.W. Tuttlebee, "Software-defined radio: Facets of a developing technology," *IEEE Personal Communications*, vol. 6, no. 2, pp. 38–44, April 1999.
[4] J. Mitola III, "Software radio architecture: A mathematical perspective," *IEEE J. Selected Areas in Communications*, vol. 17, no. 4, pp. 514–538, April 1999.
[5] M.S. Cummings and S. Haruyama, "FPGA in the software radio," *IEEE Communications Magazine*, vol. 37, no. 2, pp. 108–112, Feb. 1999.
[6] A. Stoica, R. Zebulum, D. Keymeulen, R. Tawel, T. Daud, and A. Thakoor, "Reconfigurable VLSI architectures for evolvable hardware: From experimental field programmable transistor arrays to evolution-oriented chips," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 1, pp. 227–232, Feb. 2001.
[7] E. Ramsden, "The ispPAC family of reconfigurable analog circuits," *Proc. of The Third NASA/DoD Workshop on Evolvable Hardware*, pp. 176–181, 2001.
[8] P.P. Gelsinger, "Microprocessors for the new millennium: Challenges, opportunities, and new frontiers," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 2001.
[9] J. Eyre and J. Bier, "The evolution of DSP processors," *IEEE Signal Processing Magazine*, vol. 17, no. 2, pp. 43–51, March 2000.
[10] O. Mencer, M. Platzner, M. Morf, and M. Flynn, "Object-oriented domain specific compilers for programming FPGAs," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 1, February 2001.
[11] P. Coad and E. Yourdon, *Object-Oriented Analysis*, Prentice-Hall, 1991.
[12] P. Coad and E. Yourdon, *Object-Oriented Design*, Prentice-Hall, 1991.
[13] L.C. Briand, C. Bunse, and J.W. Daly, "A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs," *IEEE Trans. on Software Engineering*, vol. 27, no. 6, June 2001.
[14] W. Nebel and G. Schumacher, "Object-oriented hardware modeling - where to apply and what are the object?," *Proc. EuroDAC'96*, pp. 428-433, Geneva, Switzerland, 1996.
[15] T. Kuhn, W. Rosenstie, "Java based object oriented hardware specification and synthesis," *IEEE Design Automation Conference*, pp. 413-418, 2001
[16] P.N. Green, M.D. Edwards, "Object oriented development for reconfigurable embedded systems," *IEE Proceedings on Computer and Digital Technology*, vol. 147, no. 3, May 2000.