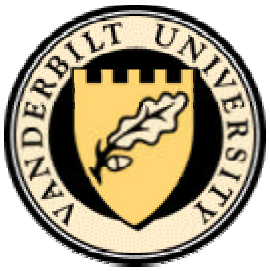
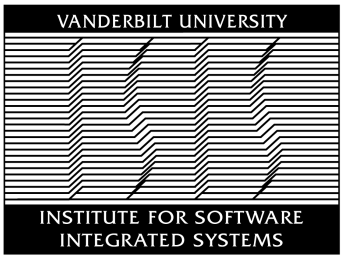




Institute
for
Software-Integrated Systems

Technical Report

TR #: **ISIS-2000-06**

Title: **Hardware/Software Runtime Environment for Dynamically Reconfigurable Systems**

Authors: **Jason Scott, Sandeep Neema, Ted Bapty,**

Abstract

Dynamically reconfigurable architecture computational devices offer a promise of high speed, low cost, and small form-factor by (1) optimizing the architecture for the application, (2) adapting to changing requirements by reallocating hardware, and (3) using low-cost commodity components. These benefits, however, can be lost if the cost of implementing an efficient system is too high, and the ability to migrate to new technology is unsupported.

A set of design and implementation tools, including a run-time architecture is required to meet these goals. The design tools must support high-level specification and synthesis of reconfigurable systems. The run-time environment must serve as a target

KEYWORDS

Reconfigurable Computing, FPGA, HW/SW Co-design, HW/SW Synthesis, FPGA, HW/SW Co-simulation, Dynamic Reconfiguration, Design Environment, Model-Integrated Computing.

ACKNOWLEDGEMENTS

This work has been supported by DARPA/ITO under project **DABT63-97-C-0020**

INTRODUCTION

Dynamically reconfigurable architecture computational devices offer a promise of high speed, low cost, and small form-factor by (1) optimizing the architecture for the application, (2) adapting to changing requirements by reallocating hardware, and (3) using low-cost commodity components. These benefits, however, can be lost if the cost of implementing an efficient system is too high, and the ability to migrate to new technology is unsupported.

A set of design and implementation tools, including a run-time architecture is required to meet these goals. The design tools must support high-level specification and synthesis of reconfigurable systems. The run-time environment must serve as a target for these synthesis tools, enabling execution of the system across a mix of hardware and software (processor) devices. Dynamic architecture reconfiguration must be supported at all levels in the design of the runtime architecture.

This runtime environment and the high-level design environment are being developed as part of the DARPA Adaptive Computing Systems Program. The overall architecture of the design tools is described in the technical report ISIS-99-001. That document describes the *model-integrated* approach to be used in the development of reconfigurable systems. The approach described there divides these issues into several categories: (1) Representation and Capture of design information in terms of *Models*; (2) Analysis of the models for design/requirements/resource trade-off studies; (3) Synthesis of architectures and executable systems directly from the models; and (4) Runtime support environments to support efficient execution of the synthesized reconfigurable systems.

The Model-Integrated Computing (MIC) approach has been successfully applied to a diverse set of applications ([4][5][6][7][8][9]). The general MIC approach involves creating a development environment that is customized for a specific application domain. The resultant development environment is a *multiple-aspect* graphical editor that directly supports the engineering concepts required in the development process. Where several engineering disciplines are involved in system development (e.g. Software, Hardware, DSP algorithms, Systems Requirement Specification, etc.), the multiple-aspect nature of the approach allows different aspects to be customized for individual disciplines. The graphical editor allows construction of system Models, which capture the specifications and components required along with their relationships. The Models form a database of design information that can then be used in system analysis, trade-off studies, and performance estimation/simulation. These same Models are used to synthesize the executing systems. The synthesis process assumes a runtime environment that hides the low-level hardware/software details from the synthesis process..

This paper describes the details and design decisions in building the reconfigurable run-

time environment. We will describe the design motivation and goals, target architectures, and implementation strategies.

DESIGN FACTORS AND GOALS

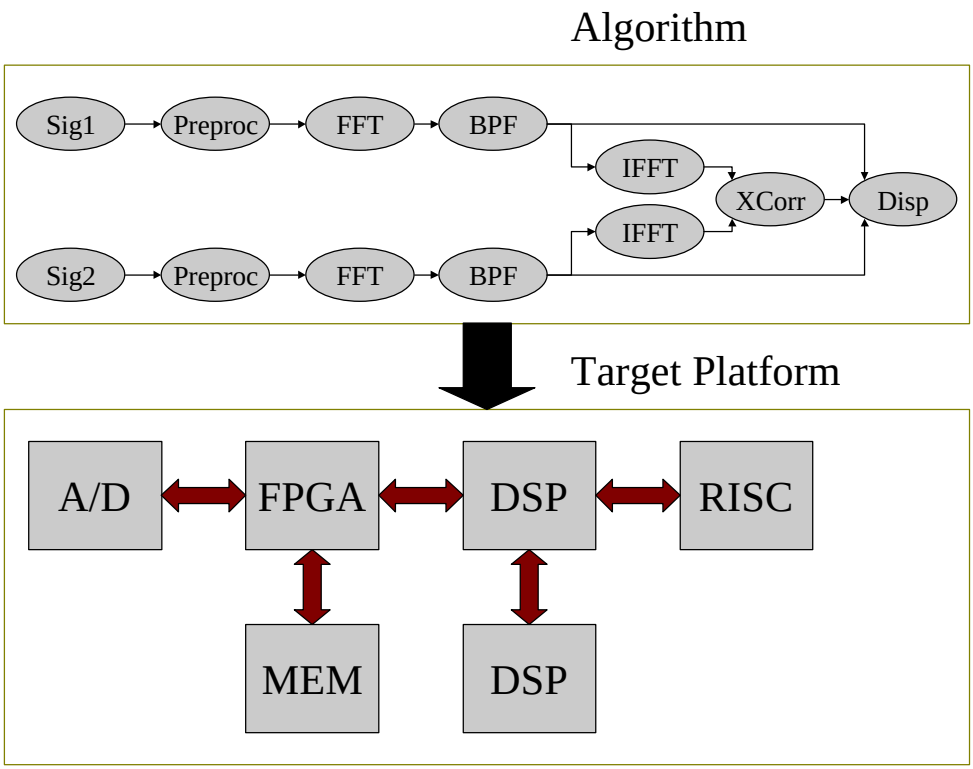
The runtime environment for a Model-Integrated Computing design environment involves a careful analysis of the needs of the design engineers, the methods and components used in the designs, and the target systems. This section will describe the concepts developed in the creation of the Adaptive Computing Systems MIC runtime environment.

To clarify matters, we start with what the environment is not intended to do:

1. It is not intended to be an end-user programmable environment. The runtime is intended to serve as a target for the synthesis tools. In this capacity, it is not necessary to incorporate the high-level functions that one could find in a standard real-time operating system such as VxWorks. Instead, it should be RISC-like in nature, since synthesis tools rarely use high-level functions.
2. It is not intended to be a write-once, run-everywhere system. Performance

The Adaptive Computing Systems (ACS) runtime environment has the following design drivers:

1. **Hardware & Software:** Target applications will be composed of functions that execute on conventional programmable processors (DSP's, RISC, CISC) and functions that are directly implemented in hardware. These devices can be FPGA-based, where the logic is reconfigurable at runtime or directly implemented in silicon (Hard cores/ASIC's).
2. **Flexible Topology**: The connection topology must be flexible to match application data flow patterns. Pipelineing and parallelization can be used in different configurations at different times within an application's lifetime.
3. **Dynamic Reconfiguration**: The topology and the functions of the computational elements must be able to be changed at specified times within the application lifetime. The extent of the reconfiguration is defined by the capabilities of the hardware and the requirements of the application.
4. **Performance**: The runtime environment must impose a very small overhead performance penalty.
5. **Real-Time**:Timing behavior must be guaranteed.
6. **Data-Flow Based**: The primary target applications are embedded signal processing systems. This paradigm supports these very well, but is non-intuitive for control-based applications.
7. **Portable**: The tools must support technology migration. The target architectures must be able to track technology, as DSP's evolve and FPGA's get bigger. This does not mean that the same binary applications will directly port, however the design tools must be able to synthesize a compatible application on new hardware.



This figure represents the types of computations that execute on the target architectures. Algorithms are specified as a data flow diagram. These algorithms are mapped to heterogeneous architectures, containing a mix of implementation technologies that is optimized to the algorithm and its performance requirements.

RUNTIME ENVIRONMENT SEMANTICS

The semantics of the execution environment implement a large-grain-dataflow architecture:

1. The Worker Function captures the tasks that are performed by the system.
2. Communication nodes capture the transfer of data between workers.

Computations can be described as a bipartite graph, where workers connect to Comm nodes, and Comm nodes connect to workers. At this level, there are no implied semantics of the workers. The execution properties of workers (Data tokens produced/consumed per execution, timing of execution, etc) are maintained at a higher level. The semantics of the Comm units are asynchronous queues.

When the generic large-grain dataflow graphs are implemented, they must be mapped down to a physical implementation. The implementation takes the form of either software or hardware. Software workers execute on a DSP or CPU, which we term Processes. Hardware workers are either implemented in reconfigurable hardware (FPGA's), ASIC implementations, or combinations of both. Processes and Processors are logically equivalent, representing functions on data. Processes/Processors are

connected via logical Comm that must buffer, communicate, and match data formats. In software implementations, the Comm object is implemented by the OS/Kernel as a Stream, a software queue in memory. In hardware, the Comm object is implemented with registers and/or FIFO, or simply wires (Figure RE7).

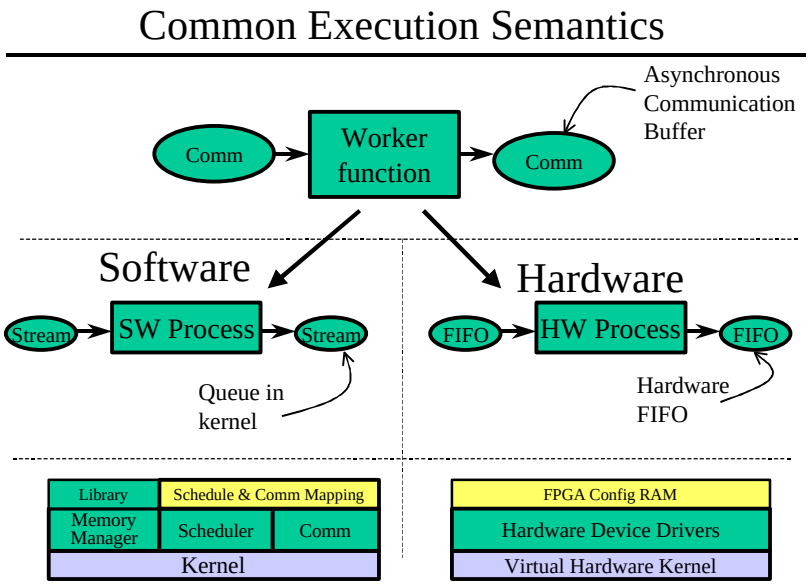


Figure RE7: Runtime Execution Environment: Common Execution Semantics

The execution environment spans software and reconfigurable hardware. The software environment consists of a simple, portable real-time kernel with a run-time-configurable process schedules, communication schedule, and memory management [14]. Communications interfaces are supported within the kernel, making cross-processor connections invisible. Memory management is integrated with the scheduler and communication subsystems, enabling (but not solving) the problems associated with dynamic reconfiguration. The kernel allows dynamic editing of the process table, and of the communications maps. The proper sequencing of these operations, including task execution phases, is necessary for the avoidance of reconfiguration problems. The current approach supports the "Reboot" approach directly, and will support the more advanced reconfiguration approaches with cooperation of the application tasks.

SOFTWARE IMPLEMENTATION

The microkernel has been developed to require a minimal memory footprint, with minimal overhead. Simplicity in development, porting, and debugging have been principal factors in its design. To maintain this simplicity, we have chosen to implement a non-preemptive scheduler. All multitasking is cooperative, with the exception of communications functions. Communications are DMA and interrupt driven, to maintain

maximal overlapping of communications and computation.

The kernel is responsible for three main functions:
1. Scheduling of computations.
2. Communications within processors and between processors.
3. Memory management.

The tight integration of these functions is vital to performance. One major factor in performance is communications and memory management. Communication of a buffer must not mandate a buffer copy. The kernel has been designed to obviate this need.

These three functions of the kernel will be discussed in the following sections.

Scheduling

The kernel itself imposes no scheduling policy. A scheduling module is added to perform these functions. In theory, any non-preemptive scheduling algorithm can be added. In our applications, we have used a simple round-robin policy.

A Data-flow execution semantics is enforced in cooperation with the processes. Under the simplified round-robin scheduler, each task is given a chance to execute at its specified order in the list. The process decides that based upon its triggering rules and the configuration of data on its inputs. Therefore, tasks must obey their own data-flow semantics.

The primary object within the scheduler is the Process. The process structure is shown below.

```
typedef struct _process_struct
{
        long status;
        PROCESS_FUNCP funcp;    /* function pointer */
        long num_inputs;        /* inputs */
        long *inputs;           /* list of input streams */
        long num_outputs;       /* outputs */
        long *outputs;          /* list of output streams */
        long local_mem;         /* holds pointer to local memory */
} PROCESS;
```

The system controller can install and remove tasks using kernel system-call functions:
1. **enqueue_process(int process_id)** : add a process to the schedule queue
2. **dequeue_process(int process_id)** : remove a process from the schedule queue

3. **schedule()** : execute the current process script step to the next process in the schedule queue

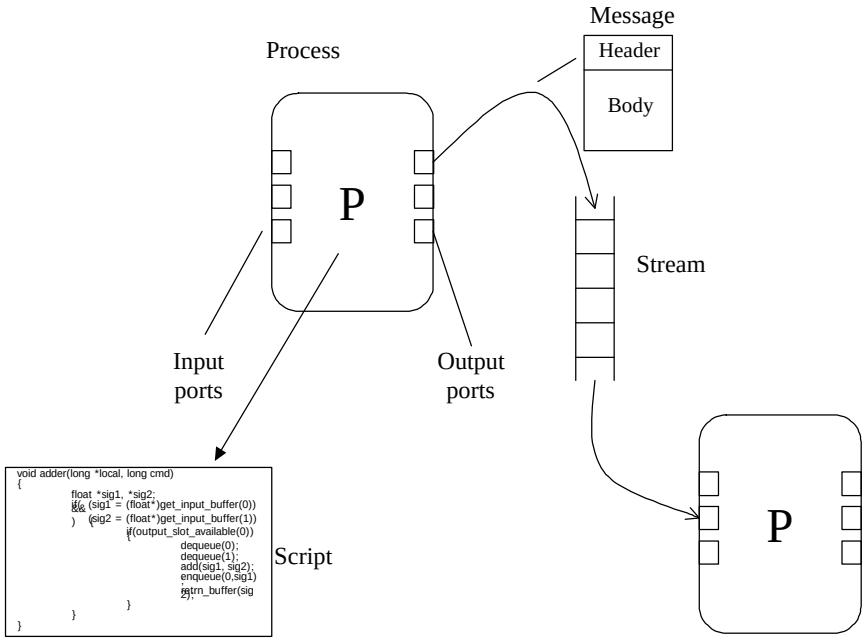
Processes are called by the kernel with a pointer to their context and a scheduling command. The context pointer (long *local) can be used to store a pointer to a context block that is allocated by the process. The command (long cmd) is used to send a reconfiguration message to the process, informing it to compute a safe state prior to being suspended, moved, or deleted. An example of a process is shown below.

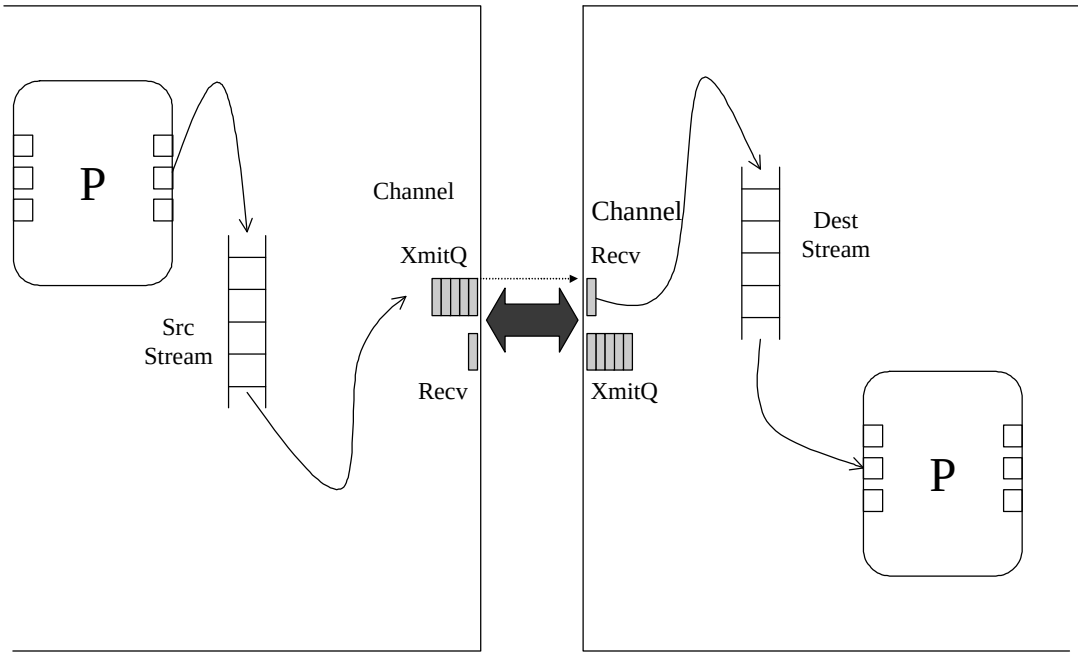
```
void adder(long *local, long cmd)
{
        float *sig1, *sig2;
        if(  (sig1 = (float*)get_input_buffer(0))  &&
             (sig2 = (float*)get_input_buffer(1))  )    {
                if(output_slot_available(0))   {
                        dequeue(0);
                        dequeue(1);
                        add(sig1, sig2);
                        enqueue(0,sig1);
                        return_buffer(sig2);
                }
        }
}
```

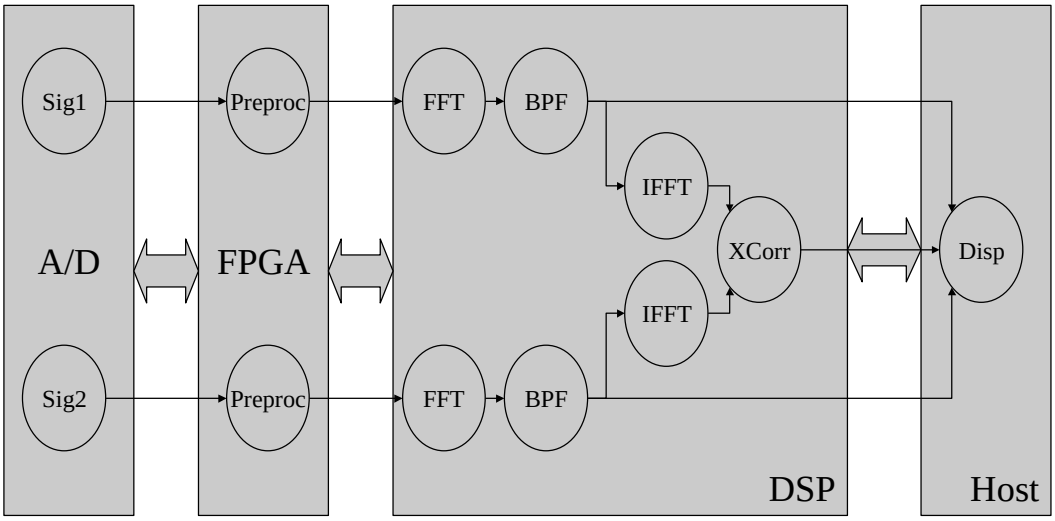
Communication Structure

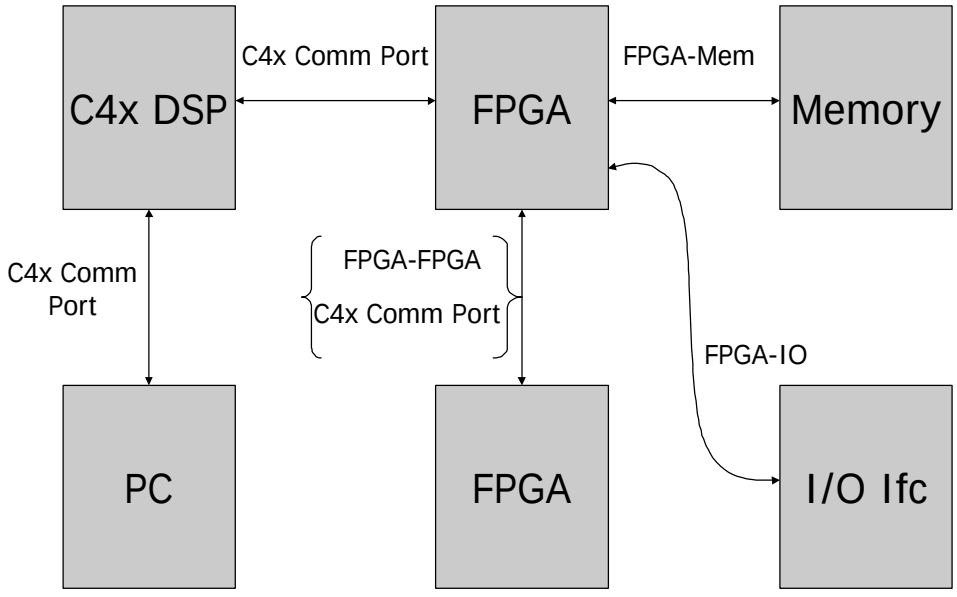
Communications

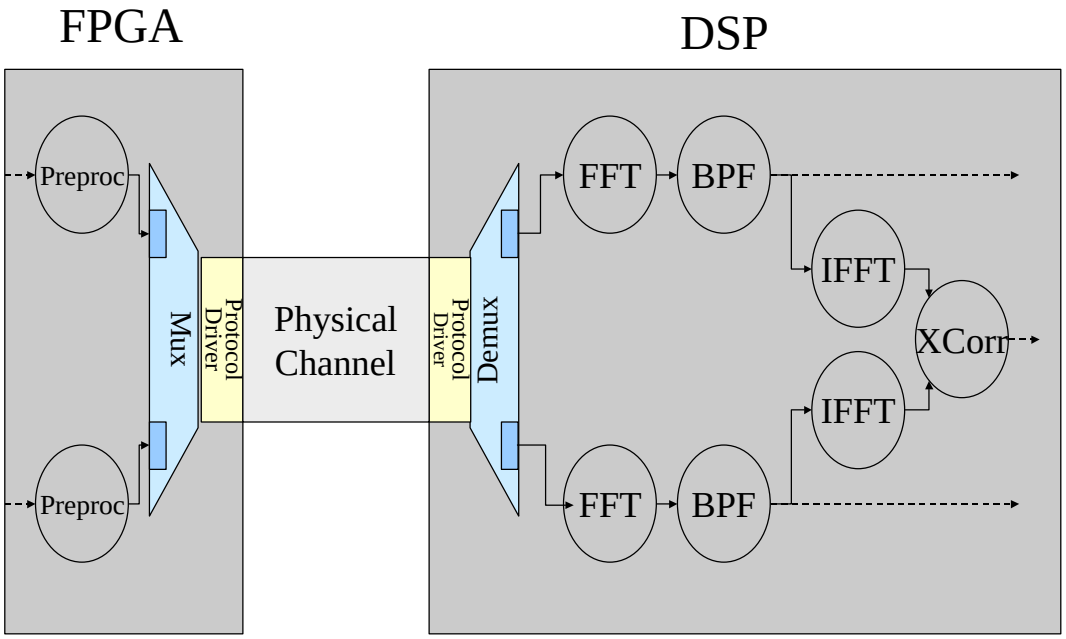
Memory Management

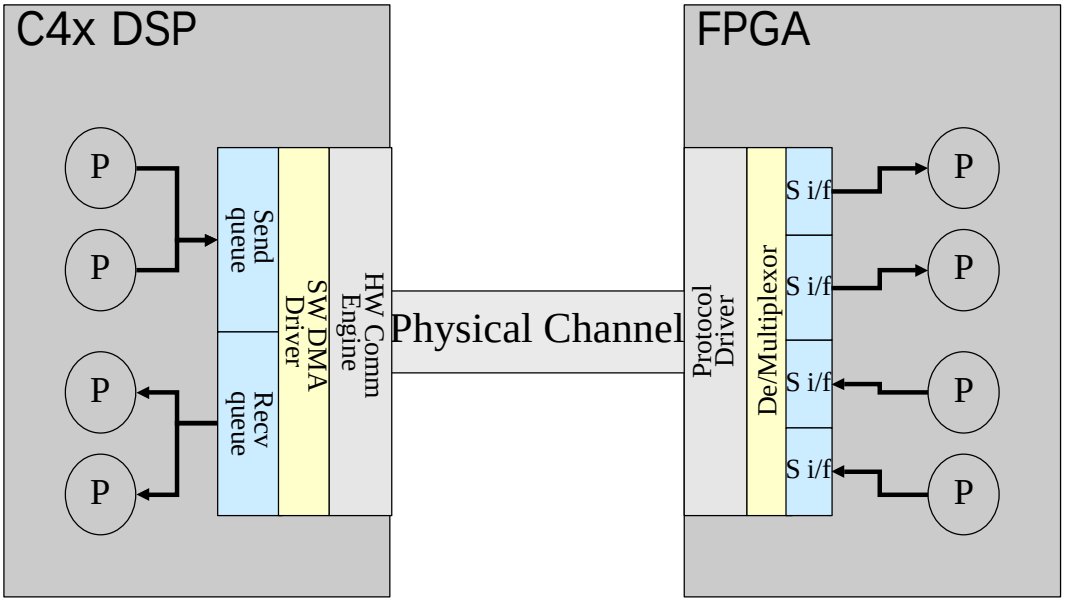

Message
Header
Body
Process
P
Stream
Input
ports
Output
ports
P
Script
void adder(long *local, long cmd)
{
float *sig1, *sig2;
if((sig1 = (float*)get_input_buffer(0))
&& (sig2 = (float*)get_input_buffer(1))
) {
if(output_slot_available(0))
{
dequeue(0);
dequeue(1);
add(sig1, sig2);
enqueue(0,sig1)
return_buffer(sig
2);
}
}
}


P
Channel
Channel
Dest
Stream
XmitQ
Recv
Src
Stream
Recv
XmitQ
P

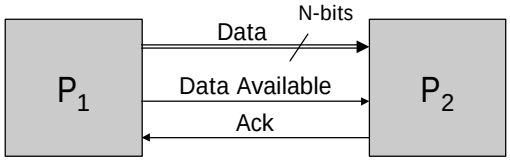

Sig1
Preproc
FFT
BPF
IFFT
XCorr
Disp
A/D
FPGA
IFFT
Sig2
Preproc
FFT
BPF
DSP
Host


C4x DSP
C4x Comm Port
FPGA
FPGA-Mem
Memory
C4x Comm Port
FPGA-FPGA
C4x Comm Port
FPGA-IO
PC
FPGA
I/O Ifc

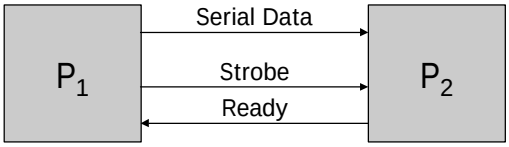

FPGA
DSP
Preproc
Preproc
Mux
Protocol Driver
Physical Channel
Protocol Driver
Demux
FFT
BPF
IFFT
XCorr
IFFT
FFT
BPF


C4x DSP
FPGA
P
P
P
P
Send queue
Recv queue
SW DMA Driver
HW Comm Engine
Physical Channel
Protocol Driver
De/Multiplexor
S i/f
S i/f
S i/f
S i/f
P
P
P
P

Hardware Process Flow Control



Parallel data transfer



Serial data transfer

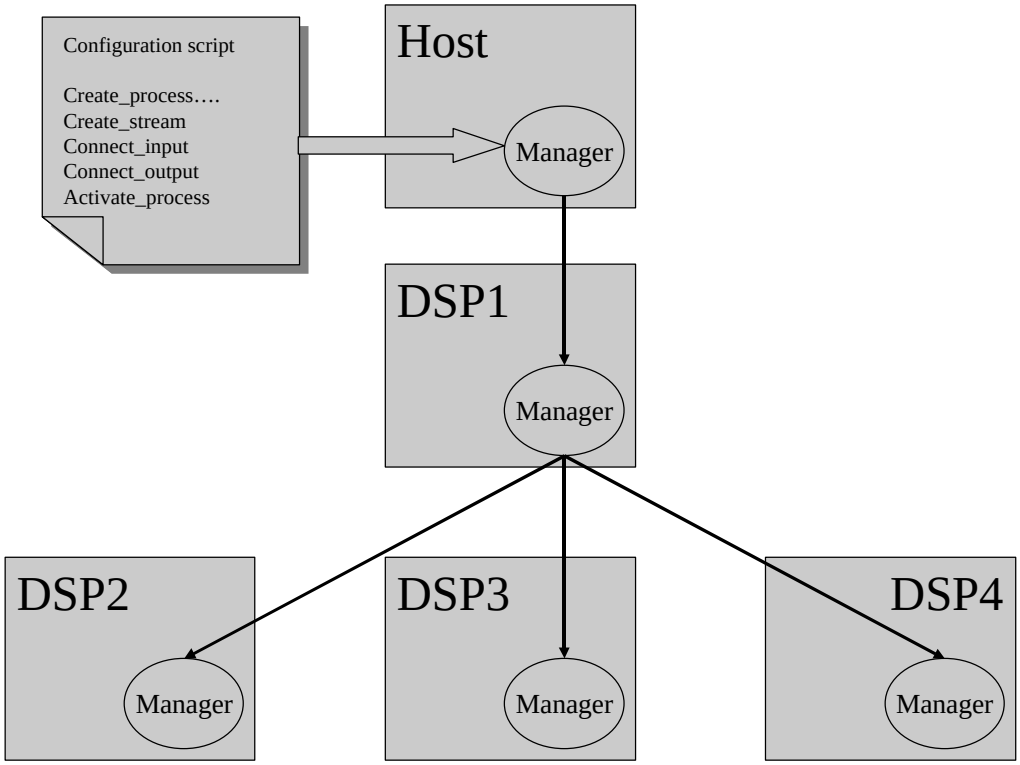
The hardware execution environment supports the same operational semantics. The implementation, however, is much different. The Virtual Hardware Kernel exists as a concept used in the system synthesis. The MIC Generator synthesizes a set of VHDL structural codes, one for each configurable device multiplied by the number of operational modes. Processors are directly synthesized using predefined components. Communications elements are selected from a library of interface types, based on the requirements of the workers on either end, the required performance, and the available resources. The communication infrastructure works in cooperation with the software communications, performing the signal buffering, and the necessary off-chip interfaces and data converters. The interface components are drawn from a library of modules. The modules implement a limited set of standardized communications protocols to transfer data between modules, and present data in the format required by the destination processor. As the system is used for more applications, the set of interface types will grow in capability.

Inherent in these interface components must be the capability to reconfigure. This involves strict synchronization mechanisms, methods for saving and restoring states, and facilities to allow function and structure modification. Global system synchronization is

greatly aided by having a common system clock, and facilities for very low-latency signaling within the system. Our current concepts for reconfiguration require a single interrupt signal to be present at each component participating in a reconfiguration.

In addition, the runtime environment must be designed with an interface suitable for synthesis from a MIC-Generator approach. The properties of the runtime environment must be tuned to simplify the generator. This demands a simple, uniform interface with a well-defined, consistent set of semantics that apply throughout the system.

Reconfiguration Manager



The reconfigurable hardware interfaces, and the flexible microkernel provide the facilities to implement system reconfiguration, however the problem of control and synchronization is critical. A global view of the system is necessary. Reconfiguration cannot be performed by the kernel alone.

This synchronization and control of a system during reconfiguration is the responsibility of the Configuration Manager. The CM contains tables capturing the behavioral state machine defined by the designers Behavioral Models. Tied to these state-based descriptions is the information necessary to configure the hardware and software components of the system.

Given this information, the Configuration Manager serves as a system observer. The CM

monitors relevant signals, as defined in the transitions leading out of the current state. When the logical conditions for a state transition are satisfied, the Configuration Manager begins the structural transition process.

The first stage of the reconfiguration involves bring the system into a known, safe state. All communication interfaces must terminate. Since many of the data ports are bi-directional, the bus token must be returned to the 'safe' state. Computations must be completed and transitioned into the 'safe' state. The safe state may involve using local algorithms to perform the basic required functions to keep the system stable.

After all necessary components are in the safe state, the global interrupt is toggled to initiate the reconfiguration event. At this point, all communications must stop for the short period required for reloading the FPGA's bitfiles and the Software schedules and communication mappings. Since the state of the system was in a known safe state prior to reconfiguration enactment, there it little overhead atop the basic information download. The CM will reload the necessary FPGA's using the standard download methods. A sequence of commands is sent to each of the processors to enact the new processing graph and interface components. Once the new programming information is installed, the system interrupt signal is toggled to ensure a globally synchronized start up operation.

CONCLUSIONS

1. .

REFERENCES

[1] Villasenor, J., Mangione—Smith, W., “Configurable Computing”, Scientific American, June, 1997.

[2] Arnold, J., Buell, D., Davis, E., “Splash 2”, Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures, June 1992

[3] David R. Martinez, “Real-time Embedded Signal Processing”, IEEE Signal Processing Magazine, September 1998.

[4] Bapty, T., Ledeczi, A., Davis, J., Abbott, B., Hayes, T., Tibbals, T.: "Turbine Engine Diagnostics Using a Parallel Signal Processor", Joint Technology Showcase on Integrated Monitoring, Diagnostics, and Failure Prevention, Mobile, AL, 1996.

[5] Karsai G., Sztipanovits J., Padalkar S., DeCaria F.: "Model-embedded On-line Problem Solving Environment for Chemical Engineering", Proceedings of the International Conference on Engineering of Complex Computer Systems, Ft. Lauderdale, Florida, Nov. 6-10, 1995

[6] Long E., Misra A., Sztipanovits J.: "Saturn Site Production Flow (SSPF): Accomplishments and Challenges", Proceedings of the Engineering of Computer Based Systems, Maale Hachamisha, Israel, AL, March, 1998.

[7] Davis, J., Scott, J., Sztipanovits, J., Karsai, G., Martinez, M.: "Integrated Analysis Environment for High Impact Systems," Proceedings of the Engineering of Computer Based Systems, Jerusalem, Israel, April, 1998.

[8] Bapty T., Sztipanovits J.: "Model-Based Engineering of Large-Scale Real-Time Systems", Proceedings of the the Engineering of Computer Based Systems (ECBS) Conference, Montery, CA, March, 1997

[9] Carnes J. R., Misra A.: "Model-Integrated Toolset for Fault Detection, Isolation and Recovery (FDIR)", Proceedings of the International Conference and Workshop on Engineering of Computer Based Systems, Friedrichshafen, Germany, AL, March 11-15, 1996

[10] Harel, D., “StateCharts: A visual Formalism for Complex Systems”, Science of Computer Programming 8, pp 231-278, 1987

[11] Bryant, R.E., “Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams”, Technical Report CMU-CS-92-160, School of Computer Science, Carnegie Mellon University, June 1992

[12] Bryant, R.E., “Graph-based Algorithms for Boolean Function Manipulation”, IEEE Transactions on Computers, C35(8), 1986

[13] Kumar, S., F. Rose, "Integrated Simulation of Performance Models and Behavioral Models," Proceedings of the Fall 1996 VIUF, pp 185-194, Durham, NC, October, 1996

[14] Bapty T., Abbott B.: "Portable Kernel for High-Level Synthesis of Complex DSP-Systems", Proceedings of the the International Conference on Signal Processing Applications and Technology, Boston, MA, May, 1995

[15] Sandeep Neema: “Constraint based System Synthesis”, Technical Report, Department of Electrical and Computer Engineering, Vanderbilt University, 1999.

[16] Hein, C. and D. Nasoff, "VHDL-based Performance Modeling and Virtual Prototyping", Proceedings of the 2nd Annual RASSP Conference, Arlington, VA, July 1995.
[17] James Rowson, "Hardware/Software cosimulation", Proceedings of the 31st Design Automation Conference, pages 439-440, San Diego, CA, June 1994.
[18] Russel Klein, "Miami: A Hardware-Software cosimulation Environment", Proceedings of the 7th IEEE International Workshop on Rapid Systems Prototyping, June 1996.