ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP070

BUILDING, DEPLOYING AND PROVISIONING EMBEDDED OPERATING SYSTEMS AT PSI

D. Anicic, Paul Scherrer Institut, Villigen, Switzerland

Abstract

In the scope of the Swiss Light Source (SLS) upgrade project, SLS 2.0, at Paul Scherrer Institute (PSI) two New Processing Platforms (NPP), both running RT Linux, have been added to the portfolio of existing VxWorks and Linux VME systems. At the lower end we have picked a variety of boards, all based on the Xilinx Zynq UltraScale+ MPSoC. Even though these devices have less processing power, due to the built-in FPGA and Real-time CPU (RPU) they can deliver strict, hard RT performance. For high-throughput, soft-RT applications we went for Intel Xeon based single-board PCs in the CPCI-S form factor. All platforms are operated as diskless systems. For the Zynq systems we have decided on building in-house a Yocto Kirkstone Linux distribution, whereas for the Xeon PCs we employ off-the-shelf Debian 10 Buster. In addition to these new NPP systems, in the scope of our new EtherCAT-based Motion project, we have decided to use small x86_64 servers, which will run the same Debian distribution as NPP. In this contribution we present the selected Operating Systems (OS) and discuss how we build, deploy and provision them to the diskless clients.

INTRODUCTION

At PSI we operate four accelerator facilities: HIPA and PROSCAN proton facilities, and SLS and SwissFEL electron facilities. They age from few years up to several decades. A variety of hardware, computers, operating and control systems have been used during this time.

Hardware-wise mostly VME based systems are used, but also quite a lot of PC based systems and a variety of commercial small computer boxes. We also use many virtualized systems.

On the software side we have VxWorks, Windows, Scientific Linux, Red Hat Enterprise Linux, and a diversity of Embedded Linux Systems, all in several versions.

Some OS-s have been installed on local hard disk, some others are network-booted. Experience shows that network-based provisioning is simpler for updates and changes.

NEW PLATFORMS

Three new platforms are supported:

- Zynq Ultrascale+ based computers
- x86_64 single board computers (SBC) in CPCI-S
- Small x86_64 servers

Zynq Ultrascale+ Based Computers

The decision to use Zynq Ultrascale+ platform was taken already some time ago. Several tests were performed, and three types were envisioned, for small, medium and high performance and/or power consumption. But because several internal groups have been involved in the development it actually resulted in almost ten different configurations. Some groups decided to go forward with ready available development boards bought directly from manufacturers, whilst others took either a Silicon-on-Chip (SoC) or Silicon-on-Module (SoM) approach, to develop their own boards. Unlike we initially assumed, this variety actually turned out not to be a problem.

x86_64 SBC

For some Zynq Ultrascale+ systems the CPCI-S bus was targeted as our crate and bus standard, providing the possibility, where needed, to create more powerful, x86_64 based systems, as CPCI-S bus controller and also as number cruncher, if needed. Presently we have only one board type with Intel Xeon CPU, dual Ethernet connection, and 16GB of RAM.

Small x86_64 Servers

For the Motion project (motors) based on EtherCAT, our initial design involved installing operating system locally. With the operating system for x86_64 NPP already in place, the decision was taken to re-use it also for the small servers. This increases the synergies between our groups, saves manpower and, will simplify maintenance. Although any kind of PC could be used, our great wish was that everybody would use the same small HP DL20 servers. This would simplify troubleshooting in case of problems and provide for common replacement stock.

CHOOSING AND BUILDING OPERATING SYSTEMS

Generally, we will use Linux for all our future projects. Our wish was also to use a RealTime (RT) patched Linux kernel.

Zynq Ultrascale+ Based Computers

For the initial considerations we have been using Xilinx PetaLinux SDK. This was quite convenient as a proof of concepts, but it turned out to be too complex to handle, because our different board development groups have adopted different versions. The development also took longer than planned, new hardware revisions have been coming, and the need for newer versions of SDK were occurring. It was not possible to easily satisfy all development needs by supporting several versions.

The RT kernel patch was also not available for all kernels provided by PetaLinux SDK.

For a functioning kernel, the Linux kernel device-tree, which describes hardware-to-kernel interface, must be provided for each board type. Unfortunately the device-tree

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP070

interfaces have been extremely incompatible between kernel versions, and there was a need to standardize in that respect, too.

Therefore we took the decision to proceed with a single development environment, and single kernel for all our board types.

For our Linux distribution, we took Yocto project in version 4.0 Kirkstone.

For the kernel, we have taken one of the latest kernels supported by PetaLinux, for which there is also the RT patch available. We do not use the vanilla kernel, but the Xilinx patched kernel, with all the necessary Xilinx drivers included. Currently, we are using kernel 5.15.19-rt29 (RT patch 29).

Although building Yocto projects provides a way to add one's own custom 'recipes' for adding or modifying packages, we have not taken that approach. We have simply configured the build system to build base Linux system, RootFS, with all packages we could need. We also build the cross-compiler toolset for offline usage. The idea is not to have to re-build Yocto project again. Then we install it for network boot (read further in BOOTING).

Next, we build the RT patched Linux kernel offline (out of Yocto build), and simply add it to the RootFS.

Besides RootFS and kernel, we also need device-tree for each board type. Those are provided by board developers. We add some common configurations, build the device-tree binaries, and add them to the RootFS.

To start booting, the standard method with u-boot, packed in Xilinx BOOT.BIN is used. U-boot also needs device-tree, so we have to collaborate with the board developers in this field, too. At PSI we are still aligning our ideas with all the involved groups, in order to define a common environment to provision the device-trees and build u-boot and BOOT.BIN for all used board types.

We are using SysVinit to bring Linux up, so we also add a few startup scripts for mounting the EPICS controls system and related NFS shares, and a mechanism which enables us to run user specified scripts, before starting the EPICS control system.

We also compile a few more applications, either our own or those for which packages are not available in Yocto, and add them to the RootFS, too.

x86_64 SBC and Small Servers

The initial idea was to use the same operating system as used for all other PSI controls workstations and servers, but with the RT kernel enabled instead. That would have meant RedHat Enterprise Linux, version 7 at that time. The main reason for this approach was to let our IT Department take care of OS installation and maintenance. However, this idea ran into a couple of obstacles. Firstly, the IT preferred kernel was not cost-free, and secondly was their insistence on regularly applying all OS updates. Given the operational environment of our machines neither of these constraints were possible for us.

Additionally, this approach would have required local disk installations, which was also not our preference.

Therefore, we have decided to go for network boot, and to manage this environment with controls personnel.

Now we had to make two decisions, which provisioning system to use, and which Linux distribution. Both would have to be free to use.

After some limited investigation, we have chosen the Warewulf, Cluster Management & Provisioning tool.

For our Linux distribution, after initially considering RedHat or CentOS, across experimenting with Ubuntu, we finally decided on using Debian.

Debian offers RT Linux patched kernel packages. And the tests showed that we got the best RT performance with it. Unfortunately, due to the long development time, the pre-built kernel package, for our preferred version, was obsoleted by newer kernel versions. Since we also need to add a few of our own kernel-version dependent drivers, we have downloaded the kernel sources, saved them locally, and we built the RT-patched kernel from the 'frozen' sources.

We have also been experimenting with Debian 11, and the newer kernel, 5.10.178-rt86, but it shows greater latencies and less determinism, so we presently stick to kernel 4.19.208-rt88. Although Debian version 11 became available, we have decided to keep using Debian 10 Buster, at least for the platforms we are currently using.

Warewulf is a tool and server for provisioning operating systems. Although it claims to be a cluster management software, it does not have to be a cluster at all, it can simply be a bunch of independent systems.

Warewulf provides all necessary services and handles their configuration and startup. We intended to use it with all necessary services in a separate network, but now we use only http service for loading (more about it in BOOTING).

One of the other Warewulf's features is node configuration. However, we do not use this feature as we found, it is much more convenient to text-edit configuration files and simply re-start the http server to activate changes.

Currently, we only use the Warewulf's container and kernel import and image build features. More concretely, Warewulf can import the Docker images, unpack them, and build images suitable for http download, which is more or less all we need.

Building Debian system is done out of Warewulf, in Docker build, which we have integrated in our GitLab CI/CD mechanism.

We have implemented a two-stage Docker build. In both cases we start with Debian Buster.

In the first stage we additionally install essential kernel build tools, then we RT patch and build the kernel from the frozen sources. The Debian kernel build process automatically creates all the needed installation packages.

Next, we do out-of-tree build of our extra kernel modules, and also package them for later installation.

In the second stage, we install all the packages required for running on the client nodes, including, in the previous stage, built kernel and modules packages. Presently our Debian system consists of almost 400 directly selected or dependent packages.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP070

Then we install and enable a few of our own additional services, which we need for EPICS. This would include mounting additional EPICS control system and related NFS shares, and a mechanism which enables us to run user specified scripts, before starting EPICS control system.

Warewulf has a feature of 'overlaying', implemented in two stages: system and runtime overlay.

The system overlay is loaded latest at boot time, and adds or overwrites existing files in the RootFS. This can be used to modify defaults, typically in '/etc' or the user home directories.

The runtime overlay behaves functionally same as the system overlay, but it gets re-loaded every minute. This could be used, for example, to modify 'user', 'group', or 'sudoers' files. This gives possibility to change or add things without a need to re-boot the client.

BOOTING

Zynq Ultrascale+ Based Computers

Our Zynq based systems load BOOT.BIN from the SDCARD. The first stage boot loader, part of it, then starts u-boot. By default, u-boot offers TFTP based network transfer for loading, and we simply take advantage of it. Since we already have clients using TFTP boot, and servers configured for that, it was a clear way to go.

Since we have different board types, and boot the same kernel and RootFS, we need a mechanism to differentiate between them. To do this we have introduced a dedicated u-boot environment variable, 'boardid', containing the unique board type-name. Further on, for every client IP-name, and used TFTP boot-server and nfs-server names have to be defined, too. Some other environment variables, are also required and are all passed to kernel, too.

The boot process first gets an IP address by DHCP, and then gets, per TFTP, our u-boot script, which continues the boot process. It downloads, also per TFTP, the Linux kernel, and according to 'boardid' selects board dependant device-tree to download. It composes the kernel parameter list and passes them to the kernel.

The Kernel then uses those parameters to mount the RootFS over NFS, from specified nfs-server. Then kernel starts Linux by running SysVinit 'init' process.

Linux also gets parameters (either through kernel's 'cmdline', or by reading u-boot environment variables from SDCARD directly). They are used to mount additional NFS shares, which are needed for running our EPICS control system.

x86_64 SBC and Small Servers

In the PSI controls group we had not previously encountered the network boot process associated with the x86_64 (better known as PC) architecture. Of course, our IT supports it for Windows and Linux network installations. Initially we assumed that making Warewulf use the same mechanism would not be a problem.

However, we were wrong. Warewulf booting would involve a PC enabling PXE network boot and then using DHCP, TFTP and HTTP for further steps. Initially we have tried this method, with our IT Department's assistance, and it worked. However, they were not happy with it, because it would have placed 'a high burden' on our network VLAN and DHCP setup and would require IT Department's expert involvement to configure each new client. Additionally, they were reluctant to delegate this task to us. Subsequently, we have tried, again with IT assistance, to create a private VLAN network for Warewulf booting purposes only, fully under our control. This also worked, but the approach was again rejected with the similar explanation. Another drawback was that every client would have also needed access to additional VLAN networks, for control system purposes. This would have potentially made possible to avoid our IT VLAN routing restrictions.

At this point it seemed that we had reached a dead end, but finally we managed to arrive at a very elegant solution.

By default, PXE would do DHCP and then by TFTP load iPXE. iPXE would have then used DHCP again to obtain the client boot configuration, and then load needed images using HTTP protocol.

The new solution is to simply build iPXE ourselves, embedding the configuration script in it (which contains required server address), and simply copy it to the USB stick. We then configure the PC to use USB as boot device instead of network PXE, and that's it.

So, now iPXE gets loaded from USB stick, does some necessary hardware initialization, and starts DHCP. It does it on all network interfaces, until it gets an answer on one of them. Then it uses that network interface to obtain the client node configuration with HTTP, and loads accordingly node specific images, also by using HTTP protocol.

Suddenly, another problem emerged. The HTTP is using port 9873 instead of default port 80. It turned out that it was blocked from some VLAN network. Fortunately, requesting and getting that port opened from our IT was not an issue.

Now we got possibility to boot images from Warewulf server, without the need for any further IT assistance.

Booting is then quite straightforward. After container (RootFS), kernel, kernel modules and system overlays are downloaded, iPXE unpacks them in RAM-disk, and starts Linux. Linux then mounts additional NFS shares, in order to run our EPICS control system. We use Warewulf provided feature of passing 'ClusterName' parameter to the clients for choosing adequate NFS shares to mount.

FINAL CONSIDERATIONS

Currently, we have only about a dozen clients booting from the Warewulf server. Many more clients are coming. The images loaded are almost 700 MB in size, compressed. This could be a burden on network in situations when many clients try to boot at the same time, for example, after power interruptions. We will deal with a problem if it arises.

Also, the images are loaded into RAM-disk, the expanded image size is almost 1.8 GB, which is quite a lot, but should be no obstacle for systems with plenty of RAM.