Reconfigurable and Evolvable Hardware Fabric

Chris Papachristou, Frank Wolff
Case Western Reserve University
EECS Department
Cleveland, OH 44106
cap2@case.edu

Robert Ewing
AFRL/AFTA, Bldg 620
WPAFB, OH 45433

Abstract. In this work, we are developing a novel reconfigurable multiprocessor architecture, environment and tools for autonomous onboard processing in space platforms. Among the important features of our method are: reconfigurability and processor adaptability for high rate wireless functions; usage of system-on-chip (SOC) technology to embed hardware modules and reconfigurable blocks; integration of real-time Operating system kernels on the SOC, low-power, quality of service, and performance.
Our approach employs a multiprocessor architecture of two basic layers, the adaptation software manager and the reconfigurable hardware fabric. The adaptation manager captures real time inputs from sensors and decides what reconfiguration, if any, needs to be performed and then sends this information to the hardware fabric which performs dynamic reconfiguration. The adaptation manager also involves a software learning process to correct adaptation and reconfiguration decisions. We employ middle level configuration granularity for rapid or dynamic reconfiguration implementing mission critical functions such as signal and imaging functions. Some preliminary results are discussed.

1. Technology Approach

1.1 Background

By definition, a *reconfigurable* system has the ability to modify its structure, behavior or function during the course of its operation. This modification, called *reconfiguration*, can be achieved either on command or autonomously [2]. There are four different classes of reconfigurations.
1. Static Reconfiguration
2. Dynamic Reconfiguration
3. Self Reconfiguration
4. Evolvable Reconfiguration.

Reconfigurable digital logic is currently implemented by FPGAs. Static reconfiguration is achieved by downloading into the FPGA chip a new configuration functionality while the FPGA is off-line, i.e. not operating in normal mode. There is an obvious disadvantage with this off-line technique especially if the reconfiguration time is significant. Many of the currently available FPGAs are still static reconfigurable. Dynamic reconfiguration means to insert a new FPGA functionality on the fly, i.e. while the chip is operating normally. One way to achieve this is by partial reconfiguration, i.e. inserting new configuration while the chip is operating, then swapping the old with the new configurations. A second way for dynamic reconfiguration occurs if the chip modifies its own configuration using internal or peripheral signals for configuration purposes, during normal operation. This techniques leads to self reconfiguration provided the configuration signals are autonomously generated and not fed by commands. Currently, partially reconfigurable FPGAs are now commercially available, e.g. Xilinx Virtex II. However, autonomously reconfigurable and self-reconfigurable FPGAs are not available.
Evolvable reconfiguration [3] implies self-growth and replication of the reconfigurable hardware. Evolvable hardware use bio-inspired approaches and may need other implementation technologies not based on CMOS.

1.2 Issues with FPGA technology

Although FPGAs are flexible, nonetheless, DSPs (digital signal processors) are faster in terms of raw speed. For example, DSPs have faster execution time of arithmetic operations provided their bit-length matches the DSP hardware bit-length. However, DSPs are not as flexible and consume much more power in full operation mode. The key advantage of the FPGAs over DSPs is that they are malleable so they can match applications in terms of overall performance, power consumption and fault tolerance. This flexibility is important for embedded on-board systems.
Nonetheless, current FPGAs have some significant disadvantages concerning granularity and scalability. Although the FPGA chip density has increased significantly, FPGAs still use fine grain configurability at the bit-level. This works well with simple digital functions but cannot scale to handle efficiently the mapping of wireless communications algorithms such as complex IR filtering, advanced imaging, multichannel CDMA. The reason is that implementing complex applications on FPGAs involves inefficiencies in resources, for example we may need to route nearby with far away logic blocks thus incurring real time delays.
Another issue is that current FPGAs are not amenable to autonomous or self reconfiguration. Although partial reconfiguration as in Xilinx Virtex II is far more flexible than static, still it is done at the bit-level on command. Autonomous dynamic reconfiguration requires architecture

scalability using larger configurable blocks and interconnects. Dynamic reconfigurable processing hardware will be a key enabling technology for on-board systems to meet future demands of NASA space missions.

Software process movement is important in real-time systems for fault-tolerance. However, power efficient process movement heuristics have not been investigated in FPGA technology. FPGA-based systems are less energy efficient compared to other architectural alternatives but they offer significant potential for power-reduction techniques since many blocks of the FPGA could be inactive and still consume power.

1.3 Reconfigurable Architecture Proposed

We propose a novel 4-layer reconfigurable multiprocessor system, Fig. 1, which has the following properties: a) consists of configurable hardware operator units as well as embedded processor cores; b) is capable of function and operator parallelism at the processor and hardware layers, respectively; c) scalable and amenable to self reconfiguration; d) suitable for embedding in on-board systems for space applications. A key feature of the proposed processor is that its reconfiguration strategy is based on the interaction and coordination of two basic entities: dynamic reconfigurable hardware and adaptable application software. This feature distinguishes our proposed approach from other research works that also use large grain configuration, [5, 6, 8]. In the following diagram we show the basis of our proposed reconfigurable system architecture, Fig. 1. There are four key layers:

- Layer 1, the dynamic reconfigurable hardware.
- Layer 2, contains the embedded processors and memory modules.
- Layer 3, the real-time operating system kernels (RTOS).
- Layer 4, the adaptation software manager.

Briefly, the adaptation manager separates functions into classes and assigns them for processing at the other layers. Note that the top and bottom layers, reconfigurable hardware and adaptation manager, are fundamental to our overall reconfiguration strategy, whereas the two middle layers are supportive. Although a Platform FPGA, e.g. Virtex II can implement the reconfigurable hardware layer, we will also develop an alternative reconfigurable hardware fabric because it has many advantages for in-space computing. In what follows, we describe the architecture layers of Fig. 1 in more detail.

1.4 Architecture Components

• **Adaptation Manager**.
Our proposed system architecture is autonomous or self reconfigurable. Our approach to autonomous reconfiguration is based on the twofold concept: adaptation of the application software coupled with dynamic reconfiguration of the reconfigurable hardware. This is achieved by careful interaction and coordination of the top and bottom layers of Fig. 1, the adaptation manager and the reconfigurable fabric, respectively. This concept is shown in

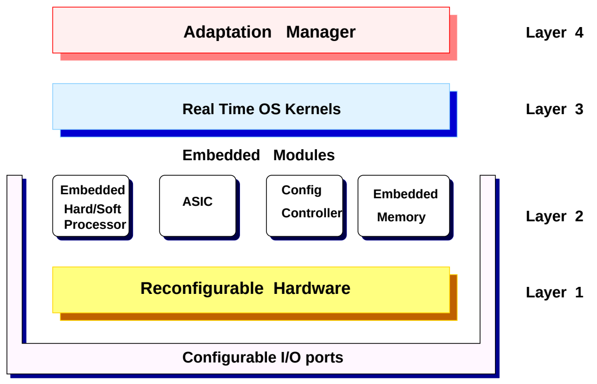

Figure 1. Reconfigurable Architecture

Fig. 2. The adaptation manager captures real time inputs from sensors and interacts with the Function Libraries. This facility will include prebuilt configuration matrices and other pertinent information. The adaptation manager decides what reconfiguration, if any, needs to be performed and then sends this information to the hardware fabric which performs dynamic reconfiguration. The adaptation manager will also involve a evolution learning process to correct adaptation and reconfiguration decisions. We will consider supervisory learning, incremental learning as well as genetic algorithm techniques for the learning process. Design of the adaptation manager is a major task of this research effort.

Another purpose of the adaptation manager is to classify and dynamically separate application function, i.e. data/signal processing and communication function families, that will be mapped for implementation into the other layers of the architecture testbed (Thrust 3 area). One of the questions of this research task that needs to be answered is what is a set of common (universally needed) application functions. Thus one adaptation strategy would be to allocate common baseline functions to the embedded processor modules, Layer 2. However, computation intensive functions (e.g. signal/processing) requiring much parallelism will be allocated to the reconfigurable fabric, Layer 1.

Moreove, we need to collect the application's data concerning power and real-time completion. In a resource allocation scheme, for example, this data can be used to tradeoff power with operation speed. Note that the Function libraries in Fig. 2 should include for each functional module power and real-time configuration profiles. This means, there should be several configuration profiles for each instantiation of a functional module, each being associated with power and real-time attributes. Thus another advantage for using the reconfigurable fabric will be power concumption as under proper low power configuration profile, the fabric consumes less power than the embedded processor(s). Our approach explores many design configuration alternatives, so that additional power knowledge about the computation can be incorporated to refine the configuration selection without requiring large overhead.

• **Real Time OS Kernels**. The OS (Operating System) Layer will actually include two major kernel com-

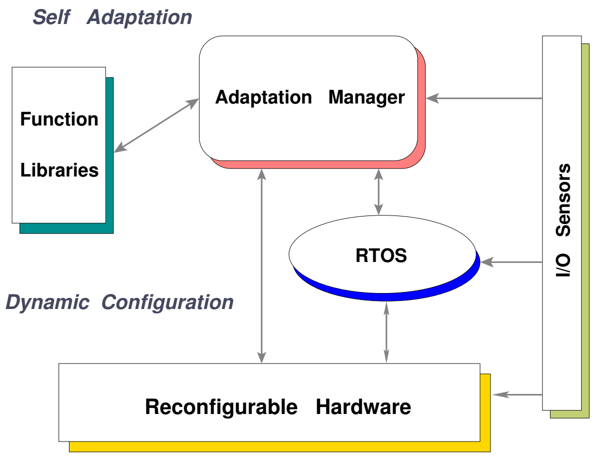

Figure 2. Adaptation and Dynamic Reconfiguration

ponents: a real-time OS kernel, and a configuration OS kernel. In mission operation environment there is need for time critical operations, e.g. real time functions, fast time responses, task scheduling, and more. We intend to employ embedded RTOS kernel(s) to handle time critical operations. The kernel will coordinate activities of the other layers and input/output. Note, standard OS components such as drivers will also be employed in this layer. We will evaluate in terms of performance a number of RTOS kernels available for migration into our architecture. We will also consider Linux-based RTOS because of the open source availability which also makes it easier to interface to the software and hardware architecture layers.

A configuration OS kernel is also needed to control and assist with the mapping and dynamic reconfiguration of the hardware fabric, Fig. 5.Development of the OS configuration kernel involves many novel real time issues and is a major task of this research.

• **Embedded Processor Modules**. This layer integrates embedded processor cores, e.g. DSP cores, and memories. Both hard and soft cores can be employed. It can also include ASIC components, if needed. There are two roles for this layer: a) to support the top two software layers, Fig. 1, i.e. adaptation manager and OS kernels. b) to provide processing capability for baseline functions that do not demand intensive computations. Hard processor cores such as the ARM core, could be used for baseline functions because the latter are quite stable. On the other hand, soft processor cores would be more efficient to handle communication functions because the latter would require some design tuning and optimization. Note that this layer is considered distinct from the reconfigurable hardware in Fig. 1. However, since advanced FPGA platforms (Xilinx, Altera) now include hard and soft processor cores, it is possible to incorporate some part of this layer into such FPGA platforms. We will analyze this issue regarding feasibility and tradeoffs in Phase 1 of our research.

• **Reconfigurable Hardware**. New system on chip (SOC) technology pulls together ASICs, microprocessors and FPGAs into a single polymorphic chip design. SOC technology supports modularity, however, optimizing performance while maintaining low power will depend on the SOC ability for quick or even dynamic reconfiguration. There are two emerging trends for SOC configurability. a) Platform FPGAs which integrate into a large FPGA structure microprocessor cores, ASIC blocks (e.g. multipliers) and memories. b) Platform SOC integrating microprocessor, ASICs and memory cores but also reconfigurable hardware fabrics. Platform FPGAs have the advantage of platform stability but they are tied to the vendor's offerings. It appears that Platform SOC fabrics are more suitable for implementing embedded onboard systems because they are more flexible, potentially can consume less power and are amenable to dynamic or even autonomous reconfiguration.

In our work, we will consider two alternative reconfigurable hardware platforms. The first is the well known Xilinx Virtex II platform FPGA, the second is a new reconfigurable hardware fabric discussed in more detail later.

- *Xilinx Virtex II*. This platform has the following characteristics. a) bit level configuration; b) partial dynamic reconfiguration; c) fine grain logic blocks (CLBs); large grain fixed functional units (Multipliers), and possibly small processor units. The Virtex II provides a good compromise and tradeoff between bit level flexibility and mixed granularity. Thus we can employ the numerous hardwired multiplier blocks in Virtex II to perform in parallel many Multiply-Add operations, which are the core elements of signal processing functions. Interfacing the Virtex II tools with our software adaptation and RTOS components will be a major effort in this approach.

However, Virtex II has several limitations e.g., it is partially dynamic configurability and it has fixed hardwired multipliers. Moreover, Virtex II can not be integrated with other modules in a system on chip (SOC). It has the main advantage that it is a well known and stable reconfigurable platform.

2. Details of Approach

2.1 Evolvable Adaptation

Evolvable Hardware (EVH) [3] refers to hardware that can change its architecture and behavior dynamically and autonomously by interacting with its environment. This interaction can range from reconfigurable, self-reconfigurable to evolvable-nano-electronics. EHW has the potential to be the underlying technology behind the avionics and space infrastructure in the near future. At present, almost all evolvable hardware use an evolutionary algorithm (EA) as their main adaptive mechanism. However, other tools available are genetic algorithms (GA's), neural networks, evolutionary programming and strategies. Evolvable Hardware can be classified into two categories, i.e., extrinsic and intrinsic EHW. Extrinsic EHW simulates evolution by software and only downloads the best configuration to hardware in each generation. Intrinsic EHW simulates evolution directly in its hardware. In what follows we discuss an approach to evolvable hardware based on a reconfigurable platform and neural network training mechanism.

We introduce an evolvable hardware model consisting of two interacting components: a dynamic reconfigurable hardware and a neural network. The idea is to achieve evolution in the hardware by evolving configuration candidates via the neural network and testing them for fitness. This is illustrated in Fig. 3. A best fit configuration is selected and provided to the hardware by the neural network in the form of instructions or configuration code. Thus a best fit configuration will modify and adapt the hardware to best responding to a particular input stimulus or event from the environment.

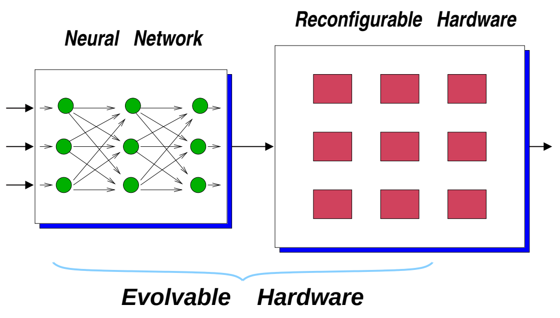

Figure 3. Evolvable Hardware Model

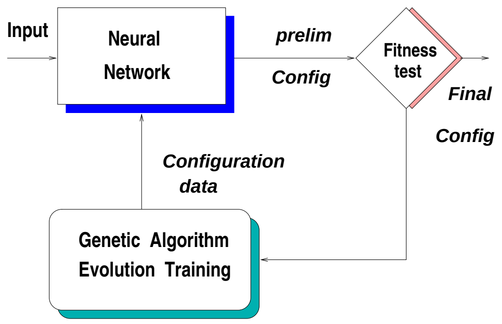

Figure 4. Genetic Training Model

In our scheme of Fig. 3 we envision two modes of evolution.

• **Operation mode**: The neural network recognizes input stimuli and generates configuration code (instructions) for the reconfigurable hardware.

• **Training mode**: The neural network has the capability to adapt by training on the basis of input stimuli and incrementally evolve configuration instructions for the dynamic hardware. This is illustrated in Fig. 4. Other modes such as self-diagnosis, self-repair and self-healing are also feasible.

There are two types of inputs to the neural network in training mode. The stimuli inputs coming from the environment; and the configuration data that are recurrently applied after being modified and improved by genetic operations. Candidate configurations are algorithms or functions that are desired to be implemented in the hardware by dynamic reconfiguration. During training, configurations are selected from a population, i.e. collection of configurations using a genetic algorithm. The training session continues until a candidate configuration passes a fitness test. This test depends on output responses from the hardware.

There are issues regarding the format of the candidate configurations as they are fed into the neural network for training. They should be in a format that the neural network can use for its basic operation. This means to adjust the threshold values of its neurons as well as the weights of the connections, Figs. 3, 4.

Training may start on command, or even autonomously. This may involve training for new environment stimuli, new algorithms, or retraining to upgrade for better performance.

A major aspect of this work in Phase 1 involves the design of a robust training mechanism based on genetic operations for configuration evolution. Further work in Phase 2 would involve the design of a prototype neural network and demonstrate its evolution training capability. For prototyping the reconfigurable hardware, we will use a) the Xilinx Virtex II platform and b) our proposed reconfigurable tile architecture discussed next.

2.2 Reconfigurable Hardware Fabric

The basic idea of the reconfigurable fabric is a distributed set of programmable processing tiles that are capable of instantaneous dynamic reconfigurability, Fig. 5. A *tile* consists of three types of hardware units or resources, i.e. operator unit, local memory, e.g. cache, and local control unit. All hardware resources are connected together through a loop of bus-line interconnects. The operator units are configurable to perform basic arithmetic/logic functions such as Add, Subtract, Multiply. The controller is normally fine grain so it can be implemented using conventional FPGA technology. The tile interconnects are also configurable by means of a switch matrix, e.g. cross bar, which is embedded in the interconnects. A programmable tile goes much beyond the current FPGA technology. A tile achieves a middle grain configuration by efficiently allocating its resources, i.e. operator units and caches as well as their interconnects. Configuration occurs within a tile, and along several tiles which can be interconnected into a reconfigurable fabric.

Basically, tiles are suitable for efficiently implementing application function modules such as FIR filter, FFT, DCT, convolution coder. There are two related problems that need to be addressed in this research: (i) mapping a function module into a tile; (ii) reconfiguring dynamically a tile for two or more functions. For the mapping problem, we propose to use the following 2-stage design process: a) data flow transformation of the function description (e.g. C code) into a resource scheduled graph, b) allocation of data flow elements into the tile hardware resources (operators, cache and interconnects, Fig. 5). The above process is in a way similar to the microarchitecture synthesis process, thus we intend to leverage our extensive work we have done in this area, [9]. Using the above process, we can derive the configuration matrix which is a time vs. resource chart describing the mapping. Clearly, the configuration matrix can be precomputed to be readily available for dynamic loading. Loading the configuration matrix into the tile initiates the application.

For dynamic reconfiguration of the tile we propose to use a dynamic mapping technique, i.e. mapping function B while function A is still working. The steps are: a) Preloading configuration matrix; b) deactivate temporarily idle modules of A to be used for B; c) swapping from A to B. These steps are similar conceptually to partial reconfiguration in FPGAs. However, we make use of the OS configuration kernel, Fig. 1, which controls resources, configuration mapping and swapping.

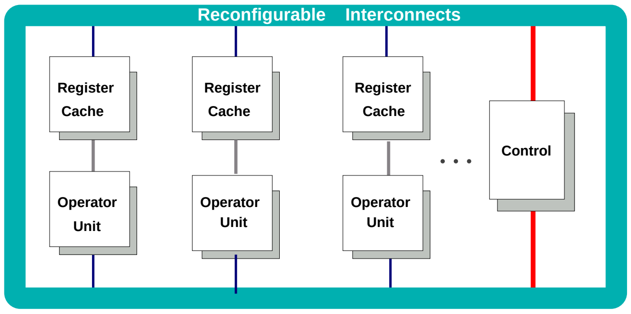

Figure 5. Reconfigurable tile

One important feature that distinguishes our proposed reconfigurable hardware from FPGAs and DSPs concerns the ability to configure the hardware datapath bit-length. We use a bit-slice approach to build flexible bit-length operator units together with their interconnects. Thus applications that demand unusual bit lengths such as 22 bits or 36 bits will be accommodated by dynamically configuring tiles to match these bit lengths. This is a real advantage of our proposed tiles with respect to both FPGAs and DSPs, because one would need a fixed 32-bit DSP to accommodate applications with irregular 22 bit-length, and would be unable to do 36 bits. At the same time, to scale an FPGA to that bit-length may require using far away logic blocks in the chip incurring delay overheads.

The reconfigurable hardware fabric is assembled by hierarchically connecting tiles into a tree structure with the tiles being the leaf nodes of the tree, Fig. 6. This hierarchy provides good scalability of the fabric for expansion. It is important both for mapping and for dynamic reconfiguration of tiles within the fabric. Of course, idle tiles can be turned off to reduce power.

2.3 Dynamic Configuration, Kernels and Fault Tolerance

• **Configuration Matrix**. We use a switch matrix in every tile to interconnect the Operator Units and I/O channels, Fig. 7. Every row is connected to an operator output or input channel, and every column to an operator input or output channel. I/O channels are not shown in detail. The matrix rows and columns are connected only through

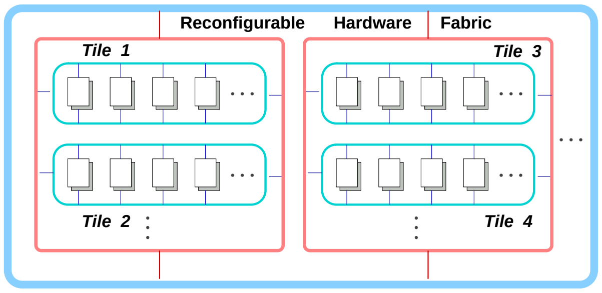

Figure 6. Reconfigurable Fabric

switching elements shown in Fig. 7. Actually each switching element consists of 2 switches and a data buffer in between. This accommodates the READ and WRITE phases of each Operator unit, respectively. More details are in Fig. 8 where the first switch enables a buffer WRITE and the second a buffer READ, respectively. Thus the buffers hold variable values coming from a source operator, stored for the lifespan of the variable, and to be used by the destination operator. A buffer can hold several variable values in sequential order. The role of the switches is to enable WRITE/READ datapath segments. Are there any buffer

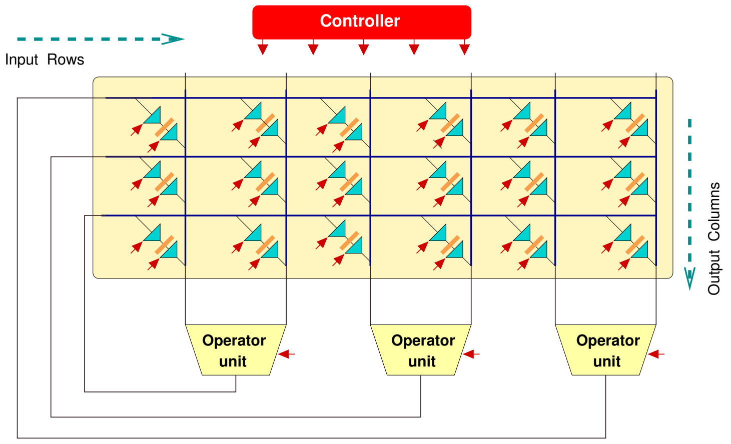

Figure 7. Dynamic Reconfiguration Switch Buffer Matrix

conflicts during the READ or WRITE phases? The switch matrix mechanism in Fig. 7 avoids WRITE conflicts because every WRITE switch in Fig. 7 is associated with a unique Operator output. However, READ buffer conflicts are possible. This occurs when two variables with conflicting lifespans are stored in the same buffer. To avoid such conflicts we have two suggestions. a) scheduling and allocating the application operations under the above lifespan constraint; b) use a prioritized buffer mechanism based on the READ time tag of the stored variables. Buffer overflow needs to be investigated further.

In addition to the data buffer, each WRITE and READ switch has also a control buffer, or FIFO, Fig. 8, which holds WRITE and READ configuration information, respectively. Every FIFO bit corresponds to a control step of the particular application running in the tile. Suppose the FIFO bitstream 110101 is loaded on a WRITE switch, time progressing from left. This means the switch will enable WRITE on its buffer at time steps 1,3,5,and 6. Note that if the Operator units are multifunctional, then additional control FIFOs are needed for each Operator, in like manner to the switch FIFOs of Fig. 8. The proposed scheme allows for variable latency operations, however, latency information should be reflected in the FIFO bitstreams of the switches.

Shown in Fig. 9 is a proposed scenario for dynamic configuration. Assume that an application has been precompiled and loaded into the Configuration memory as a binary. Note that the Configuration memory can be structured into long words each long word corresponding to the entire switch matrix of a tile. Further, each long word is parti-

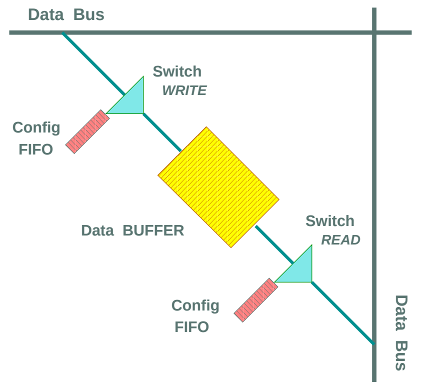

Figure 8. Double Switch Buffer

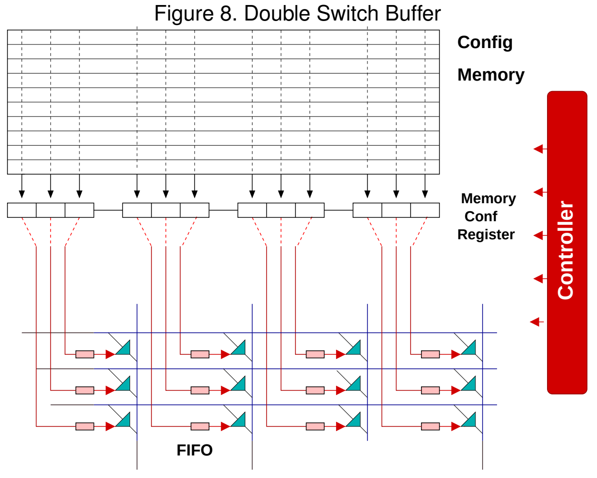

Figure 9. Configuration Memory

tioned into data fields where each field corresponds to a particular switch FIFO of the matrix. Note, if the application demands longer execution time, then we may need several long words in the Configuration memory for the application. Configuring the application on the tile amounts to shifting the data fields into the FIFOs as shown in Fig. 9.

There are few points worth mentioning. a) Applications can be loaded back to back simply by loading their FIFOs back to back. b) It is possible to configure an application X truly dynamically on the tile while application Y is still running. To do that we will need to load X in the tile so that it will not cause resource conflict with Y. c) Relocation of an application is possible at configuration time. This means that the actual tile operator resources for an application can be decided on the fly depending on availability.

• **Configuration Kernel**. The management of our dynamic configuration strategy is implemented by a configuration OS kernel. The description of the kernel can be summarized in the kernel status diagram in Fig. 10. It consists of two parts, active and inactive. The tiles in the hardware fabric are separated by the configuration kernel into two classes *active* and *inactive*. Active tiles carry the applications that currently are running or ready to run. Applications not running are suspended and then either terminated or swapped into an inactive tile. Clearly swapping and terminating free up tile resources, used by the application, for the benefit of another application.

An application is first loaded into the active tile by loading its configuration FIFOs as discussed previously. It then goes through the states READY, RUN, SUSPEND, SWAP or TERMINATE. A decision to terminate rather than swap an application could be made by the based on past history. It would be faster to reactivate an application from an inactive tile rather than reload it from the Configuration memory. In the inactive tile, an application remains idle until the kernel decides to recall it into an active tile. It is also possible to terminate a useless application all together from being on an inactive tile. The scheme in Fig. 10 also allows to preload an application into an inactive tile, a mechanism similar to precaching. This is useful if the application will be used soon but not soon enough to justify loading it into an active tile.

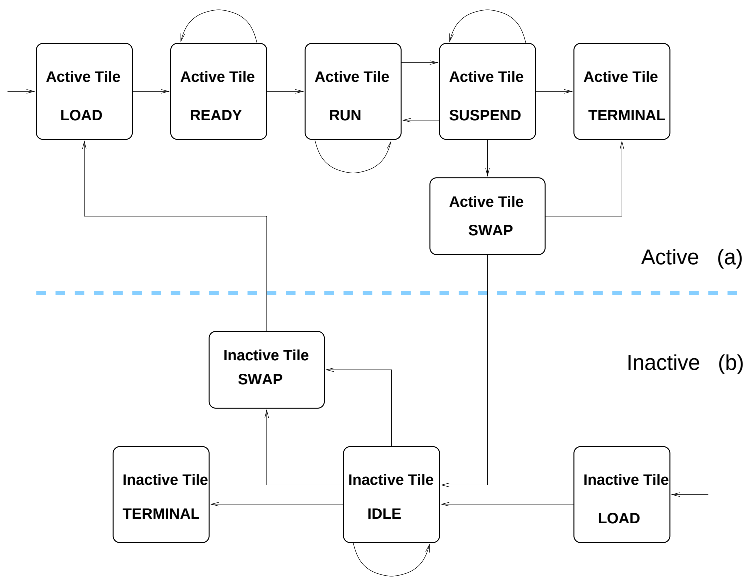

Figure 10. Active and Inactive Tile Status Diagram

• **Fault Tolerance Scheme**. An important advantage of our configuration kernel scheme is its ability to perform fault tolerant operations in the fabric. Specifically, the well known check-point and roll back recovery technique, which is employed in software systems, can be implemented quite efficiently and very fast at the fabric level. Our approach is as follows:

1) an on-line error detection mechanism should be implemented in the fabric. A very suitable scheme could be a sort of software BIST whose functionality could be configured in the fabric. However, other schemes are possible.

2) Once an error is sensed, the configuration kernel will suspend the affected configuration and roll back to a previous check-pointed state of this configuration that would exist in an inactive tile Fig. 10.

3) The kernel will load the roll back configuration and remap it into another area of the active tile which is not affected by the fault.

Note that the status diagram of Fig. 10 will need to include a ROLLBACK state to accommodate this fault tolerance scheme. Also, check points need to be inserted at the applications level, i.e. the libraries of Fig. 2. Check-point placement and recovery policy as it applies to the fabric can be a powerful mechanism to infuse fault-tolerance in the context of the reconfigurable system proposed.

3. Discussion on Challenges

There are several technical challenges facing this project which we define next.

• **Flexibility and Autonomy**. Capability for autonomous dynamic reconfiguration in response to surrounding changes and perform complex processing and communication tradeoffs, in real time, which is not possible under current technologies. Autonomous self reconfiguration is very important is space missions.

• **Reconfigurability**. Ability to modify its structure, behavior or function during the course of its operation. Capability to dynamically change the system structure or functionality on the fly.

• **Modularity**. Ability for integrating advanced modules, configurable blocks, and RTOS kernels into SOC technology.

• **Scalability and Granularity**. Reconfiguration based on variable bit-length operator-level granularity. This has a significant scalability advantage over FPGAs and DSPs.

• **Complexity and Cost**. Ability for feature insertion and upgrades such as algorithms, protocols, coding schemes by reconfiguration. This provides significant reduction in the complexity, size and overall cost over currently used technology.

• **Fault tolerance, self repair**. Using reconfiguration to to achieve fault tolerance and even self repair, which is obviously very important in space missions.

• **Low power consumption**. Efficient reconfiguring its system resources to consume less power than DSPs or FPGAs.

Meeting these challenges is crucial to achieve the project objectives. We consider each one of the challenges as being an attribute to achieve so that the project deliverables should be measured up against these attributes. Some of these attributes such as power, cost and even complexity have directly quantifiable metrics. We will need to establish metrics for the other attributes such as autonomy, reconfigurability, scalability. Our approach will be to first identify which of the above attributes are relevant for each project deliverable, then to use testbench application modules to measure their performance on each deliverable. More details will be given in the Statement of Work.

Another technology metric that we will use is based on the Technology Readiness Level (TRL) that each of our deliverables is supposed to reach. Thus by the completion of each deliverable we will grade its quality with regard to its expected TRL. (In addition to a "Pass" and "Fail" grading we can also provide a point grading system). The project deliverables and their TRLs are described in the Statement of Work.

References

[1] Trimberger M., (ed.), *Field Programmable Gate Array Technology*, Kluwer Acad. Publ., 1994.

[2] Tessier R., Reconfigurable Computing in Digital Signal Processing: A Survey. *Journal of VLSI Signal Processing*, Vol. 28, pp. 7-27, 2001.

[3] Evolvable Hardware, *3rd NASA/DoD Workshop on Evolvable Hardware*, IEEE Computer Society, July 2001.

[4] DeHon A., The Density Advantage of Configurable Computing, *IEEE Computer*, Vol. 33, No. 4, April 2000, pp. 41-49.

[5] Copen S., et all, PipeRench: A Reconfigurable Architecture and Compiler, *IEEE Computer*, Vol. 33, No. 4, April 2000, pp. 70-77.

[6] Salefski B., L. Caglar, Reconfigurable Computing in Wireless, *Design Automation Conference (DAC-2001)*, June 2001.

[7] Lahiri K., et al, LOTTERYBUS: A New High Performance Communication Architecture for System-on-Chip Designs, *Design Automation Conf. (DAC-2001)*, June 2001

[8] Heysters P.M., et al, Mapping of DSP Algorithms on Field Propgrammable Function Arrays, newblock *10th Intern. Workshop on Field Programmable Logic (FPL-2000)*, August 2000.

[9] M. Nourani, C. Papachristou, Stability-Based Algorithms for Scheduling and Allocation in High Level Synthesis of Digital Systems, *IEEE Transactions on VLSI*, Jan. 2001.