Integration of flattened device trees in Adam

Author: Nicolas Pouillon
Contact: nipo@ssji.net

Date: 2010-01-26
Revision: 124

Contents

Goals 1

Hardware 1

Boot sequence 2
- Bootloader: Early boot sequence 2
- Coarse BIST and Network configuration 2
- Fine BIST 2
- Running a kernel 2
- Runtime monitoring 2

Need for interoperability 2

Flattened device trees 2
- Origin 2
- FDT format 3
- Contained data 3
- APIs and example 4
- Usage for our problem 5

Online remapping 5
- Remapping application 5
- The cost function 5

Goals

Adam aims to create a System-on-chip (SoC) platform tolerant to defects. Defects may be caused by two different causes:

- Production imperfections: semiconductor technology is not perfect, and newer circuit manufacturing processes can't create perfect chips at a reasonable cost any more.
- Wear: semiconductors usually heat, and most of the time, this slowly degrades the hardware. With time, most circuits begin to have erratic behaviours, and get broken.

To be able to work with those defects, chips designers can make a choice: fault tolerance may either be ``offline'' or ``online''.

- On the one hand, offline fault detection and avoidance consists in testing chips at the factory, deciding which parts are usable, at what frequency, and guarantee different yet lifetime-constant characteristics for each chip produced. This is the easiest solution. Nowadays, many chips are sold as different versions but are actually the same design. The usual example is the Intel ``Pentium'' and ``Celeron'' which are actually the same die, but cache is partly non-functional in the latter. Another examples are the IBM Cell, on which some SPUs are disabled; or current GPUs where not all pipelines are enabled.
- On the other hand, online fault detection consists in designing a chip that can internally enumerate functional hardware, self-reconfigure in order to avoid broken parts, and still run even if system changes in its lifetime. This solution is harder to achieve, but it has many economical benefits. Online recovery is not mainstream yet. Some Flash chips internally remap memory blocks during the lifetime of the medium, but most of the time, the logic has no self-recovery yet.

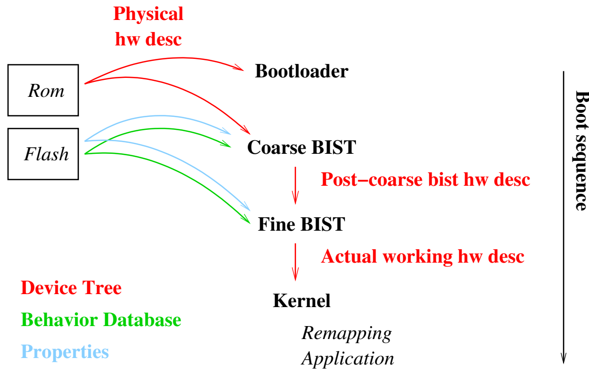
Here we'll focus on the online fault detection and avoidance, and more specifically on the software involved; from the hardware reset to the embedded Application execution.

Hardware

Adam is a specific hardware design, it is specific but yet quite generic chip architecture containing:

- a global Network-on-chip (NoC), made of routers and links,
- many clusters, consisting of Processors (CPUs), memory banks, and other small devices, interconnected through a local interconnection component. In turn, this local interconnect is attached to the NoC through a Network interface Controller (NIC).
- Moreover, each cluster contains an hardware controller dedicated to self-testing, monitoring and reconfiguration of the chip (COMOR).

Boot sequence



Bootloader: Early boot sequence

When the chip boots, NoC is not yet tested, thus it is not assumed configured nor usable. Each cluster has to boot on its own, self-test, and configure its NIC.

This early boot code must be present in each cluster. We saw it can fit in a small ROM, and should not have to change with the firmware updates. The ROM also contains an enumeration of original hardware present in the cluster, this list is used to know where to perform basic tests.

These tests are quite basic: no in-depth functional testing is performed here.

Coarse BIST and Network configuration

Then each working cluster tries to configure its NIC, and communicate with others. The cluster's ROM informs about coordinates of each cluster in respect to the NoC, thus each cluster knows addresses of its neighbours.

Here, some clusters may not be able to communicate because of broken network; they'll remain alone, and should self-disable. Remaining clusters elect a master.

At this stage, there is one global software system running on the chip, with one master. This master has to exhaustively check the chip is actually good.

Fine BIST

Now NIC is configured and network is declared available, clusters declared good so far are able to fetch big testing applications from external memories. This test actually determines if every piece of hardware can safely be declared good.

This part creates a global chip map, containing the actual running hardware subset.

Running a kernel

Now the chip is tested, we can run a classical Operating System (OS). This operating system will be the host for the application. It needs a description of the hardware.

Runtime monitoring

While the operating system runs, another software application continuously runs to monitor the chip evolution. This watches temperature, voltage, CPU load, ... All these values are stored in Distributed Raw Event Tables (DRET).

Need for interoperability

As the successive software codes running in the boot sequence may be developed by different teams, may use different supporting OS kernels, and may be updated, the hardware description passed from stage to stage cant be a simple runtime pointer-based in-memory structure. We have to choose a representation at each stage, or if possible, a global representation format for architecture definition: this allows reusing of basic software blocks, and factors-out some tedious code.

An unique format, not tied to a particular memory layout, has to be translated from a runtime easy-to-use from code version. This is called serialization; the other operation, translating the common format to a runtime structure is deserialization.

If algorithms are able to use the commot format without deserialization, this is called working in-place. Working in-place is important as early software stages do not implement a memory allocator, forbidding use of complex pointer-based data structures.

From the hardware description present in the ROM of each cluster, to the hardware description produced by the exhaustive testing; we had better be able to serialize and deserialize interoperable hardware descriptions. We could have developed a new hardware representation from scratch, but as we end-up booting an existing operating system, we tried to look if this kind of problem had been solved before. It had. We saw the Flattened Device Tree as a good candidate for hardware description serialization.

Flattened device trees

Origin

Flattened Device Tree (FDT) is a subset of the OpenFirmware standard, also known as IEEE 1275, developed by IBM, Sun and Apple to address the problem of BIOS equivalent for Sparc and PowerPC architectures.

This specification defines a clean set of services provided by a compliant firmware, and an interface for the guest OS to query the Firmware. The firmware passes hardware description to the guest OS through the FDT.

Altogether with the FDTs, IEEE 1275 also defines usage of a Forth interpreter, but this is not needed for our tasks, and will be omitted.

Nowadays, OpenFirmware is not present is many machines any more, but the FDTs are becoming the

de-facto standard for hardware platform definition. Embedded platforms (DSL routers, network equipments, PDAs, phones, ...) usually run common kernels like Linux. These platforms use a bootloader like U-boot, which acts as a firmware, and passes the kernels a FDT definition of the platform.

FDT support in Linux and support in NetBSD is mainline.

FDT format

The Device Tree can be represented in a textual form for human edition, but is preferred as a binary equivalent (Device Tree "Blob") when manipulated from software code. Existing tools and libraries are available and provide parsing routines, and a compiler from textual representation to binary form.

FDT blobs are self-conained, position independant. They can be moved around in memory without any need for processing. The blob format allows in-place walk-through at no cost, and in-place random-access at a minimal cost.

FDT access code is tiny; a typical bootstrap code retrieving memory layout from a blob can fit under 2KiB for a PowerPC.

FDTs are a tree of records (nodes), containing key-value pairs (properties), value being a string, a number, an address, or a reference to another node. Base type is a 32-bit integer called "a cell". One or more cells can be used to describe an address or a size. Properties are named, and have an optional value. Some property names are defined by the standard (they usually are prefixed by "#"), but all non-reserved names are available for implementation-dependant use. This makes this format easy extensible.

Here is such an example tree:

```
/dts-v1/;

/ {
    model = "Basic With Topology";
    #address-cells = <1>;
    #size-cells = <1>;

    cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        Arm,v6k@0 {
            name = "Arm,v6k";
            device_type = "cpu";
            reg = <0>;
            icudev_type = "cpu:arm";
            ipi = <&{/xicu@0xd0200000} 0>;
        };

        Arm,v6k@1 {
            name = "Arm,v6k";
            device_type = "cpu";
            reg = <1>;
            icudev_type = "cpu:arm";
            ipi = <&{/xicu@0xd0200000} 1>;
        };

        Arm,v6k@2 {
            name = "Arm,v6k";
            device_type = "cpu";
            reg = <2>;
            icudev_type = "cpu:arm";
            ipi = <&{/xicu@0xd0200000} 2>;
        };

        Arm,v6k@3 {
            name = "Arm,v6k";
            device_type = "cpu";
            reg = <3>;
            icudev_type = "cpu:arm";
            ipi = <&{/xicu@0xd0200000} 3>;
        };

    };

    tty@0xd0200000 {
        device_type = "uart";
        reg = <0xd0200000 0x10>;
        irq = <&{/xicu@0xd0200000} 0>;
    };

    block@0xd1200000 {
        device_type = "block_device";
        reg = <0xd1200000 0x20>;
        irq = <&{/xicu@0xd0200000} 1>;
    };

    xicu@0xd2200000 {
        device_type = "interrupt_controller";
        input_lines = <2>;
        output_lines = <4>;
        ipis = <4>;
        timers = <1>;
        reg = <0xd2200000 0x1000>;
    };

    memory@0x7f400000 {
        device_type = "memory";
        cached;
        reg = <0x7f400000 0x00100000>;
    };

    memory@0x9f400000 {
        device_type = "memory";
        cached;
        reg = <0x9f400000 0x00100000>;
    };

    chosen {
        console = &{/tty@0};
    };
};
```

Contained data

Device Trees (DT) address the problems of variable bus widths, IRQ routing, mix of auto-enumerated buses (like PCI or USB) and fixed address buses (like AMBA or NoCs). DTs are a processor-centric (thus software-centric) view of the hardware. They define:

- Available devices, their addresses, model, version, ...
- IRQ routing between devices
- Physical memory regions, with their attributes (address, size, latency, coherence model, ...)
- Processors, with their cache attributes, supported extensions
- Memory regions that are already used by firmware or OS bootstrapping code

This basic information is enough to get an OS booting. It will be able to initialize hardware, allocate memory, find mass-storage devices, ...

Moreover, being trees, DTs can contain information specific to NUMA architectures: as an extension to this standard dataset, we can add subtrees and properties to describe the topology of the platform, even if it should not be visible from a software-centric point-of-view.

For instance, the following hypothetical example contains a subtree of the previously seen device tree containing the topology of the system:

```
topology {
    interconnect_type = "MyNoC";
    hop_latency = 2;
    layout = "mesh";

    cluster@0 {
        coordinates = <0 0>;
        interconnect_type = "crossbar";
        hop_latency = 1;

        devices = <&{/xicu@0xd2200000}
                  &{/block@0xd1200000}
                  &{/memory&0x9f400000}>;
        processors = <&{/cpus/Arm,v6k@0}
                     &{/cpus/Arm,v6k@1}>;
    };

    cluster@1 {
        coordinates = <0 1>;
        interconnect_type = "crossbar";
        hop_latency = 1;

        devices = <&{/tty@0xd0200000}
                  &{/memory&0x7f400000}>;
        processors = <&{/cpus/Arm,v6k@2}
                     &{/cpus/Arm,v6k@3}>;
    };
};
```

APIs and example

DT blobs are formatted binary data, designed to be easily walked-through by a software library. Here we'll describe the accessing API.

Let's comment the API through an example based on real code. We'll implement a simple device-tree copier. The example will be spread accross the API definiton.

A simple serialization library may be specified as an constructive 4-calls API:

Each call adds data to the tree:

- create a new context, which writes a blob

```
error_t fdt_writer_init(
    struct fdt_writer_s *writer,
    void *blob,
    size_t available_size);
```

- enter in a node, with a node name

```
void fdt_writer_node_entry(
    struct fdt_writer_s *writer,
    const char *name)
```

- add a property on current node, with a name and value

```
void fdt_writer_node_prop(
    struct fdt_writer_s *writer,
    const char *name,
    void *data,
    size_t len);
```

- leave current node

```
void fdt_writer_node_leave(struct fdt_writer_s *writer);
```

- add a memory reservation, which is a specificity of DTs we don't explain here

```
void fdt_writer_add_rsvmap(
    struct fdt_writer_s *writer,
    uint64_t addr,
    uint64_t size);
```

- finalize the context

```
error_t fdt_writer_finalize(
    struct fdt_writer_s *writer,
    size_t *real_size);
```

On the other hand, a simple consumer program may implement the following protocol, symmetrical to the other:

User provides 4 functions, to be called by library code on different events when walking through the device tree, let's provide sample codes that implement a blob copier:

- on node entry, it will receive the node name

```
bool_t copy_node_entry(
    void *private,
    struct fdt_walker_state_s *state,
    const char *path)
{
    struct fdt_writer_s *writer = private;

    fdt_writer_node_entry(writer, name);
    return 1;
}
```

- on property, it will receive the property name and its raw value

```
void copy_node_prop(
    void *private,
    struct fdt_walker_state_s *state,
    const char *name,
    const void *data,
    size_t datalen)
{
    struct fdt_writer_s *writer = private;

    fdt_writer_node_prop(writer, name,
                         data, datalen);
}
```

- on node leave

```
void copy_node_leave(void *private)
{
    struct fdt_writer_s *writer = private;

    fdt_writer_node_leave(writer);
}
```

- on memory reservation, which is a specificity of DTs we don't explain here

```
    void copy_mem_reserve(
        void *private,
        uint64_t addr,
        uint64_t size)
    {
        struct fdt_writer_s *writer = private;

        fdt_writer_add_mem_reservation(writer, addr,
                                       size);
    }
```

Both these APIs may be implemented without need for dynamic memory allocation, thus easing their use in a highly-constrained environment, like a bootloader.

An example context using all the defined functions above, and illustrating on-stack only memory usage is:

```
void copy_blob(
    void *out,
    size_t out_size,
    void *in)
{
    /* Prepare context for library functions
     */
    struct fdt_writer_s writer;
    struct fdt_walker_s copier = {
        .private = &writer,
        .on_node_entry = copy_node_entry,
        .on_node_leave = copy_node_leave,
        .on_node_prop  = copy_node_prop,
        .on_mem_reserve = copy_mem_reserve,
    };

    /* Initialize constructive DT creator
       API */
    fdt_writer_init(&writer, out, out_size);

    /* Tell the consumer API to walk through
       blob pointed by "in", using context
       and functions pointed in "copier"
     */
    fdt_walk_blob(in, &copier);

    /* Tell the constructive API to finalize
       the produced blob */
    fdt_writer_finalize(&writer, &real_size);
}
```

Usage for our problem

There are four consecutive usages of FDTs in Adam:

- In the ROM of each cluster, we must have a detailed description of the cluster's hardware, and a global description of the chip:
  - position of the cluster in the whole chip,
  - global chip's connections to the external memory and devices.
- Between the early boot sequence and the exhaustive test application, early boot sequence must pass validated hardware subset to the next layer.
- Between the exhaustive test application and the OS, we need the actual usable hardware description.
- In the application remapping phase, when the OS loads the final application on the SoC, the OS needs the enumeration of available processors, memories and global topology of the chip (the final FDT).

Online remapping

Once the OS booted, the SoC is not in its final state yet. The main purpose of our SoC is probably to implement a given functionality. In order to get the most of the chip, the application needs to be smartly mapped on the hardware.

This smart mapping basically involves assigning software threads to processors and assigning software objects to memory banks in a manner to minimize NoC usage and latencies.

Remapping application

Mapping is done by a dedicated piece of software. It may implement any mapping algorithm, like genetic algorithms, or simulated annealing. This class of algorithms needs:

- A description of a parallel application
- A description of the mapping target (here is is the final DT)
- A cost function

The cost function

In order to efficiently map the application on the usable hardware, application description must contain:

- explicit communication paths between tasks, thus creating a task graph;
- memory usage of any software resource;
- memory size of any communication channels (dedicated message-passing channels, shared memory, ...);
- normalized relative load of each task in the graph.

Having a task graph makes cost function an easier problem, as we can statically tell whether the NoC, the CPUs, and the memory banks will be loaded or not.