ISTITUTO NAZIONALE DI FISICA NUCLEARE

Sezione di Roma 1

INFN/TC-98/33
23 Novembre 1998

The APEmille Collaboration:

THE APEOS PROJECT: ARCHITECTURE OF THE APEMILLE OPERATING SYSTEM

Published by ***SIS–Pubblicazioni***
dei Laboratori Nazionali di Frascati

INFN - Istituto Nazionale di Fisica Nucleare
Sezione di Roma 1

INFN/TC-98/33
23 Novembre 1998

THE APEOS PROJECT: ARCHITECTURE OF THE APEMILLE OPERATING SYSTEM

The APEmille Collaboration

INFN, Sez. ROMA I - P.le A. Moro, 5 - 00185 Roma (Italy)
INFN, Sez. PISA - v. Livornese, 1291 – 56010 S. Piero a Grado (Italy)
DESY, Platanenalle 6, 15636 Zeuthen (Germany)

Abstract

This paper describes the APEmille Operating System, developed in the APEmille experiment. APEmille OS is written in C++ and has a strong layered structure. This paper is a review of the overall structure. It attempts to be a self-consistent document.

The first part contains a brief introduction to the APEmille hardware characteristics. In the second part we give an overview of the layer structure of the OS and a detailed description of the architecture including the features of the single operating system modules.

The APEmille operating system described in this paper is already working using our hardware simulation environment.

The hardware drivers are under development as well as the complete set of System Services, and user oriented tools such as the Symbolic Debugger.

1 - OVERVIEW

This section introduces some basic concepts about the APEmille project.
In the first subsection we briefly describe the APEmille architecture mainly as far as it concerns the Operating System.
In the second we introduce our hardware set-up and the software framework, namely the PCI standard and the Linux OS, that are used in APEmille.
In the third we give some details on the APEmille board. We describe a system made of several clusters of APEmille boards and the networked PCI based system.

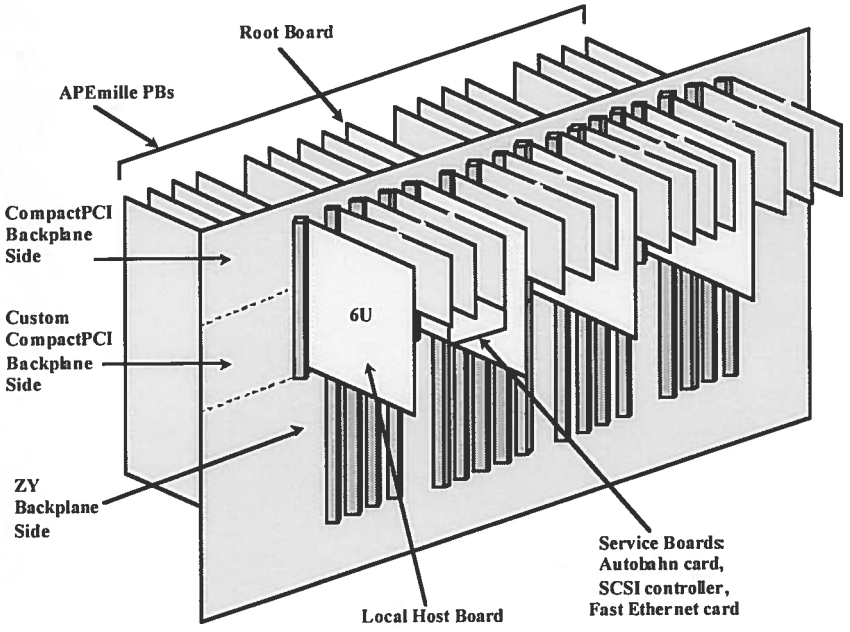

Figure 1: the APEmille crate

1.1 - APEMILLE

APEmille is a massively parallel, SIMAMD[1] supercomputer. In fact, APEmille is a complex system whose structure results from the connection of processing boards grouped in so-called APEmille Units.
The interconnection between processing boards is guaranteed by a synchronous network managed by the communication processors on APEmille **PB**'s. Physically an APEmille Unit is a module made up of:

- Four **Processing Boards** – namely **PB**'s.

[1] Single Instruction, Multiple independent Address generation and Multiple Data.

- One commercial PC (**Local Host** or **LH**) based on Compact PCI standard boards, which acts as a local supervisor. This **LH** accommodates the local part of the APEmille operating system.
- A custom backplane, which hosts four **PB**'s on one side and one Local Host board on the other.

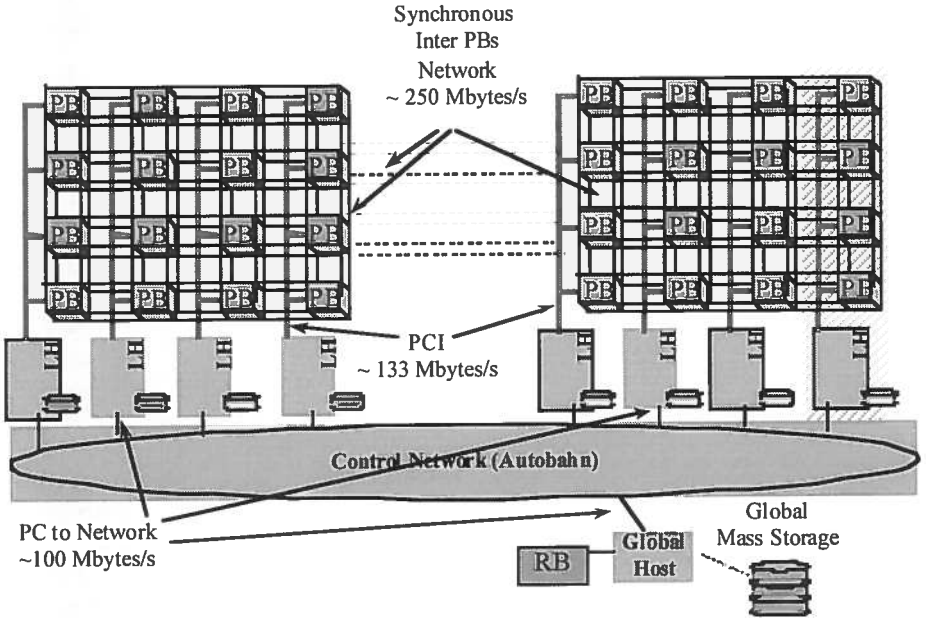

Figure 2: APEmille networks

An APEcrate – see Figure 1 – is a configuration made up of four APEmille units housed in a standard rack.

A multi-unit machine is an aggregate of crates. The **PB**'s of such an aggregate are connected in several different ways:

1. The **Synchronous Network**, which links all the **PB**'s.
2. A very low latency synchronous control network manages halt requests, exceptions, if-conditions, etc. coming from all the components of the machine, and it delivers the obtained global signals back to each of them (Global signals are not reported in Figure 1). This control network is physically based on a hardware component named **Root Board** (**RB**).
3. A high performance, general purpose, Asynchronous Control Network links all **Local Hosts**. It can be either a Fast Ethernet network or our custom **Fast Serial Network.**

The **RB** is seen as a slave of a special PC belonging to the asynchronous network that acts as the Global Host (**GH**) for the entire APEmille machine. The **GH** manages the global disk storage. The Control Network physically connects it to all the **LH**'s.

1.2 - THE (COMPACT) PCI STANDARD

The PCI standard is a specification which covers both hardware and software details of a high performance, low overhead, local bus[1] architecture.
From the hardware point of view, it is a 33 MHz, 32 bit wide bus with a 133 MByte/s peak transfer rate. It supports both CPU driven and Direct Memory Access (DMA) transfers, allows interrupt request (IRQ) routing to standard CPU interrupt facility and reserving of address space regions for I/O purpose. On the system software side, the standard covers a BIOS[2] extension (BIOS32) to implement Plug'n'Play (PNP) capability.
The Compact PCI specification is electrically a superset of desktop PCI with a different physical form factor. CompactPCI utilizes the Eurocard form factor popularized by the VME bus. It is particularly suitable for industrial embedded environments.

1.3 - LH SYSTEM CONFIGURATION AND LINUX OS

Each **Local Host** includes suitable disk storage, disk controllers and networking cards; in the following we will introduce the **Fast Serial Network**. From the **LH** point of view, the four **PB's** are plain Compact PCI boards. We chose to support this piece of hardware under Linux OS.
The Linux OS is a public domain implementation of UNIX standard specs. It is made up of a monolithic, POSIX-compliant kernel and a complete set of standard UNIX-like programs and utilities.
The advantages of such an OS over commercial UNIX flavours are large installed base, high interest from Internet developer community and multi-platform support[3], — even more important in our case — source code availability.
From the functional point of view, it supports key services as:

- Uniform access to hardware devices from user programs by a well-suited device driver architecture.
- Multi-user, multitasking operations.
- Process isolation and protection.
- Fast networking.

---

[1] By definition, a local bus is a bus tightly linked to CPU address/data path.
[2] The ROM code that is executed at bootstrap time, which checks the hardware state and then loads the operating system. It allows boot time card configuration and access to standard PCI registers.
[3] Although the original development took place on Intel-based PC, today Linux OS supports all major CPU architectures such as Alpha, PowerPC, 680x0, Mips and Sparc.

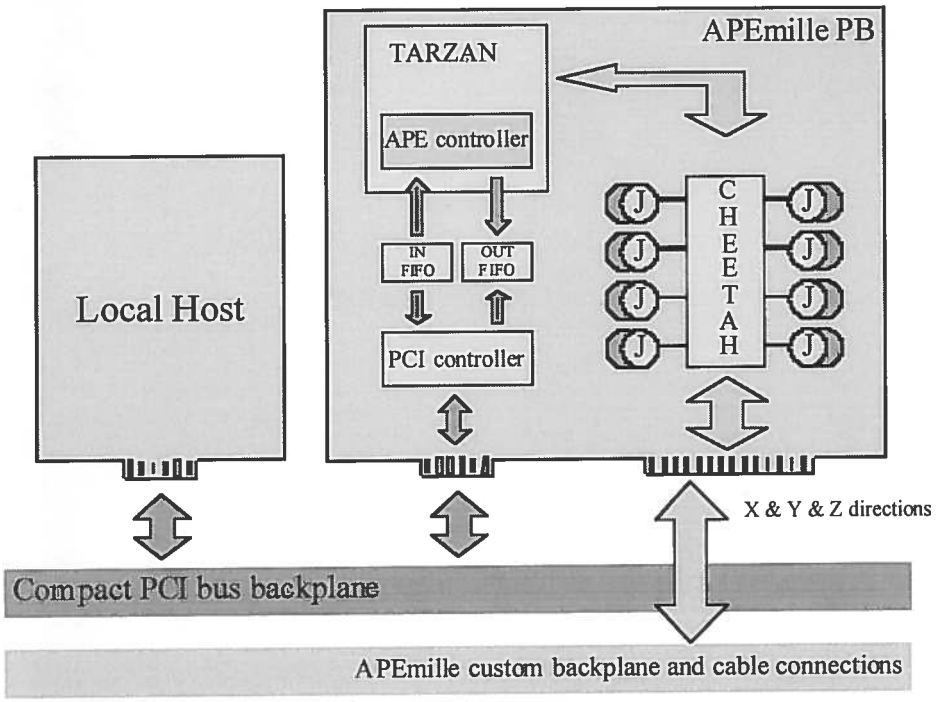

Figure 3: the APEmille PB

1.4 - THE APEMILLE PROCESSING BOARD

The APEmille **PB** contains:

- Eight processing nodes (**Jane**) with local data memories.
- One control processor (**Tarzan**) with its data memory.
- One synchronous communication controller (**Cheetah**).
- A common program memory addressed by **Tarzan**.
- A PCI interface.

A **PB** – see Figure 3 – can be physically connected to:

- A Compact PCI bus, as a slave of the **Local Host**.
- 6 other **PB's** (along the X^+, X^-, Y^+,Y^-, Z^+ & Z^- directions data paths), to create a three dimensional mesh of nodes, providing a sustained inter-node bandwidth of 266 MByte/s.
- The Root Board (**RB**), to transmit local flags and to receive both the global status and the global clock.

A PB is able to generate interrupt requests for the LH CPU in many situations – such as transfer operation completed, HALT request asserted, exception generated and so on. - By writing the appropriate register, it is possible to mask some or all IRQ's.

1.5 - THE APEMILLE CONTROL NETWORK

While Fast Ethernet networking is supported, a faster custom network is under development. The custom network will be based on a high speed (100 MByte/s), **Fast Serial Link**.
A custom board with 4 serial connections will be inserted in the **Local Host** side of the PCI backplane. This custom network organises the Hosts as a 2D mesh. On each column and each row a token ring is implemented.
The serial interface implements a low latency serial protocol.
The bandwidth on each token ring is ~ 100 MByte/s.

2 - THE APEMILLE OS

2.1 - OVERVIEW

The APEmille OS is a substantial improvement of the old APE100 Operating System architecture. It has a multi-layer, object oriented structure, mainly based on the C++ language. Given the structure of the hardware, it has also to be a distributed OS; in fact, it partly resides on the **GH**, which controls the entire machine, and partly on the **LH's**.
In the following, we sum up the main requirements as:

- We have to provide a high level interface to the APEmille services, such as loading and execution of a program, I/O toward APEmille devices, switching between run and system mode, exception handling, error detection and recovery.
- We have to support the System Service model needed by the APEmille programs, i.e. user interfaces, I/O toward both local and global storage, pseudo-random number generation, etc.
- As APEmille contains at least 8 nodes and up to 2048, a specific set of system services is required to give the user a friendly parallel I/O interface.
- For the Multi Unit (Multi Local Host) machine, we need to support the synchronous and the asynchronous network, providing optimised access serialisation to both global and parallel resources: graphics or *tty* user console, global disk, parallel disk server, pseudo-random number generation of seeds, synchronous start/halt of the processors, etc.

As higher level development issues, there are:

- Use of the simulator as a replacement of the real hardware that is still under development.
- Support for different programming paradigms (SIMD, SIMAMD).
- Dynamic loading of user-designed OS modules to implement special system services.
- Machine partitioning.

- Integration with the UNIX programming environment through shared memories etc., useful to leverage the transition to APEmille environment by integration with traditional workstation sequential code.

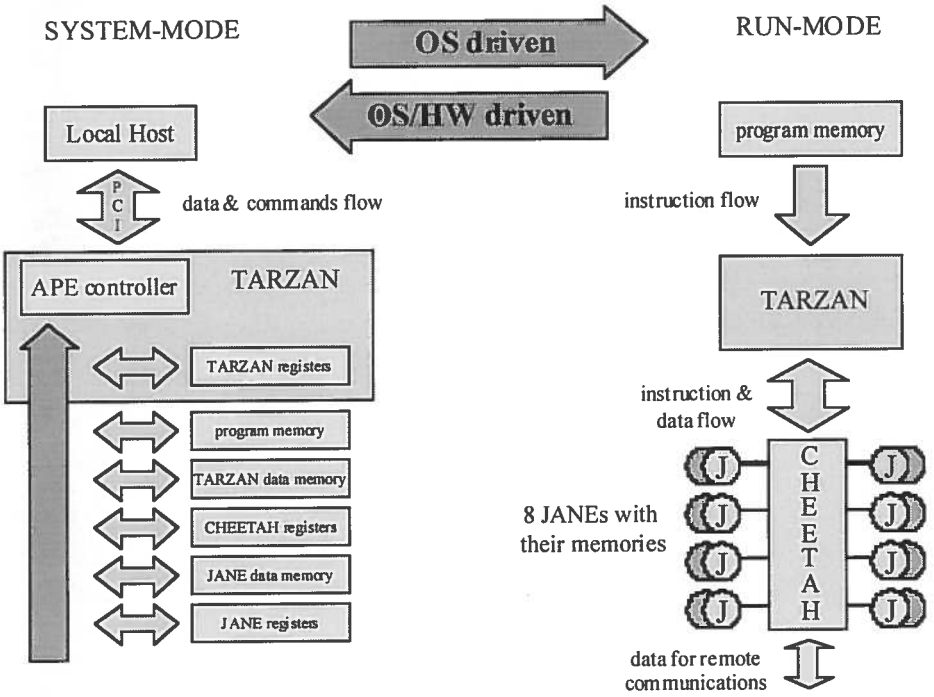

Figure 4: APEmille modes

2.2 - RUN MODE/SYSTEM MODE AND SYSTEM SERVICES

APEmille processing boards have strong computing power, but essentially no facilities to support the Operating System. Indeed, we leave to the local and global hosts all the management duties. On purpose, the **PB** leaves the interrupt management and I/O handling to the **LH**, which accesses and fully controls the **PB**'s across the above-described PCI interface.
We define two basic operation modes: *run-mode* (**RM**) and *system-mode* (**SM**) – see Figure 4, – with possible transition acted both by software and hardware.
The *run-mode* is the state in which the hardware executes programs. The *system-mode* is the state in which the processing hardware is idle and the **LH** has full control over memories and state/configuration registers of the different devices.
When the machine is powered up, it is in *system-mode*; then the OS loads the program code and data and initialises the appropriate registers. Then the OS generates a transition into *run-mode* and waits for requests coming from the program.
Program code can trigger a transition from **RM** to **SM** by executing a dedicated assembler instruction, a HALT. All software-generated transitions to system mode are called *traps*.
Hardware can also trigger such a transition when an anomalous condition happens and an exception occurs.

The HALT instruction can be viewed as the *trap* instruction of standard commercial microprocessors where it manages the transition to kernel-mode. In APEmille the analogue of the kernel-mode is the **SM**.
When a program needs a certain service, such as I/O toward a storage device or from/to the screen, it writes in standard location a special data block which describes the required service. Then it HALTs the machine. The **LH** decodes the data block and performs the request. In case of exceptions, each **LH** reads **PB**'s status registers, reports the kind of problem and calls the exception handler possibly defined in the running program.

2.3 - APEMILLE OS LAYERS

We can describe the APEmille OS as a set of layers.
The main principles behind this architecture are:

- Clear definition of services offered by each layer. It is required that APEmille OS uses a uniform, one forever, solid set of interfaces.
- Different programs, such as a debugger or a hardware machine test, will use those same interfaces.
- Managing by object encapsulation and inheritance.

In Figure 5, the APEmille OS structure is outlined at the logical level. Each horizontal band represents a code layer that exports one or more interfaces to upper layers. A code layer is made up of modules. Sometimes a module is a single C++ class; other times an aggregation of classes.
Each layer offers services to the upper layer and automatically performs actions when certain events happen on the lower layer.

LAYERS DESCRIPTION

In our guided tour we begin the description from the lowest level — the Linux device driver and the simulator — and go on with higher-level software layers up to the Master layer and the user interfaces.

PB's layer

This layer offers the basic functions of the machine. The services offered are rather simple and near to the hardware functionality. Both the upper layer and low-level hardware test programs access APEunit functionality through this layer. The **PB**'s layer monitors the state of all the four **PB**'s and *dispatches* events — such as HALT requests, exceptions — to the upper layers.

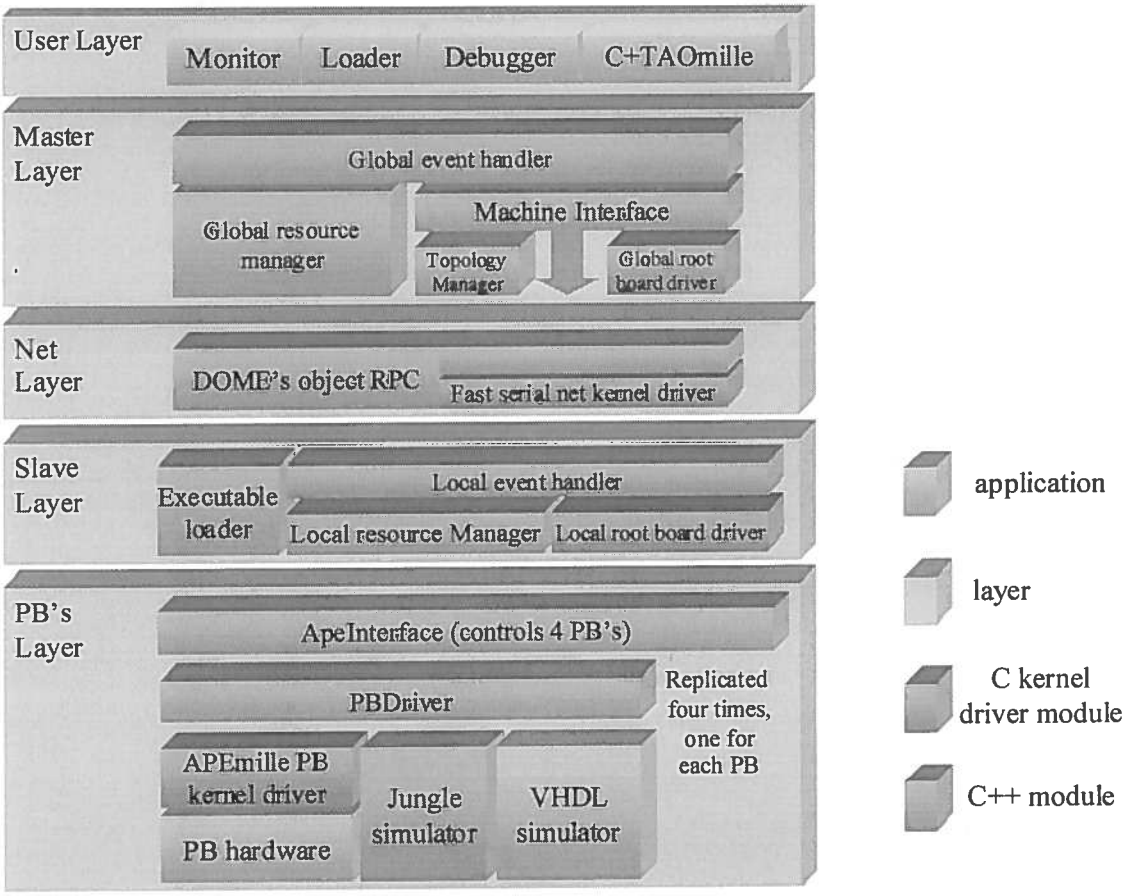

Figure 5: APEmille OS layers

Slave Layer

This layer offers to the upper layer a friendly interface toward machine resources.
As an example this layer can:

- Load an entire program on the appropriate memories.
- Enable or disable some options such as exceptions handling
- Manage the transition for the execution of a program by writing into the appropriate registers.

This layer handles the program traps, decodes System Service requests and is able to complete local request — such as I/O toward local disk storage. — Otherwise it dispatches the request to the upper layers.

Net Layer

It offers a homogeneous interface to different configurations of the parallel machine; it completely masks the details of managing objects living on different Local Hosts preserving the C++ syntax of object method invocation. At its core, the Net Layer fully exploits our APE Distributed Object Management System (**DOME**) technology — see below. —
This layer makes possible to use several different HW options:

- Single Unit vs. Multi Unit HW configurations.

- Commercial (i.e. fast Ethernet) vs. APE **Fast Serial Link** networking.

For Single Unit machines, it adopts standard, efficient C++ method invocation.
For Multi Unit machines, **DOME** automatically generates proxy/stub code.
The code automatically switches to the fastest network subsystem available.

Master Layer

This is the upper layer of the operating system. It offers complex primitives like run of an application, etc.
At this level it is possible to examine a variable by name or to set a breakpoint.
It manages all kind of services such as reading from user console and writing to global disk storage.

User interface Layer

This layer contains all the end-user programs that use the underlying layers and offer user-friendly shells to the operating system.

MODULES DESCRIPTION

In the following we describe each module of the APEmille OS. Modules are grouped in layers. The first two modules (physical hardware and simulator) are shortly reviewed here only to give a consistent view of the OS.

The **PB**'s Layer

This layer deals with the interaction of the APEmille hardware present on each **APEunit**. This code is physically replicated over each **LH**'s. It interfaces to the real hardware as well as to our simulation environment. This environment is made of the APEmille behavioural simulator — **Jungle** — and of the commercial VHDL simulator.

The **Jungle** Simulator

It is a complete behavioural simulator of APEmille hardware; this package is code-named **Jungle**[1] and is built as an independent program, written in C++. This program easily simulates from one to many processing boards and is capable of distributing its workload on different workstations to achieve better performance.
In other words **Jungle** substitutes the real machine allowing early tests of both software (APEOS, compilation chain) and hardware systems (test vectors production and comparison with VHDL results). In this way we solve the problem of an early integration between hardware and software.
Jungle simulates both APEmille operation modes. In *system-mode*, it offers an interface to other programs - using standard inter-process communication methods - by which it can be driven; in *run-mode*, it is capable of simulating the APEmille synchronous network – i.e. inter-node communications - using standard network facilities to connect to Jungle programs running on other workstations. As such, it can be distributed over a network of workstations to achieve better performance or to simulate larger machines.

The **PB** kernel driver

As we have already stressed, the **PB** board is a fully compliant PCI device; it supports all the features of a high performance card. So, we have developed the know-how to support those features in a kernel level device driver for Linux OS.

By this mean, we have reached two key purposes:

- Use of the standard API functions to manage system resources, such as memory, IRQ's, I/O address space. In this way, all the benefits coming from future release of Linux OS will be readily supported.
- Integration in the standard device driver structure; the device is identified by a special file like `/dev/apepb0`. Standard C library functions such as `open`, `close`, `ioctl()` can be used to access the driver from user programs.

To support different programming paradigms, the driver allows two different transfer modes, synchronous and asynchronous; the switch between them is done by a proper `ioctl()` call. In the former mode, all `read` and `write` operations are blocking calls, so that the caller process is kept in sleeping state till the end of the operations. In the latter one, I/O calls return as soon as possible and the process can either *poll* the transfer status using an `ioctl()` call or be signalled by the operation completion.

The PBDriver

This class is an abstract access point — a so-called C++ virtual base class — toward the APEmille **PB**.

The `PBDriver`'s abstract interface offers:

- Methods to read/write **Jane/Tarzan** data/program memories.
- Methods to read/write **Cheetah** configuration registers.
- Activation of performance preserving transfer methods - broadcasting or data-vault. –Using these features, it is possible to merge accesses to different homogeneous devices using single PCI transactions.
- Managing of transition from system to run-mode.

From this abstract class, three classes are derived through C++ inheritance mechanism:

- One that interfaces to the real **PB**'s HW through our kernel driver.
- One that interfaces to our **Jungle** HW simulator written in C++.
- Another one that remotely connects to our VHDL simulators running on SUN's workstations.

In other words, the `PBDriver` class offers a uniform interface toward both the real and the simulated hardware.

As both **Jungle** and the VHDL environment simulate all the hardware up to, but not including, the PCI interface, a code module has been developed that attaches the APEmille OS to a process executing the Jungle simulator.

The `PBDriver` module retains the kind of access typical of low level device drivers. A further module, namely `ApeInterface`, offers user friendly methods such as `WriteJnMem()` and `ReadTzPrgMem()`, which map to several calls to `PBDriver` methods. It is more device-oriented and eases the access to APEmille devices. Furthermore, it automatically chooses the best transfer method supported by a particular device.

The ApeInterface class

This C++ class offers three kinds of methods:

- Reading/ Writing of all the available devices of the four **PB**'s, such as program/data memories and *system-mode* registers.
- A `select()`-style method that waits for events coming from all the **PB**'s and wakes up the process.
- An exception handling method which decodes exceptions and cleans APEmille **Processing Board** internal state.

For each **Processing Board**, the `ApeInterface` class contains a `PBDriver` object. I/O operations on groups of APEmille devices are resolved with iterations over that set of objects.

The SLAVE Layer

The interface offered by this layer is a really complex one. Simple calls are sufficient to upload an APEmille program on the **PB**'s, to start/stop the execution, to single step. It is meant to be remotely accessed through the net layer – see below. – It contains two independent modules.

The Executable Loader

This is the real APEmille executable (JEX) loader. It receives the binary directly from the Global Host and parses it into records. There are different kinds of record; usually, they are labelled after the devices they will be loaded onto. There are special records to support the *Special System Service* mechanism — see below. —

The Local Event Handler

The **Local Event Handler** (**LEH**) deals with transitions from run-mode to system-mode. When such a transition happens, **LEH** decodes it and distinguishes between system service requests and abnormal exception conditions.
If it is a system service of a local kind — i.e. access to a local file — it manages to execute it all by itself; otherwise it waits for orders coming from the Global Host.
Exceptions are treated differently depending on user settings — software traps, masks, etc.—

The NET Layer

The aim of this layer is to manage the slave hosts, to dispatch commands, to transfer data between the master layer and the slave layer using the available network interface. This layer achieves its goal using DOME[2], a distributed object management system, that is based on Remote Procedure Call (RPC) concepts and is able to execute procedures on the slaves, even on all of them at the same time using low overhead, broadcasting techniques. Using DOME a user program can create objects in other computers, so it is possible to write a multi-crate operating system using remote objects as if they were local objects.

APE Distributed Object Management Environment

DOME allows the distribution of objects in a way that is transparent to the details needed to pass data through a communication network.

Its major advantage with respect to other commercial solutions, such as OMG CORBA or MS DCOM, is proper optimisation for the typical situation found in APEmille. Typically in our case the traffic pattern is a one-to-many communication from the **GH** to the **LH**'s to deliver *orders* and a subsequent many-to-one *collecting* of the results. Without proper handling, this event chain can generate both a bandwidth and a latency bottleneck in the network. In fact, using point-to-point connection from **GH** to each **LH** involves multiple transmission of the same information; then simultaneous access to the network by all the **LH**'s at the same time causes network contention — generating collisions and retransmissions. —

Using DOME, the user must only *describe* the interface of a C++ object that she/he wants to be remotely callable. Using that description, *proxy/stub* classes are automatically generated. The description consists of:

- The name to be assigned to the *proxy* object.
- The name of its C++ correspondent, that is, the original class to be remotely callable.
- Public methods prototype with additional information regarding the way they are used[1].

A suitable parser reads this description and automatically produces the C++ *proxy* code — to be linked to the user program — and the *stub* code — to be linked to the server code.—

Two method invocation modes are available:

- Unicast mode, by which a call to a proxy method generates a call to a remote method.
- Multicast mode, by which a single call to a set of proxy objects generates a call to all of them, using the bandwidth saving broadcasting feature present in network protocols.

DOME has a layered architecture. In particular, it uses an abstract C++ interface to access the network[2] so that it is network independent.

The MASTER Layer

This layer is fully localized on the Global Host. It knows the machine structure — number of crates (Multi Crate machine), number of units (Single Crate, Multi Unit) or number of **PB**'s (Single Unit) — but it acts on the whole machine according to its topological structure (node slices).

Topology manager

This module reads a configuration file and manages:

- The hardware layout; according to APEmille specifications, this layout uniquely defines a precise topology, i.e. the number of nodes along the X direction.

---

[1] In order to optimize network dispatching of method calls, it is necessary to mark each method parameter as input, output or input/output. With this additional information, the dispatcher chooses which parameters have to be transferred to the remote object and which parameters are changed by the method and have to be transferred back.

[2] The APEmille hardware specification covers two different network links between Local Hosts and Master Host: a standard (Fast) Ethernet and/or the **Fast Serial Link** connection. When Ethernet is used, DOME uses the standard Linux socket interface; otherwise it is used a special low overhead, custom protocol over the **Fast Serial Link**.

- The connectivity; it is possible to connect a big machine as several independent smaller units, i.e. a Single Crate, usually a 2*8*8 nodes machine, can also be connected as four independent 2*2*8 units.

This module is used by the Machine Interface to properly operate on the machine.

Machine Interface

This (**MI**) is the key module; each upper module or layer accesses APEmille HW devices through that interface. Its main advantage is that it offers the same coordinate system as the synchronous network — that is, (X, Y, Z) node coordinate, — as opposed to asynchronous network point of view, where each node is identified by (unit id, board id, node id) triplets.
It supports high-level functions such as reading/writing of entire slices of nodes using topological coordinates. Internally, it reorganizes the data in a suitable format according to Topology Manager knowledge; afterwards it sends them to the Slave layer of each involved **LH** using DOME.

Global root board

It consists of a Linux kernel device driver for the Root Board PCI card (**RB** driver or **RBD**). The Global Host acknowledges events coming from the **PB**'s — such as run-mode to system-mode transitions, HALT conditions and exceptions — by **IRQ**'s activity of the **RB**. This activity is detected by the driver and passed to the Machine Interface level. Conversely, **MI** uses this driver to manage the global state of the machine, i.e. the synchronous start of all the **PB**'s when going to run-mode.

Global Resource Container

It is a polymorphic object container. It stores pointers to objects of `Resource` C++ class. A `Resource` object abstracts the interface supported by a generic I/O device:

- Reading/Writing of the different APEmille basic types, such as integer, single & double precision real, complex and vector numbers as well as binary raw data.
- Random access (seeking).
- Memory mapping.

`Resource` objects are the targets of the System Services environment offered to APEmille programs.
`Resource` class is abstract as it defines a pure interface. Real classes — later on simply called resources — are derived and properly implemented according to the kind of I/O devices they involve, such as `FileResource` to access standard files, `RandomResource` to produce different kinds of pseudo-random numbers, and so on.
For example, a TAOmille file open operation involves the creation of a `FileResource` object and the storage of it along with a unique numeric identifier — the resource handle — into the **Resource Container**. Later I/O operations on that resource are referenced by the resource handle.
Each resource class is tagged with a unique number which acts as a resource type identifier. This identifier is used in the process of creating new objects of that kind.

The APEmille OS I/O extension model relays on the use of dynamic link libraries which implements new `Resource` derived classes, e.g. to interface with new, previously unsupported devices. These libraries can even be packaged within APEmille executables files and broadcast on the network of **Local Hosts**.

Global event handler

This is a module (**GEH**) that acts as an independent thread of execution inside the APEmille OS. It uses the **RB**'s driver to wait for events from the **PB**'s, reads the event cause and activates suitable code. It deals with:

- Decoding and completion of system service after HALT requests. It includes reading/writing to **GH** storage (global storage) or to the screen (both graphics and `tty`), seed generation for the APEmille random number generator, etc.
- Decoding and reporting of exceptions. In case an APEmille program defines an exception handler code, **GEH** manages to execute it.
- It dynamically attaches to special system service code — in the form of user provided dynamic libraries — as a way to extend APEmille OS capacity.

The **GEH** acts on the APEmille hardware using the abstraction interface provided by the Machine Interface module.

The User Interface Layer

At least four shells will be provided.

Basic program loader. A simple load/run/execute Unix like command taking an executable file as input and giving full interface to the user keyboard and console.

Basic monitor. It is a monitor capable of diagnostic features like reading and writing memories and examining register status. This tool can be used for hardware and software diagnosis or for low level debugging of system applications.

Symbolic debugger. A symbolic debugger is under development for debugging of user application at the source code level — for TAOmille or C++ code written for APEmille. — It will provide both a graphics and a console interface. It will be integrated with the APEmille profiler to analyze high-level code performance. This software will be covered by a future publication on the APEmille Development Tools.

C+TAOmille. That is a library of C functions. It allows C/C++ to run user applications on the Global Host — or on another workstation connected to it, — and:

- To launch one or more APEmille application;
- To wait for their completion;
- To put (get) data to (from) APEmille data memories.

That is best viewed as a basic tool in the development of a mixed parallel-sequential programming environment.

CONCLUSIONS

We feel that the main achievements of the APEmille OS project are:

- It provides the user of the APEmille High Level Language (TAOmille) an I/O model that is independent of the machine configurations.
- It thoroughly adopts the C++ object oriented features, even extending that object model across networked processes.
- Its layered structure shields against the use of our complex simulation environment and future HW evolutions.
- It is dynamically extensible in both the kind of system services and I/O targets (resources) supported.

BIBLIOGRAPHY

[1] A. Michelotti *et al.*: "*Behavioral simulation of APEmille computer*", in preparation.

[2] APEmille collaboration: "*Overview of the Distributed Object Management Environment in the APEmille operating system*", in preparation.