Experience of Building and Deployment Debian on Elbrus Architecture

Andrey Kuyan, Sergey Gusev, Andrey Kozlov, Zhanibek Kaimuldenov, Evgeny Kravtsunov
Moscow Center of SPARC Technologies (ZAO MCST)
Vavilova street, 24, Moscow, Russia
{kuyan_a, gusev_s, kozlov_a, kajmul_a, kravtsunov_e}@mcst.ru

***Abstract*—This article describe experience of porting Debian Linux distribution on Elbrus architecture. Authors suggested effective method of building Debian distribution for architecture which is not supported by community.**

I. Introduction

MCST (ZAO ”MCST”) is a Russian company specializing in the development of general purpose CPU with Elbrus-2000 (e2k) ISA [1] and computing platforms based on it [2].Also in the company are being developed optimizing and binary compilers, operating systems. General purpose of microprocessors and platforms assume that users have the ability to solve any problems of system integration with its help. At the user level universality is provided by distribution of operating system. Distribution uses architecture-dependent capabilities of software components, such as the kernel, arch-dependent system libraries and utilities, compilers. Nowadays there are several large and widely used distributions supported by community: Gentoo, Slackware, Debian. We chose Debian guided by wishes of our customers and because Debian is one of the most stable and well supported Linux distributions. Debian has system of package management, installer and all this components are supported by a large community of developers. 99% of Debian packages are architecture independent applications and libraries, which is written in C/C++ or Perl, Python, etc. There is a popular way to building distribution for unsupported by community architecture: download a limited number of software sources with different versions and add a popular package manager, for example dpkg with limited functionality. This approach allows to provide for user a distribution with basic functionality and even call it a Debian-like. A significant drawback of this way is the complexity or even impossibility of extending the package set. Due to software dependencies, even if they are, do not match with dependencies of pure Debian and nothing additional can be built with using dpkg-buildpackage. This drawback is not so significant for specialized systems that solve the limited range of tasks, but a problem for the platform, which is claimed as universal. This problem is solved by porting Debian on new architecture in its purest form with preserving all dependencies. The resulting distribution will allow to solve all kind of the current problems, even more a system integrator will be able to build new packages using the package manager.

II. Debian package managment system

Debian is built from a large number of open-source projects which maintained by different groups of developers around the world. Debian uses the package term. There are 2 types of packages: source and binary. Common source package consists of `*.orig.tar.gz` file, `*.diff.gz` file and `*.dsc` file. `*.orig.tar.gz` file contains upstream code of a project, maintained by original developers. `*.diff.gz` file contains a Debian patch with some information about project, such as build-dependencies, build rules, etc. `*.dsc` file holds an information about `*.orig.tar.gz` and `*.diff.gz`. Some source packages, maintained by Debian developers (for example dpkg) may not comprise `*.diff.gz` file because they already have a Debian information inside. Binary packages can contain binary and configuration files, scripts, man pages, documentation and another files to install on the system. In addition, each package holds metadata about itself. Binary packages represented as `*.deb` files. Source and binary packages contains information about build and runtime dependencies respectively. Build dependencies are binary packages that has to be installed on the system for building source package. Runtime dependencies are binary packages that has to be installed on the system for correct work of package.

In fact, Debian solves two main problems:

1) supporting appropriate versions of packages
2) managing packages with dpkg and support tools

For building any source package, some utilities have to be installed on the system. For example, every source code required make utility. Set of packages that have to be installed on the system for building called build-essentials. Some of them are Debian-specific. This because Debian has it’s own tools and features for building source code directly into packages. Debian patch for source package holds, as mentioned, build rules. Rules is the makefile with set of standard targets, such as “clean” and “build”. Build process starts with run `dpkg-buildpackage` script which is part of dpkg-dev package. This script checks build dependencies, gets some information about environment and runs the desired targets from rules.

III. Architecture-dependent software and Debian release selection

Architecture-dependent part of software stack (see Fig.1) comprises the following components: Linux kernel, glibc library, toolchain (compiler lcc and binutils), strace and debuger gdb. Development of this components for new CPU architecture is long and laborious process. Versions of this components are crucial for Debian release number selection.

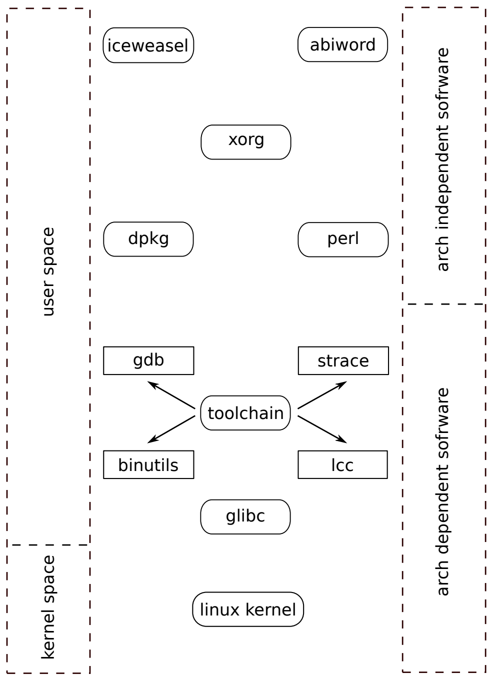

Fig. 1. Software stack

In the table 1 represented comparison of MCST and Debian architecture-dependent components, according to this table Debian Lenny is appropriate one for porting.

TABLE I
ARCHITECTURE-DEPENDENT COMPONENTS COMPARISON

version	kernel	glibc	binutils	gcc
MCST	2.6.33	2.7	2.18	3.4.6
Lenny	2.6.26	2.7	2.18	4.3.2
Squeeze	2.6.32	2.11	2.20	4.4.5
Wheezy	3.2	2.18	2.10	4.7.2

MCST compiler team has developed two kinds of toolchain: cross and native. Cross-toolchain, which is running x86-machine, allow to generate code for e2k ISA. MCST architecture-dependent components consist of binutils-2.18, glibc-2.7, gdb-7.2, gcov, dprof, libstlport, libffi and lcc (compiler compatible with gcc 3.4.6). Compiler lcc is original development of MCST [3] and uses frontend which is licensed from Edison Design Group (EDG)[4]. Remaining components are the result of porting the GNU utilities on e2k-architecture and contain a large number of changes arising from architectures peculiarities.

IV. Technical issues for challenging

Define main technical problems in porting Debian on a new CPU architecture:

1) The chicken and egg problem: for build any package we need build-essentials set. But we haven't this one because we haven't build and runtime dependencies for build-essentials packages.
2) Some packages may cyclically dependent on each other. Example of cyclic dependencies shown in figure 2.
3) While building build-essentials there isn't simple way to pass arch identifier to configure script. This because we built build-essentials without dpkg, which provides features for auto-detecting cpu type. The problem is compounded by the fact that some packages strictly depends on the architecture type.
4) Compilation speed problem. For running iceweasel, gnumeric and so on we need to have almost 2500 binary packages in our repository. Some of them are very large and build takes more than a day.
5) Difference between gcc and elbrus toolchains: gcc toolchain packages set and elbrus toolchain packages set vary greatly, elbrus compiler don't support some new language extensions or compiler flags.

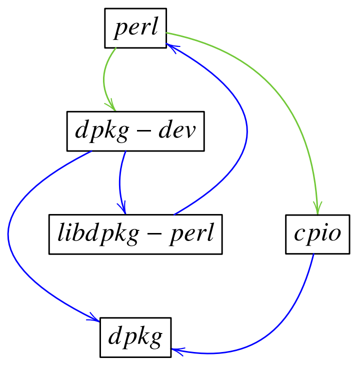

Fig. 2. Example of cyclic dependencies.

Fig.2 illustrates a part of runtime and build dependencies graph for dpkg package. This graph represents existence of cyclic dependencies, blue arrows depict runtime dependencies and green arrows depict build dependencies. Package dpkg-dev is build dependence of perl and it is used only in process of building perl package, but dpkg-dev required for working pakage libdpkg-perl, which depend on perl. Thus perl interpreter should run on machine for building perl package. Solutions of these problems are presented below.

V. Solution for package manager: building essentials, breaking cyclic dependencies

Packages from build-essentials set have been built in the following algorithm:

1) Using debtree utility we built dependency graph of required packages for build-essentials set.
2) Every package from graph has been built with native toolchain and configure-make mechanism, without dpkg.
3) In build process, we broke some cyclic dependencies. Break dependencies algorithm shown in figure 2. It

is seen that if package A depends on package B and package B depends on package A, we should build package B 2 times: first time with broken A dependence, which mean we don't pass corresponding option to configure second time after building A package in due form.

4) Result of building has been installed on the machine and at the same time wrapped in the package manually.
5) After building all graph elements we built dpkg with configure-make mechanism.
6) Then we verifyed package manager efficiency by install on the machine all deb packages that we got manually.
7) All build-essential set have been rebuild with dpkg-buildpackage.

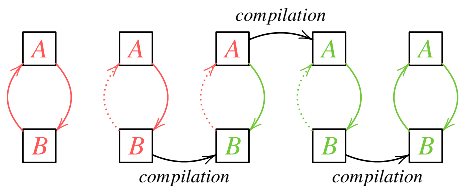

Fig. 3. Algorithm of resolving circular dependencies

After sequential implementation all 7 points of algorithm we've got small system which can be used for building all other neccesary packages with dpkg. So, manual building with configure-make used only in initial phase. After dpkg all packages have been building with standard debian rules.

VI. Solution for compiler problems

As was mentioned above, Debian for e2k ISA is based on using lcc compiler, which was developed by MCST compiler team. Lcc compiler is using edg frontend, which is compatible with gcc 3.4.6. Lcc dosen't support some extensions of C language, for example nested functions. Programs written with using of nested functions, should be patched to unwrap this functions. Fortunately as the experience of porting Debian distribution nested functions are seldom used, so patches require only for several packages. One of those packages is `bogl-bterm`, which is used by Debian Installeras a graphical frontend. Many of software developed by GNU project are using gcc directives `__attribute__((attribute-list))`, and some of this directives are not supported by lcc, so for succesful compilation of such code corresponding patches should be applied to source package. MCST toolchain contain library for STL support - libstlport, which is different from standard STL library libstdc++, also libstlport is not supported some STL features. Due to this distinctions such packages as mysql and exim require patches or special tuning in makefiles to be successfully compiled.

VII. Hybrid compile farm

Due to the existence of two toolchains (cross and native) feasibility to decrease compilation time appeared. This technique is based on using hybrid scheme of source-code compilation via distcc. Distcc (distributed C/C++/ObjC compiler)

TABLE II
Compile farm configuration

parametr	e2k	x86
CPU name	Elbrus-2C+	Core2 Duo E8400
CPU frequency	500 MHz	3.00GHz
Number of cores	2 + 4(dsp)	2

[5] is a software for speeding up compilation process by using distributed computing over network. Compilation of source package is starting on e2k host with native toolchain and distcc client, this client is sending preprocessed files to servers with x86-architecture for code compilation using cross-toolchain. After compilation server return object files to clients which perform operations of linking and `*.deb` packages forming. Hybrid compile farm, which configuration is described in Tab. 2, was used for building linux distribution of 350 source packages. This method allow to decrease up to 5 times average time of package building as compared with native compilation. Fig. 2 shows a generalized scheme of compile farm, which is described above.

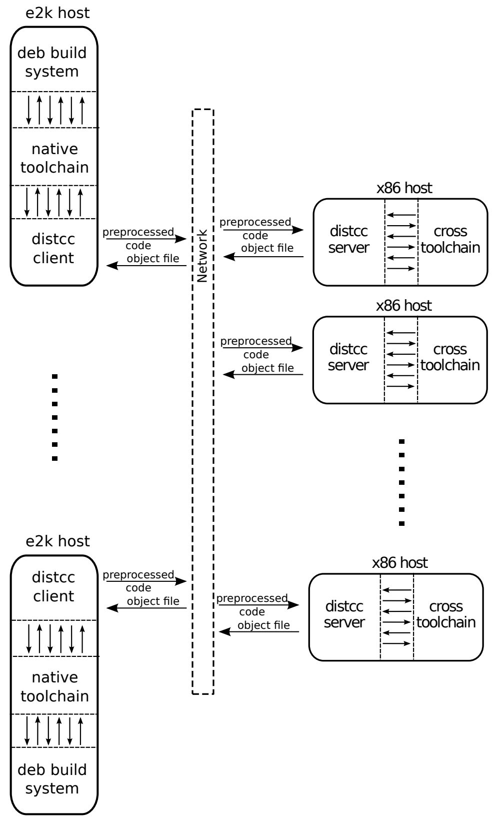

Fig. 4. Hybrid compile farm

VIII. Solution for arch type

When any package is builded with `configure-make`, there are two main ways to pass arch identifier to the configure script: pass `--build=arch` option to the configure fix config.guess script which contains all known by community arch identifiers For correct building we used both. Typical config.guess patch is as follows:

```
--- config.guess-orig
+++ config.guess-fix
@@ -854,6 +854,9 @@
        crisv32:Linux:*:*)
                echo crisv32-axis-linux-gnu
                exit ;;
+       e2k:Linux:*:*)
+               echo e2k-unknown-linux-gnu
+               exit ;;
        frv:Linux:*:*)
                echo frv-unknown-linux-gnu
                exit ;;
```

Typical configure script run as follows:
`./configure --build=e2k-unknown-linux-gnu`

Dpkg has feature for auto-detect arch type of the host and target machine which is called dpkg-architecture. It uses dpkg internal config files, such as cputable. This one contains all Debian known CPU and consists of three columns: Debian name for the CPU, GNU name for the CPU and regular expression for matching CPU part of config.guess output. So we have to patch cputable too:

```
--- cputable-orig
+++ cputable-fix
@@ -34,3 +34,4 @@
 sh4    sh4            sh4
 sh4eb  sh4eb          sh4eb
 sparc  sparc          sparc(64)?
+e2k    e2k-unknown    e2k
```

Almost all packages use dpkg-architecture and get correct architecture identifier. If they don't, we fix it.

IX. Conclusion

This article is attempt to share experience in porting Debian on the architecture which is not supported by the community. General and specific for e2k architecture problems were described, as well as methods for their solutions. Authors hope that the article will be useful and interesting for developers, who support Debian on different architectures.

References

[1] Babayan B., "E2K Technology and Implementation", *Proceedings of the Euro-Par 2000 - Parallel Processing: 6th International*, Volume 1900/2000, pp. 18-21, January 2000.

[2] Dieffendorf. K., "The Russians Are Coming. Supercomputer Maker Elbrus Seeks to Join x86/IA-64 Melee" *Microprocessor Report*, Vol. 13, №2, pp. 1-7, February 15, 1999.

[3] Volkonskiy V., "Optimizing compilers for Elbrus-2000 (E2k) architecture", *4-th Workshop on EPIC Architectures and Compiler Technology*, 2005.

[4] http://www.edg.com/

[5] Hall J., "Distributed Compiling with distcc", *Linux Journal*, Issue 163, November 2007.