Статическая инфраструктура для сборки кросс-компилятора

А. А. Асонов[1], С. В. Самборский[2]

[1]НИЦ «Курчатовский институт» - НИИСИ, Москва, РФ, asonow@niisi.ras.ru;

[2] НИЦ «Курчатовский институт» - НИИСИ, Москва, РФ, sambor@niisi.msk.ru

Аннотация. В работе рассматривается проблема эксплуатации кросс-компилятора на базе GCC и бинарных утилит для отечественных микропроцессоров в условиях разнообразия аппаратных платформ и дистрибутивов ОС Linux. Показано, что использование разделяемых библиотек (прежде всего glibc) и зависящих от конкретного дистрибутива сборок приводит к множеству несовместимостей и усложняет сопровождение. Предложена статическая «самодостаточная» инфраструктура для сборки кросс-компилятора, включающая компилятор, бинарные утилиты и набор библиотек, собранные с использованием mus - альтернативной реализации стандартной библиотеки языка С. Описана многошаговая процедура раскрутки (bootstrapping), позволяющая получить конечный комплект инструментов, не использующий разделяемые библиотеки и минимально зависящий от параметров конкретной системы (кроме разрядности процессора и интерфейса ядра Linux). Обсуждаются ограничения, связанные с отказом от разделяемых библиотек (поддержка LTO, санитайзеров), и возможные направления расширения инфраструктуры за счет включения дополнительных утилит для сборки, тестирования и отладки.

Ключевые слова: кросс-компилятор, статическая линковка, инфраструктура сборки, GCC, бинарные утилиты, musl libc, glibc, отечественные микропроцессоры, Linux-дистрибутивы

1. Введение

При портировании и доработке для отечественных микропроцессоров и ОС реального времени кросс-компилятора на основе GCC [1] и бинарных утилит [2, 3] возникла проблема с тем, как обеспечить эксплуатацию этих утилит на разнообразной вычислительной технике.

Даже если ограничиться только операционной системой Linux на 32-битной архитектуре Intel (80386 и выше) и 64-битной архитектуре AMD (X86-64), компьютеры могут отличаться разрядностью процессора, установленными дистрибутивами ОС Linux, версиями этих дистрибутивов и ранее установленным программным обеспечением.

Можно точно задать конфигурацию компьютера для эксплуатации компилятора и бинарных утилит, точно указав версию дистрибутива, и потребовать, чтобы использовалась только эта конфигурация. Такой подход оказался не очень удобен, так как выбор дистрибутива ОС Linux и его версии часто определяется необходимым дополнительным программным обеспечением, опытом и предпочтениями пользователя или историческими причинами.

Использование для эксплуатации компилятора и бинарных утилит отдельного компьютера (пусть даже виртуального) крайне неудобно. Технологии контейнеризации, подобные docker, могут быть незнакомы пользователям, а также требуют достаточно свежего ядра системы Linux.

Собирать кросс-компилятор и бинарные утилиты по отдельности для использования на разных версиях всевозможных дистрибутивов ОС Linux – плохое решение, так как потребует много времени и соответствующих компьютеров (хотя бы виртуальных). А главное: на разработчиков возлагается ответственность за работоспособность каждого варианта сборки и, следовательно, требуется отдельное тестирование.

Более реалистичный подход – собирать ограниченное число вариантов поставок, для конкретных конфигураций, на которых работоспособность гарантируется, но при этом стремиться к тому, чтобы собранные кросс-компилятор и бинарные утилиты запускались и нормально работали на разнообразных (по возможности) компьютерах с операционной системой Linux.

2. Причины несовместимости

Причины, по которой собранные на одном компьютере программы не будут работать на другом компьютере, можно разделить на две основные группы: аппаратные несовместимости и программные различия.

Очевидная аппаратная несовместимость

возникает при попытке запустить 64-битные программы (скомпилированные для x86-64) на 32-битном процессоре. Впрочем, сейчас 32-битные процессоры остались в различном встроенном оборудовании, где не требуется запускать компилятор и бинарные утилиты. Настольных компьютеров и, тем более, серверов с 32-битными процессорами осталось очень мало.

Другая аппаратная проблема возникает с тем, что при сборке некоторых библиотек, на этапе конфигурирования изучается процессор компьютера, на котором идет сборка, для того чтобы использовать возможности данного процессора (обычно векторное расширение) с целью повышения производительности изготавливаемой библиотеки. В результате собранная с использованием этой библиотеки программа не будет работать на компьютере с процессором, не поддерживающим эту возможность. Именно это может случиться с библиотекой GMP, реализующей арифметику повышенной точности, необходимую для сборки компилятора GCC, подробнее можно прочитать в [4].

Для борьбы с такой несовместимостью нужно запретить настройку на конкретный процессор при сборке программного обеспечения. Во всех случаях следует указывать при конфигурации базовую архитектуры без всяких расширений и дополнений. Здесь следует иметь в виду, что вызванная этим потеря в производительности относится только к скорости компиляции, а не к производительности программ для отечественных микропроцессоров, скомпилированных этим компилятором.

Несовместимости, вызванные программными различиями, возможны разные, но для компилятора и бинарных утилит основная причина, препятствующая их установке на компьютеры с разной конфигурацией – использование разделяемых библиотек. В случае простого отсутствия необходимой разделяемой библиотеки ее можно дополнительно установить, если есть свободный доступ в интернет. Хуже ситуация, когда для нового программного обеспечения требуется версия разделяемой библиотеки более старая, чем уже установленная на компьютер. Одновременное использование нескольких версий одной библиотеки возможно, но часто требует ручной настройки.

Еще хуже, если на компьютер установлены разделяемые библиотеки, специфичные для конкретного дистрибутива ОС Linux. Тогда при сборке программного обеспечения на этом компьютере могут быть созданы исполняемые файлы, которые в принципе нельзя будет запустить на компьютерах с другими дистрибутивами.

Для того чтобы избежать этих проблем, желательно собирать программное обеспечение так, чтобы минимизировать зависимости от прочего программного обеспечения. Подобные «самодостаточные» программные пакеты должны содержать в себе все необходимые разделяемые библиотеки или не использовать их совсем.

Вариант без разделяемых библиотек выглядит предпочтительнее, поскольку выигрыш от собственных разделяемых библиотек – небольшая экономия дискового пространства и оперативной памяти, что не так актуально для современных компьютеров. При этом будет проигрыш в усложнении процесса запуска программы и потенциальных проблемах с тем, что динамический загрузчик не найдет нужную библиотеку.

Среди разных библиотек следует особенно выделить библиотеку libc, которая содержит runtime-поддержку языка C (ввод/вывод, динамическая память, работа со строками и еще очень много всего). Кроме этого, libc обеспечивает интерфейс с операционной системой, поэтому практически невозможно представить полезную программу, запускаемую в ОС Linux, которая бы не использовала прямо или косвенно функции из libc.

В большинстве дистрибутивов ОС Linux библиотека libc реализована в пакете glibc [5, 6] (и связанных с ним). Эта реализация содержит весьма продвинутую функциональность, но ценой большой сложности. При этом есть статичная (неразделяемая) версия libc (пакет glibc-static в дистрибутиве Fedora), но с неразделяемой версией libc много проблем, что признают авторы дистрибутива:

«*The glibc-static package contains the C library static libraries for -static linking. You don't need these, unless you link statically, which is highly discouraged*».

Компилятору и бинарным утилитам требуется не очень много функциональности libc, поэтому собрать их с неразделяемой версией libc из glibc-static удается, но с проблемами, которые к тому же зависят от версии библиотеки.

3. Альтернативы glibc

Существует много реализаций runtime-поддержки языка C кроме glibc. Некоторые альтернативные реализации libc ориентированы

на встроенные системы, ограничены по возможностям, зато крайне нетребовательны к ресурсам. Другие реализации полностью поддерживают стандартные возможности и могут быть полноценной заменой glibc, и действительно заменяют glibc в некоторых дистрибутивах ОС Linux.

Одной из таких полноценных альтернатив является пакет musl [7]. Данная библиотека доступна с 2011 года и продолжает развиваться, а в качестве одного из основных преимуществ заявлена качественная поддержка статической линковки, т. е. использования неразделяемой библиотеки (в документации musl охарактеризована как «static-linking-friendly»). Поэтому musl представляется подходящей заменой glibc для сборки самодостаточного программного обеспечения.

4. Что требуется

Поставлена цель: сделать так, чтобы можно удобно собирать кросс-компилятор и бинарные утилиты для отечественных микропроцессоров, не используя никаких разделяемых библиотек, получая тем самым «самодосточный» пакет без зависимостей. Самый удобный способ, чтобы не было путаницы с тем, какая именно библиотека использована: изготовить специальный комплект из компилятора (не кросс-компилятора!), бинарных утилит и всех необходимых библиотек. При этом, компилятор и утилиты должны быть настроены на использование по умолчанию «комплектных» библиотек. Тем самым исключаются недоразумения с ошибочно подключенным glibc.

Таким образом, задача сборки кросс-компилятора и кросс-утилит сводится к сборке (не «кросс») компилятора, утилит и библиотек. При этом, для того чтобы ими было удобно пользоваться на разных компьютерах, эту «инфраструктуру» также желательно собрать с неразделяемыми библиотеками.

5. Как сделано

Для того чтобы собрать самодостаточный комплект из компилятора, утилит и библиотек, была использована раскрутка («bootstrapping»), состоящая из последовательной сборки компиляторов, утилит и библиотек таким образом, чтобы последний вариант удовлетворял нашим требованиям.

Последовательные шаги:

1) Прежде всего копируются и распаковываются дистрибутивы GCC (компилятора), бинарных утилит, библиотеки musl и трех библиотек, необходимых для сборки GCC: GMP, mpfr и mpc.

2) Дальше компилируются только неразделяемые варианты библиотек GMP, mpfr и mpc и устанавливаются в первый временный каталог.

3) Затем собираются и устанавливаются бинарные утилиты во второй временный каталог. Далее собирается компилятор с использованием библиотек из первого временного каталога, при этом для изготовления своих библиотек и стартовых файлов (libgcc.a, crt1.o и т. п.) он использует бинарные утилиты из второго временного каталога, туда же он и устанавливается.

4) Далее еще один раз собираются бинарные утилиты уже при помощи бинарных утилит и компилятора из второго временного каталога и устанавливаются в третий временный каталог.

Потом изготавливается неразделяемая версия библиотек из пакета musl также при помощи бинарных утилит и компилятора из второго временного каталога, и устанавливается в третий временный каталог.

Еще раз собирается компилятор, уже при помощи компилятора из второго временного каталога, и устанавливается в третий временный каталог. При этом он собирается с библиотекой libc.a, установленной в третий временный каталог, т. е. уже не glibc, а musl. Поэтому новый компилятор конфигурируется так, что он собирается статически (без разделяемых библиотек).

5) Последний раз собираются бинарные утилиты и устанавливаются в окончательный каталог (где планируется эксплуатация инфраструктуры). Собираются они при помощи компилятора, бинарных утилит и библиотек (musl!) из третьего временного каталога.

С использованием компилятора и бинарных утилит из третьего временного каталога снова собираются библиотеки из пакетов musl, GMP, mpfr и mpc и устанавливаются в окончательный каталог.

Наконец собирается компилятор при помощи компилятора и бинарных утилит из третьего временного каталога, но библиотек из окончательного каталога. Компилятор устанавливается в окончательный каталог.

Все вышеописанное удобно оформить в виде единого рекурсивного make-файла, в котором используется параллельная сборка на всех доступных ядрах процессора. Если есть много оперативной памяти, то можно значительно ускорить процедуру, разместив все кроме окончательного каталога в оперативной памяти (используя файловую систему tmpfs). Осталось не забыть в конце удалить все временные каталоги (особенно, если они занимают место в

оперативной памяти).

6. Что получилось, не получилось и что еще можно сделать

Сборка всего установленного в окончательный каталог (и компилятора, и бинарных утилит) производится только с неразделяемыми библиотеками.

По-умолчанию сборка происходит для 32-битной (i686) или 64-битной (x86-64) архитектуры в зависимости от архитектуры операционной системы. Но можно также собирать на 64-битном компьютере инфраструктуру для сборки 32-битного программного обеспечения.

Возможно, часть шагов можно исключить, но так как вся описанная последовательность действий осуществляется только один раз, то нет необходимости ее сокращать. Более того, можно сделать еще один шаг, пересобрав все еще один раз с включенной оптимизацией (оптимизировать раньше – пустая трата времени). Также при последней сборке компилятора можно заказать дополнительные языки программирования, кроме C и C++, необходимых для пересборки компилятора.

Окончательный вариант инфраструктуры можно пополнить дополнительными утилитами. Цель – сделать так, что при установке на «чистую» машину было все необходимое не только для простой пересборки кросс-компилятора и бинарных утилит, но и для тестирования, отладки, модификации. Например, можно дополнительно собирать texinfo, isl, bison, flex, expect, tcl, dwarfdump, elfutils, gdb, diff, patch, automake, autoconf, libtool. Из всего этого наиболее актуален texinfo, так как если он отсутствует, то его необходимо в первую очередь пересобрать или установить собранный.

Сборка без разделяемых библиотек плохо совместима с некоторыми возможностями компилятора, например, lto («link time optimization»), а также с санитайзерами (верификация времени выполнения). Впрочем, технология lto не особенно популярна при изготовлении встроенного программного обеспечения.

С санитайзерами ситуация сложнее: во-первых, большая часть санитайзеров не может быть включена при кросс-компиляции, так как отсутствует поддержка на целевой платформе. Но сам кросс-компилятор (и кросс-утилиты) могли бы быть собраны с включенным тем или иным санитайзером. Кроме того, санитайзер неопределенного поведения, UBSan («Undefined Behavior Sanitizer») не требует никакой особенной поддержки. Тем самым желательно его включать при конфигурировании компилятора (как кросс-компилятора, так и компилятора для сборки кросс-компилятора).

7. Заключение

Можно собрать комплект из библиотек, бинарных утилит и компилятора для сборки кросс-компилятора и кросс-утилит без использования разделяемых библиотек. Причем сам этот комплект также не будет использовать разделяемых библиотек.

Тем самым исключаются все зависимости, кроме зависимости от аппаратуры (разрядность процессора) и интерфейса ядра ОС Linux.

Собственно, разделяемые библиотеки, и в первую очередь glibc, должны устранять зависимость от ядра, так как программы не делают системные вызовы сами, а вместо этого обращаются к библиотеке. Но похоже, что «лекарство хуже болезни»: если интерфейс ядра изменяется редко и при этом сохраняется поддержка старых программ, то glibc меняется существенно чаще.

«Публикация выполнена в рамках государственного задания НИЦ «Курчатовский институт» - НИИСИ» по теме FNEF-2024-0001».

Static Infrastructure for Building a Cross Compiler

A. A. Asonov, S. V. Samborskiy

Abstract. The paper addresses the problem of deploying a GCC- and binutils-based cross-compiler toolchain for domestic microprocessors in the context of a wide variety of hardware platforms and Linux distributions. It is shown that the use of shared libraries (primarily glibc) and distribution-specific builds leads to numerous incompatibilities and complicates maintenance. A static, self-contained infrastructure for building a cross-compiler is proposed, comprising the compiler, binary utilities, and a set of libraries built using musl – an alternative implementation of the C standard library. A multi-stage bootstrapping procedure is described that makes it possible to obtain a final toolchain which does not utilize shared libraries and only minimally dependent on the parameters of a specific system (apart from CPU word size and the Linux kernel interface). The limitations associated with abandoning shared libraries (LTO support, sanitizers) are discussed, as well as possible directions for extending the infrastructure by adding auxiliary tools for building, testing, and debugging.

Keywords: cross-compiler, static linking, build infrastructure, GCC, binutils, musl libc, glibc, domestic microprocessors, Linux distributions

Литература

1. GCC, the GNU Compiler Collection. https://gcc.gnu.org/ (дата обращения 08.11.2025).
2. GNU Binutils. https://www.gnu.org/software/binutils/ (дата обращения 08.11.2025).
3. gdb and binutils. https://sourceware.org/git/binutils-gdb.git (дата обращения 08.11.2025).
4. В. А. Галатенко, Г. Л. Левченкова, С. В. Самборский. Особенности сборки кросс-компилятора GCC и бинарных утилит. «Труды НИИСИ РАН», Т. 12 (2022), № 4, 43-49.
5. The GNU C Library. https://www.gnu.org/software/libc/ (дата обращения 08.11.2025).
6. The GNU C Library (glibc). https://sourceware.org/glibc/ (дата обращения 08.11.2025).
7. musl libc. https://musl.libc.org/ (дата обращения 08.11.2025).