Методы обеспечения переносимости ПО

Д. В. Силаков, А.В. Хорошилов
{silakov, khoroshilov}@ispras.ru

Аннотация. Статья посвящена проблеме переносимости приложений между программно-аппаратными платформами. Предлагается обзор подходов к решению этой проблеме, появившихся за время развития ИТ, а также анализ преимуществ и недостатков каждого из них. Рассматриваются области применения существующих решений.

1. Введение

Проблема переносимости приложений между различными программно-аппаратными платформами ненамного моложе собственно компьютерных программ. Еще в конце шестидесятых годов озабоченность некоторых сотрудников AT&T Labs проблемой переносимости ОС UNIX на новые аппаратные платформы привела к созданию языка Си. Темпы развития компьютерной индустрии таковы, что проблемы сорокалетней давности кажутся достаточно простыми и решаемыми, по сравнению с тем, что мы имеем сегодня. Стремительное развитие связанных с компьютерами отраслей приводит к постоянному появлению новых программно-аппаратных платформ, информационных систем, и т.п., в то время как устаревшие комплексы уходят в небытие.

Производители ПО, как правило, заинтересованы в быстром переносе своих продуктов на новые системы, чтобы захватить соответствующую долю рынка. Если приложение изначально проектировалось с оглядкой на возможность портирования, то этот процесс может оказаться существенно дешевле создания нового продукта. Будет проще и конечным пользователям, которые в новой системе увидят то же самое приложение, с которым работали раньше, что также способствует популярности продукта.

В некоторых случаях большое внимание переносимости приложений уделяется и производителями платформ, на которых эти приложения выполняются — например, одним из принципиальных аспектов при разработке IBM мэйнфрейма System/S370 в конце 60х-начале 70х годов было сохранение обратной совместимости с System/S360, с целью упростить миграцию приложений своих клиентов [1]. Обратная совместимость с

System/S360 сохранялась на протяжении всего жизненного цикла System/S370; более того, сохраняется она и в ее последователях — System/S390 и zSeries.

Однако далеко не все производители столь щепетильны, и далеко не все платформы могут похвастаться столь длительным сроком жизни. Нередко с рынка уходят не только сами системы, но и их создатели, так что труднодоступными становятся не только сами системы, но и все сведения об их архитектуре. Исчезновение тех или иных платформ приводит к появлению унаследованного ПО — программных продуктов, необходимых для функционирования той или иной организации, но требующих для работы устаревшей программно-аппаратной платформы. В случае выхода из строя аппаратного обеспечения может оказаться, что найти ему замену очень сложно, дорого, а иногда и попросту невозможно, так как устаревшая ОС не работает на современном оборудовании (либо по причине принципиальных отличий архитектуры, либо по более прозаическим причинам — например, ввиду отсутствия необходимых драйверов). Для многих предприятий задача переноса таких приложений в более современное окружение является крайне актуальной [2,3].

2. Примеры из современности

Несмотря на то, что о проблеме переносимости известно достаточно давно и ее решению посвящено множество работ и исследований, постоянно возникают ситуации, когда выясняется, что данному вопросу вовремя не было уделено должное внимание и это привело к неприятным последствиям. Рассмотрим несколько реальных примеров таких ситуаций.

В начале 2000-х годов компания Borland решила обратить свой взор на ОС Linux и выпустила среду разработки Kylix — аналог Delphi и C++ Builder. Такой шаг был изначально положительно оценен Linux-сообществом (даже несмотря на то, что Kylix не являлся продуктом с открытым исходным кодом, а бесплатной была только базовая версия системы) — в то время в этой ОС не было сравнимых по функциональности аналогов упомянутых программ. Однако в основе Kylix лежал исходный код соответствующих сред разработки для ОС Windows, а для запуска на Linux использовался эмулятор wine[4]. Как показала практика, такой прямолинейный перенос, как использование эмулятора, не привел к созданию конкурентноспособного продукта — довольно быстро выяснилось, что wine не является достаточно надежным, чтобы гарантировать его стабильность [5]. Разработчикам приходилось иметь дело как с ошибками в своих программах, так и с некорректным поведением эмулятора. Ввиду закрытости проекта сложно оценить, насколько затратен был бы перенос программ на использование «родных» библиотек Linux; но основываясь на том факте, что работа над Kylix была заморожена, можно предположить, что задача оказалась слишком ресурсоемка.

Другой пример недальновидного подхода к этому вопросу проявился в ходе организации проекта по разработке пакета свободного программного

обеспечения (СПО) для образовательных учреждений России. Практически все программное обеспечение, которое разрабатывалось по заказу Министерства образования РФ в последние годы было предназначено для работы исключительно на платформе Microsoft Windows. Поэтому при внедрении пакета СПО на основе операционной системы Linux б**о**льшая часть разработанных ранее образовательных программ оказалась недоступна, и только часть из них удавалось запустить с помощью эмулятора wine [4].

Схожие проблемы возникали и в стане разработчиков web-приложений. Известно несколько случаев, когда при разработке интернет-сервисов заказчик ограничивался требованием совместимости с браузером Internet Explorer, а через некоторое время под давлением клиентов был вынужден дорабатывать ПО для поддержки набирающего популярность Mozilla Firefox.

Например, на основе опроса пользователей приложения Tasktop о желаемых нововведениях, проведенного в 2008 году, выяснилось, что наиболее востребованными являются поддержка ОС Linux и браузера Firefox. Реализация этих свойств стала приоритетным направлением разработки, и была представлена пользователям уже в ноябре 2008 года, в Tasktop 1.3 [6]. Отметим, что добавление такого нетривиального свойства, как поддержка новой операционной системы, не заняло много времени, поскольку основная часть приложения написана на интерпретируемом языке Java, а виртуальные машины для исполнения этого кода существуют как в Windows, так и в Linux. Более того, разработчики Tasktop планируют портировать свой продукт и на MacOS — ввиду наличия в этой ОС виртуальной машины Java, такая задача также не представляется слишком сложной.

Приведенные примеры демонстрируют, что изначальная нацеленность на создание переносимого продукта позволяет достаточно безболезненно портировать его на платформы, поддержка которых изначально не планировалась (либо которых просто не существовало в момент начала разработки). Это позволяет производителям не оставаться в стороне от стремительного развития рынка и своевременно реагировать на запросы пользователей.

В то же время невнимание к проблеме переносимости может привести к различным негативным техническим, экономическим и политическим последствиям:

- проблемам с поддержкой ПО в долгосрочной перспективе;
- сокращению доступных рынков и недополучению прибыли;
- попаданию в зависимость от одного поставщика.

Как мы уже отметили, проблема известна давно, и за время эволюции программного обеспечения появилось достаточно много подходов к созданию приложений, переносимых между различными системами. Рассмотрим те из них, что представляются наиболее популярными.

3. Переиспользование бинарных файлов

Перенос приложения на новую программно-аппаратную платформу может пройти безболезненно для разработчиков, если старая и новая системы совместимы на бинарном уровне, то есть новая система поддерживает двоичный интерфейс приложений (Application Binary Interface, ABI), что позволяет использовать старые двоичные файлы приложения без каких-либо изменений. Можно выделить две основные составляющие ABI:

1. форматы исполняемых файлов и библиотек;
2. набор библиотек и их функций, предоставляемых системой.

Совместимость форматов является критичной для возможности использовать двоичные файлы без изменений. При этом формат файлов достаточно сильно связан с аппаратной частью платформы, на которой функционирует система - в силу технических причин (разный размер указателей, различный порядок нумерации разрядов и т.п.) в общем случае сложно достичь совместимости систем, работающих на различных архитектурах.

Что касается наборов библиотек и их функций, то их полной идентичности в различных системах ожидать трудно, однако во многих случаях пересечение множеств предоставляемых функций является достаточно большим.

Многие производители операционных систем в настоящее время заботятся об обратной совместимости своих продуктов на бинарном уровне — гарантируется, что приложение, работающее в некоторой версии ОС, будет работать без перекомпиляции в более новых версиях системы на той же аппаратной архитектуре.

В ряде случаев поддерживается возможность запуска приложений, написанных для той же системы, но на другой платформе — так, операционная система MacOS X, работающая на компьютерах Apple с процессорами Intel, использует динамический транслятор Rosetta для выполнения программ, предназначенных для машин с процессорами PowerPC [7]. Однако пользоваться такой возможностью следует с осторожностью. Во многих случаях совместимость обеспечивается за счет некоторого дополнительного слоя совместимости между системой и приложением, который может и не гарантировать полной совместимости — так, уже упомянутая Rosetta позволяет исполнять код для процессоров G3, G4 и AltiVec, но не для G5. Кроме того, дополнительный компонент системы является дополнительным потенциальным источником ошибок, а использование посредника в общем случае снижает производительность. Например, процессор Itanium способен выполнять код, созданный для платформы x86, однако производительность его в этом случае может уступать оригинальному x86 процессору с такой же тактовой частотой [8].

4. Переиспользование исходного кода

К сожалению, в большинстве случаев совместимость на бинарном уровне выполняется в рамках систем одного производителя, однако лишь немногие системы способны загружать исполняемые файлы, предназначенные для выполнения на платформах других поставщиков. Альтернативой использованию одних и тех же бинарных файлов явилось использование одного и того же исходного кода для сборки приложения на различных системах.

Исторически, возможность переноса исходного кода между различными платформами появилась не сразу. В те времена, когда программы писались на ассемблере конкретной аппаратной платформы, добиться компиляции приложения на системе с другой архитектурой было практически нереально. Существенной подвижкой в этом направлении стало создание высокоуровневых языков программирования, не привязанных к конкретной архитектуре и позволяющих использовать одни и те же конструкции на различных системах. Например, именно озабоченность некоторых сотрудников AT&T Labs проблемой переносимости ОС UNIX на новые аппаратные платформы привела к созданию языка Си.

Для обеспечения возможности использования одного и того же кода целевые системы должны предоставлять компиляторы для соответствующего языка программирования, вместе с необходимыми библиотеками. Естественно, что компиляторы для различных систем создаются различными производителями и могут достаточно сильно отличаться друг от друга. Такие отличия следует учитывать при написании кода.

Частично эта задача облегчается следованием международных стандартов, разработанных для многих языков программирования. Однако далеко не все компиляторы поддерживают стандарты в полном объеме (хотя, как правило, не поддерживаются некоторые специфические конструкции языка, в плане же предоставления библиотечных функций ситуация гораздо лучше). Другой проблемой является относительная узость стандартов — несмотря на наличие во многих из них перечня функций, которые должны предоставляться любой удовлетворяющей стандарту средой разработки, эти перечни не описывают существенную часть функциональности, которая могла бы быть полезна при создании приложений — например, функции графического интерфейса, либо функции работы с мультимедиа.

Помимо стандартов языков программирования, существуют стандарты, описывающие интерфейс прикладных программ (API, Application Programming Interface) — например, POSIX. Однако такие стандарты также, достаточно узки, и являются недостаточными для написания большинства приложений.

В случае, если необходимая приложению функциональность не охватывается ни одним из стандартов, могут быть использованы продукты сторонних

разработчиков, существующие на всех целевых системах и предоставляющие необходимые возможности. Такой продукт выступает в роли медиатора между приложением и операционной системой, скрывая от приложения процесс взаимодействия с последней. Характерным примером подобных продуктов являются кросс-платформенные библиотеки графического интерфейса пользователя — такие, как Qt, Gtk или wxWidgets. Реализация этих библиотек присутствует во всех основных операционных системах (FreeBSD, Linux, Windows, MacOS), а функции API, доступные программистам, практически идентичны для всех платформ. Кроме того, либеральность лицензий позволяет использовать эти библиотеки при создании достаточно большого спектра приложений.

Если же медиатора, способного удовлетворить нужды разработчиков, не нашлось, то может быть целесообразно создать свой собственный, который можно будет использовать во многих продуктах компании. Такой подход оправдан, если можно четко выделить функциональность, которую должен предоставлять медиатор, и отделить реализацию этой функциональности от остальных частей продукта. Так, разработчики Mozilla в свое время решили создать собственный набор библиотек поддержки различных стандартов безопасности (в результате появился набор Network Security Services, NSS), а также собственную кросс-платформенную библиотеку, реализующую достаточно низкоуровневые функции (работы с потоками, памятью, вводом-выводом и т.п.). Результатом второй инициативы стало создание библиотеки NSPR (NetScape Portable Runtime). Отметим, что поскольку и NSS, и NSPR, являются продуктами с открытым исходным кодом, их использование в настоящее время не ограничивается проектами, разрабатываемыми Mozilla Foundation.

5. Использование интерпретируемого кода

Еще одним подходом к обеспечению переносимости приложений является написание программного кода на интерпретируемых языках, использование которых не подразумевает создания исполняемых файлов в формате целевой операционной системы. Вместо этого интерпретатор последовательно считывает и выполняет инструкции непосредственно из текста программы.

Прямолинейная интерпретация достаточно неэффективна — у интерпретатора практически нет возможностей для оптимизации кода. Для повышения эффективности во многих языках (Java, Perl, Python, семейство .NET) исходный код сначала транслируется в некоторое промежуточное представление (байт-код), который уже подается на вход интерпретатору. Стадия получения байт-кода фактически является разновидностью компиляции, и при ее осуществлении могут выполняться различные оптимизирующие преобразования. Однако и в этом случае зачастую не удается достигнуть производительности, присущей «родным» для системы приложениям,

поскольку определенная часть времени тратится на разбор байт-кода и транслирование его в вид, подходящий для исполнения в целевой среде.

Большей производительности удается достигнуть при использовании компиляции “на лету” (Just In Time compilation), когда байт-код транслируется в машинный код во время работы программы. Однако разработчикам JIT-компиляторов также приходится идти на компромисс между временем, уходящим на оптимизацию, и эффективностью получаемого кода, в результате чего производительность приложений может уступать коду, создаваемому “обычными” компиляторами. Кроме того, использование данной технологии увеличивает потребление памяти приложением, поскольку оттранслированный код также хранится в оперативной памяти. Также следует иметь в виду, что эффективность реализации интерпретаторов, также как и JIT-компиляторов, на различных платформах может отличаться.

6. Использование эмуляторов ABI

Говоря о бинарной переносимости, мы уже упомянули, что в ряде случаев операционная система может обеспечивать бинарную совместимость с другой системой за счет дополнительного слоя совместимости. При этом большинство существующих реализаций ограничивается обеспечением совместимости в рамках систем одного производителя. Из других примеров можно выделить поддержку запуска исполнимых файлов Linux в системе FreeBSD. Сам Linux также имел поддержку запуска исполнимых файлов других UNIX-систем, хотя эта функциональность не оказалась востребованной.

В то же время существует много продуктов сторонних разработчиков, позволяющих загружать файлы других операционных систем путем использования транслятора, способного загружать файлы требуемого формата, преобразуя вызовы функций, осуществляемые внутри файла, в соответствующие вызовы текущей ОС (фактически, такой транслятор реализует ABI старой системы в новой системе).

В качестве примера можно привести wine [4], предназначенный для запуска Windows-приложений в Linux, а также cygwin [9], обеспечивающий переносимость в обратную сторону. При этом, например wine достаточно легко использовать как часть приложения, не полагаясь на его доступность в целевой системе.

Недостатком использования такого рода эмуляторов является потенциальная неполнота реализации интерфейса, необходимого приложению. Так, разработчики того же wine ориентируются на публично доступную информацию об API Windows (например, стандарт ECMA-234); так что если приложение использует какие-то недокументированные возможности этой ОС, то попытка его запуска в wine может оказаться неудачной.

Отметим, что для достижения высокой производительности программы-эмуляторы зачастую используют достаточно низкоуровневые способы взаимодействия с системой, что, в свою очередь, ограничивает их собственную переносимость. Так, тот же wine содержит достаточно много кода на языке ассемблера архитектуры x86. Соответственно, использование этого эмулятора возможно либо в непосредственно x86 системах, либо в системах, предоставляющих возможность запуска x86 приложений. Поэтому при стремлении охватить большое число различных программно-аппаратных платформ, может возникнуть необходимость разных эмуляторов на различных системах.

Кроме того, в определенных ситуациях прямолинейное использование эмуляторов по определению дает менее эффективные продукты, чем те, что используют «родные» средства целевой системы. В качестве примера можно привести уже упоминавшуюся ранее среду разработки Borland Kylix для Linux, основанную на использовании эмулятора wine. Косвенно использование эмулятора в этом случае означает использование тех же методик создания программ, которые применяются в исходной системе — ОС Windows. Однако, как показала практика, эмуляция этих методик не выдерживает конкуренции с аналогичными средствами ОС Linux — в частности, с компилятором GCC (прежде всего, в плане производительности программ, *получаемых* с помощью среды Kylix).

7. Виртуализация

Альтернативой эмуляции ABI некоторой системы является запуск копии такой системы внутри основной ОС, с использованием программ, эмулирующих аппаратное обеспечение — виртуальных машин. На такой машине устанавливается операционная система и другое окружение, необходимое приложению, а само приложение запускается уже в родной для него среде.

Возможность запуска приложения на виртуальной машине зависит, в основном, от возможностей самой машины, нежели от разработчиков приложения. Тем не менее, программы, работающие с аппаратурой напрямую, могут встретить определенные трудности, поскольку им будет предоставлен доступ к устройствам виртуальной машины, а не непосредственно компьютера. Такая особенность ограничивает, например, возможность работы с графическими ускорителями изнутри виртуальных машин.

В общем случае использование виртуальной машины достаточно ресурсоемко — ведь помимо собственно приложения, ресурсы компьютера потребляются самой машиной, а также работающими внутри нее программами, необходимыми для функционирования приложения (например, операционной системой). Поэтому выигрыш в производительности достигается, как правило, только в случае запуска машин, эмулирующих достаточно маломощные платформы на более производительных системах.

Проблеме производительности виртуальных машин уделялось много внимания еще в 70-е годы. Впервые требования, которым должна удовлетворять аппаратная архитектура машины, чтобы на ней можно было эффективно реализовать виртуализацию, были сформулированы Попеком и Голдбергом в 1974 году [10]. Основное требование сводится к разделению всех инструкций по разным уровням привилегий; при этом все инструкции, способные изменить состояние ресурсов виртуальной машины, а также инструкции, поведение которых зависит от конфигурации этих ресурсов, должны быть привилегированными. Монитор виртуальных машин (Virtual Machine Monitor, VMM — программная прослойка, занимающаяся распределением физических ресурсов между работающими виртуальными машинами), сам работая с наивысшими привилегиями, может перехватывать эти инструкции и эмулировать их поведение в соответствии со своими потребностями. Все непривилегированные инструкции должны выполняться непосредственно на аппаратуре. Поскольку доля привилегированных инструкций, как правило, невелика, то и затраты на их эмуляцию будут малы.

Несмотря на то, что работа Голдберга и Попека посвящена машинам третьего поколения (IBM 360, Honeywell 6000, DEC PDP-10), сформулированные в ней требования справедливы и сегодня. Однако, для большинства современных архитектур приведенное выше требование не выполняется — существуют непривилегированные инструкции, влияющие на конфигурацию ресурсов либо зависящие от нее. В частности, для «классической» архитектуры x86 к числу таких инструкций относятся такие популярные инструкции, как push/pop, call, jmp и другие (более подробно данная проблема рассмотрена в [11]).

Безусловно, построение виртуальной машины возможно и при наличии таких инструкций. Существуют подходы по определению и перехвату необходимых инструкций в процессе работы программы; по такому принципу работают популярные продукты типа VirtualBox [12] и VMWare [13], старающиеся напрямую выполнять все инструкции, для которых это возможно. Тем не менее, необходимость дополнительного отслеживания выполняемых инструкций может замедлить производительность программ внутри виртуальной машины по сравнению с «живой» системой.

Отметим, что осознание разработчиками важности виртуализации привело к появлению расширений от Intel (Intel Virtualization Technology [14]) и AMD (AMD Virtualization [15]) для ее поддержки на платформах x86 и x86-64, которые позволяют либо вовсе избавится, либо существенно снизить число перехватываемых и эмулируемых инструкций.

Альтернативным методом борьбы с «вредоносными» инструкциями является паравиртуализация, основанная на внесении изменений в гостевую операционную систему перед ее запуском в виртуальной среде; известным примером машин, работающих по принципу паравиртуализации, является Xen [16]. Однако в реальной жизни такая модификация не всегда доступна

конечным пользователям. Существуют и гибридные подходы — например, проект vBlades для Itanium [17].

Стоит обратить внимание и на экономическую составляющую использования виртуальной машины конечным пользователям — в ряде случаев стоимость операционной системы и компонентов окружения, которые необходимо установить, может быть довольна значительна. Поэтому достаточно редки ситуации, когда производители сами советуют использовать виртуальные машины для запуска своих программ в системах, не поддерживаемых напрямую. Примерами исключений являются различные специфические системы, безопасное функционирование которых требует запуска в изолированной среде. Например, обучающая система Linuxgym [18] распространяет клиентскую часть в виде образа VMware, содержащего Ubuntu Linux с предустановленными необходимыми компонентами.

8. Использование Web-технологий

Становящиеся с каждым годом все популярнее web-технологии также могут быть использованы для повышения переносимости программных продуктов. Можно выделить два основных способа построения приложений, использующих эти технологии:

- создание полностью локального приложения, располагающегося на машине пользователя, и использующего web-браузер для взаимодействия с ним;
- реализация приложения в виде сервиса; при этом серверная и клиентская части приложения могут быть разнесены на разные машины.

Такое разделение довольно условно, поскольку в первом случае приложение также, как правило, имеет серверную и клиентскую части, и может допускать подключение клиентов с других машин. Например, подобным образом устроен Web-интерфейс сервера печати CUPS (Common Unix Printing System), который может быть использован для настройки принтеров как на локальной, так и удаленных машинах.

В любом случае, использование web-браузера и сопутствующих инструментов облегчает задачу разработки графического интерфейса пользователя. Программа может либо создавать HTML-представление интерфейса, непосредственно интерпретируемое браузером, либо использовать технологии типа Flash, Silverlight и JavaFX, позволяющие создавать интерактивные файлы, для воспроизведения которых достаточно наличия соответствующего проигрывателя на целевой платформе. Зачастую такой проигрыватель встраивается в браузер как расширение.

Однако хотелось бы подчеркнуть, что во многих ситуациях требования к среде, где функционирует серверная части приложения, существенно более

строгие, чем требования к клиентской системе. Последние зачастую сводятся к требованию наличия браузера, поддерживающего определенные возможности. Для серверной же части может требоваться вполне конкретная программно-аппаратная платформа.

Использование web-браузера для отрисовки графического интерфейса имеет как преимущества, так и недостатки. К преимуществам, помимо упомянутого выше снижения затрат на разработку, стоит отнести возможность настройки интерфейса пользователем по своему вкусу — изменением шрифтов, масштаба страницы и т.п. Основной недостаток подхода является следствием его достоинств — в разных браузерах одна и та же страница может отображаться по-разному — как вследствие пользовательских настроек, так и из-за особенностей конкретного браузера. Безусловно, существуют различные стандарты на HTML, CSS, JavaScript и прочие элементы, используемые при отображении web-документов, однако разные браузеры соответствуют этим стандартам в разной степени [19]. Поэтому разработчикам приходится либо проводить тестирование на различных браузерах [20], либо ограничиваться поддержкой только некоторых из них. Для автоматизации процесса тестирования существуют различные свободно доступные инструменты — например, сервис BrowserShots [21] позволяет получить снимки экрана с видом заданной страницы в различных браузерах (на данный момент доступно более 80) в Linux, Mac OS и Windows.

Также отметим, необходимость создания больших распределенных систем привела к созданию архитектур, более сложных, чем клиент-серверная, и подразумевающих взаимодействие множества компонентов различной структуры. Исследования проблем разработки сложных программных комплексов привели к разработке парадигм, подразумевающих разбиение системы на отдельные компоненты, которые взаимодействуют друг с другом по строго определенным протоколам. При таком подходе каждый компонент не только может работать на отдельной машине, но также разрабатываться независимо от других. При этом (в идеале) процесс переноса компонента на другую платформу либо замена его на альтернативную реализацию никак не затрагивает других частей системы.

К одной из первых попыток описания механизма взаимодействия компонентов распределенной системы можно отнести спецификацию CORBA (Common Object Request Broker Architecture), разработанную консорциумом OMG [22]. Однако CORBA в силу ряда причин не снискала большой популярности, и в настоящее время гораздо больший интерес проявляется к web-сервисам, использующим протоколы обмена сообщениями на базе XML. При проектировании сложных распределенных программных комплексов используется парадигма сервисно-ориентированной архитектуры (Service-oriented architecture, SOA [23]); при этом программные комплексы часто реализуются как набор web-сервисов.

9. Заключение

Если невнимание к проблеме переносимости приводит к негативным последствиям и существует множество путей по ее решению, то возникает вопрос: так почему же ИТ индустрия не переориентируется на разработку переносимого ПО? Несложно догадаться, что разработка переносимого ПО имеет свои недостатки.

Среди рассмотренных видов переносимости приложений очень привлекательным с точки зрения разработчиков является перенос непосредственно бинарных файлов на новую систему, позволяющий при относительно небольших затратах (в основном уходящих на тестирование) получить на новой системе приложение, имеющее всю необходимую функциональность. При этом потери в производительности если и возникают, то совсем небольшие.

Однако для любой ОС число платформ, совместимых с ней на бинарном уровне, достаточно невелико. Использование эмуляторов может расширить их круг, но эмулятор — дополнительный потенциальный источник ошибок, который при этом может и не предоставлять всех необходимых функций.

Потенциально больший охват дает переносимость исходного кода. Сложность портирования в этом случае может варьироваться в зависимости от того, насколько такая возможность учитывалась при разработке приложения; полезной с этой точки зрения является ориентация на различные интерфейсные стандарты, регламентирующие взаимодействие приложения с окружающей средой. Но существующие стандарты охватывают достаточно небольшую функциональность; в ряде случаев может помочь использование кросс-платформенных библиотек, другой же альтернативой является использование интерпретируемых языков.

Спецификации таких языков не привязаны к конкретной платформе и можно полагаться на то, что интерпретаторы на разных системах поддерживают один и тот же набор функций. Среди недостатков подхода можно выделить меньшую производительность по сравнению с бинарным кодом.

Архитектура SOA затрагивает более сложную проблему организации сложных программных комплексов, предлагая строить их в виде набора достаточно изолированных компонентов, каждый из которых может работать на своей собственной платформе и в случае необходимости может быть перенесен на другую (либо заменен на альтернативную реализацию).

Использование виртуальных машин также не требует больших усилий со стороны разработчиков ПО, хотя этот способ достаточно накладен, как в смысле производительности, так и ввиду необходимости иметь лицензии на все используемые операционные системы. Применение виртуализации оправдано в тех случаях, когда перенос приложения каким-то другим способом представляется экономически неэффективным. В частности, это относится ко многим унаследованным системам, для которых портирование

на новую платформу означало бы практически полное переписывание приложения.

В таблице 1 приведены примеры использования различных подходов к обеспечению переносимости ПО, которые используются разработчикам для создания программ, функционирующих как в Windows, так и в Linux. Интересно отметить подход Google, который не побоялся положиться на слой эмуляции wine для запуска Google Picasa в ОС Linux, несмотря на практически полное отсутствие успешных примеров крупных приложений, официально использующих такой метод. Подход с использованием библиотек-медиаторов более традиционен и применяется не один десяток лет.

Подход	Примеры
Ориентация на стандарты	Ряд утилит GNU (tar, wget и др.), написанные с ориентацией на POSIX
Медиаторы сторонних разработчиков	Skype — использование библиотеки Qt для реализации GUI
Собственные медиаторы	Mozilla Firefox — использование собственных кросс-платформенных библиотек NSS (Network Security Services) для поддержки различных стандартов безопасности
Эмуляция	Google Picasa — применение wine для работы в Linux
Виртуализация	Linuxgym — клиент распространяется в виде образа VMware

Таблица 1. Примеры использования различных подходов к обеспечению функционирования приложений как в Windows, так и в Linux.

Вопрос обеспечения переносимости следует рассматривать в самом начале проекта, на стадии проектирования и выбора технологий и инструментов, которые будут использованы при его реализации. К сожалению, вряд ли можно сформулировать универсальное правило выбора средств для увеличения переносимости – такой выбор сильно зависит от конкретных требований, предъявляемых к приложению. На взгляд авторов, в настоящее время потенциально наибольший охват дает использование интерпретируемых языков – многие интерпретаторы работают на достаточно большом числе программно-аппаратных платформ. Естественно, использование таких языков имеет свои недостатки, и для ряда приложений может оказаться неприемлемым.

Следующим по масштабу «охвата», на наш взгляд, идет использование кросс-платформенных библиотек и других медиаторов. Однако медиаторов, реализующих нужную функциональность, может и не существовать, а разработка

собственных – оказаться достаточно трудоемким процессом. Тем не менее, даже при отказе от поддержки нескольких систем ввиду отсутствия средств и при ориентации на одну конкретную платформу, стоит строить архитектуру приложения таким образом, чтобы по возможности отделить части, использующие специфические для данной платформы интерфейсы и сервисы.

Подводя итоги, отметим, что возможны ситуации, когда отказ от обеспечения переносимости разрабатываемого ПО оправдан; например, с достаточно большой долей вероятности такой выбор будет выгоден в краткосрочной перспективе. Однако при создании продуктов, которые планируется поддерживать в течении достаточно длительного периода, обеспечение переносимости может оказаться одним из ключевых факторов успеха. Повышение мобильности приложения в общем случае всегда связано с увеличением расходов на его разработку; однако чем раньше об этой проблеме задумаются разработчики и архитекторы, тем меньше будет стоимость переноса приложения на новую платформу.

Литература

[1] James D. Mooney. "Bringing Portability to the Software Process". // Technical Report TR 97-1, Dept. of Statistics and Computer Science, West Virginia University, Morgantown WV, 1997.

[2] Ian Sommerville. Software Engineering, 8th Edition, Addison Wesley, 2006.

[3] Wikipedia, the free encyclopedia. IBM System/370. http://en.wikipedia.org/wiki/System/370

[4] Wine - Open Source implementation of the Windows API. http://www.winehq.org/

[5] Borland's Net Loss Desaster. // Архив дискуссионного листа «borland.public.delphi.non-technical». Август 2005. http://coding.derkeiler.com/pdf/Archive/Delphi/borland.public.delphi.non-technical/2005-08/msg01903.pdf

[6] Ryan Slobojan. Tasktop 1.3: Support Added for Firefox and Linux. // InfoQueue News. Nov 12, 2008. http://www.infoq.com/news/2008/11/tasktop-13

[7] Apple Rosetta. http://www.apple.com/rosetta/

[8] Андрей Ященко. 64-х битные процессоры. // 3DNews, 7 августа 2003. http://www.3dnews.ru/cpu/cpu-64/index2.htm

[9] Cygwin - Linux-like environment for Windows. http://www.cygwin.com/

[10] Gerald J. Popek, Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. // Communications of the ACM, Volume 17, Issue 7, July 1974, pp. 412-421.

[11] Крис Касперски. Аппаратная виртуализация или эмуляция "без тормозов". http://www.insidepro.com/kk/159/159r.shtml

[12] Virtual Box. http://www.virtualbox.org

[13] VMware. http://www.vmware.com

[14] Matias Zabaljauregui. Hardware Assisted Virtualization. Intel Virtualization Technology. Buenos Aires, June 2008. http://linux.linti.unlp.edu.ar/kernel/wiki/images/f/f1/Vtx.pdf

[15] AMD Virtualization. http://www.amd.com/us-en/0,,3715_15781,00.html

[16] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebarger, Ian Pratt and Andrew Warfield. Xen and the Art of Virtualization. // Proceedings of the 19th Symposium on Operating Systems Principles, October 2003.
[17] Daniel J. Magenheimer, Thomas W. Christian. vBlades: Optimized Paravirtualization for the Itanium Processor Family. // Proceedings of the 3rd Virtual Machine Research & Technology Symposium. May, 2004. pp 73–82, USENIX.
[18] Linuxgym Teaching System. http://www.linuxgym.com/
[19] Tom Dahm. Browser Compatibility Tutorial. http://www.netmechanic.com/products/Browser-Tutorial.shtml
[20] Browser Tests, Services and Compatibility Test Suites. // Smashing Magazine, October 2007. http://www.smashingmagazine.com/2007/10/02/browser-tests-services-and-compatibility-test-suites/
[21] http://browsershots.org/
[22] OMG CORBA 3.0. http://www.omg.org/ technology/documents/formal/corba_2.htm
[23] Ben Margolis, Joseph Sharpe. SOA for the Business Developer: Concepts, BPEL, and SCA. MC Press, 2007.